# `super`-Charging Object-Oriented Programming Through Precise Typing of Open Recursion

**Andong Fan**
The Hong Kong University of Science and Technology (HKUST), Hong Kong, China

**Lionel Parreaux**
The Hong Kong University of Science and Technology (HKUST), Hong Kong, China

## Abstract

We present a new variation of object-oriented programming built around three simple and orthogonal constructs: *classes* for storing object state, *interfaces* for expressing object types, and *mixins* for reusing and overriding implementations. We show that the latter can be made uniquely expressive by leveraging a novel feature that we call *precisely-typed open recursion*. This features uses "this" and "super" annotations to express the requirements of any given partial method implementation on the types of respectively the current object and the inherited definitions. Crucially, the fact that mixins do *not* introduce types nor subtyping relationships means they can be composed even when the overriding and overridden methods have incomparable types. Together with advanced type inference and structural typing support provided by the MLscript programming language, we show that this enables an elegant and powerful solution to the Expression Problem.

**2012 ACM Subject Classification** Software and its engineering → Object oriented languages

**Keywords and phrases** Object-Oriented Programming, the Expression Problem, Open Recursion

**Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2023.11

**Related Version** *Extended Version*: `https://lptk.github.io/superoop-paper`

**Supplementary Material** *Software (ECOOP 2023 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.9.2.22`
*Software (Open-source implementation)*: `https://github.com/hkust-taco/superoop` archived at `swh:1:dir:7446abcf043f3546fae3ebce3efd85c07c70afa3`
*Software (Online demonstration)*: `https://hkust-taco.github.io/superoop`

**Acknowledgements** We thank the anonymous reviewers, Yaozhu Sun, and Marco Servetto for their helpful comments, as well as Cunyuan Gao for his help with the implementation. This work follows up on concepts previously presented by the first author as a research abstract [10].

## 1 Introduction

Every object-oriented programming (OOP) developer regularly uses the **`super`** keyword to access overridden definitions from inherited classes. Yet, this keyword has received relatively little attention in previous OOP literature and has been conspicuously absent from most previous research, with few exceptions [17]. This may be due to the assumption that **`super`**-calls can be resolved statically and are thus a mere syntactic convenience that is easily desugared into traditional core OOP features [2]. In this paper, we propose to challenge this assumption: noting that **`super`** is in fact *late-bound* in mixin-composition systems,[1] we describe an OOP approach which assigns *precise types* to **`super`**-calls to reflect the "open" nature of this late binding. Consider the following prototypical `Point` example class:

```
class Point(x: Int, y: Int)
```

[1] **`super`** is bound at the time the mixin method where it appears is composed into a class, which can happen as late as runtime in many mixin-composition languages.

This class simply defines two coordinates `x` and `y` as immutable fields.

Suppose we want to define a comparison function that works on points. We place this definition in a *mixin* declaration, for reasons that shall soon become clear:

```
mixin ComparePoint {
  fun compare(lhs: Point, rhs: Point): Bool =
    lhs.x == rhs.x and lhs.y == rhs.y }
```

Now suppose we want to compare colored points, but we would like colored comparison to be generally specified, so that it can be directly reused with other things than points. This can be done using the following combination of interface and mixin (`Base` is a type parameter):

```
interface Colored { color: String }
```

```
mixin CompareColored[Base] {
  super: { compare: (Base, Base) → Bool }
  fun compare(lhs: Base & Colored, rhs: Base & Colored): Bool =
    super.compare(lhs, rhs) && lhs.color.equals(rhs.color) }
```

We define an interface specifying that a `Colored` object should contain a `color` method or field of type `String`. We also define the `CompareColored` mixin, which implements a comparison method *based on* an assumed existing comparison method, inherited from an unknown parent implementation and referred to through **`super`**. The `Base` type parameter denotes the type compared by that unknown parent implementation; it is needed in order to leave the mixin *open-ended*, i.e., to allow mixing it with arbitrary parent implementations. Notice that the type of `compare` in `CompareColored` is *different* from the one specified in the **`super`** annotation, and is in particular not a subtype of it: the version defined in `CompareColored` takes parameters of *more precise* type `Base & Colored`, where `&` is an intersection type constructor, meaning that each parameter should be *both* a `Base` *and* a `Colored`. This difference is a crucial ingredient in our precisely-typed open recursion approach.

We now define `ColoredPoint` and place its comparison implementation in a module:[2]

```
class ColoredPoint(x: Int, y: Int, color: String)
    extends Point(x, y) implements Colored
```

```
module CompareColoredPoint extends ComparePoint, CompareColored[Point]
```

`CompareColoredPoint` did not need to define its own comparison method – that method was *composed automatically* by inheriting from the `ComparePoint` and `CompareColored` mixins, the latter using the correct `Point` base type argument. Note that mixins on the right override those on the left. The signature of `CompareColoredPoint`'s `compare` method, which allows passing in colored points, is:

```
CompareColoredPoint.compare: (Point & Colored, Point & Colored) → Bool
```

which is *not* a subtype of `ComparePoint`'s `compare` method. This is fine because mixins in our approach do not introduce types, and there is thus no subtyping relationship between `CompareColoredPoint` and `ComparePoint`, which is reminiscent of Cook et al. famous assertion that *inheritance is not subtyping* [7].

[2] A module declares a class with a singleton instance, similar to Scala's **`object`**.

Now imagine we want to deal with "*nested*" objects, which are objects that may optionally have a parent.[3] We can similarly define a comparison mixin for nested objects as follows:

```
interface Nested[A] { parent: Option[A] }

mixin CompareNested[Base, Final] {
  super: { compare: (Base,  Base)  → Bool }
  this:  { compare: (Final, Final) → Bool }

  fun compare(lhs: Base & Nested[Final], rhs: Base & Nested[Final]): Bool =
    super.compare(lhs, rhs) &&
      if lhs.parent is Some(p)
        then rhs.parent is Some(q) and this.compare(p, q)
        else rhs.parent is None
}
```

In this variant, we additionally use a `this` refinement, which specifies the *eventual* types of the methods the current object should support, after all inheritance and overriding is performed. The reason we use `this` and not `super` in the recursive `this.compare(p, q)` call is that we should take into account that `p` and `q` *themselves* may be nested points!

Finally, it is possible to compare points that are *both* nested *and* colored by directly composing the corresponding implementations:

```
class MyPoint(x: Int, y: Int, color: String, parent: Option[MyPoint])
    extends Point implements Colored, Nested[MyPoint]
```

```
module CompareMyPoint extends ComparePoint, CompareColored[Point],
                              CompareNested[Point & Colored, MyPoint]
```

Or alternatively, in a different order:

```
module CompareMyPoint extends ComparePoint, CompareNested[Point, MyPoint],
                              CompareColored[Point & Nested[MyPoint]]
```

Mixin composition order is meaningful because it determines overriding order; moreover, in our approach, the types of methods may change through overriding – here, notice how we pass different type arguments to `CompareColored` and `CompareNested` in each version.

To support this idea of precisely-typed mixin composition, we present the **SuperOOP** system, a simple yet uniquely expressive core description of OOP built around three orthogonal concepts: *classes* for storing object state, *interfaces* for expressing object types, and *mixins* for reusing and overriding implementations.[4] Notably, we only support inheriting from interfaces and mixins, not from classes.[5] We show that these simple, orthogonal concepts are sufficient to explain the usual features of object-oriented programming languages, including those with complicated multiple-inheritance disciplines, like Scala's trait composition approach.

We also describe how the ideas of SuperOOP can be integrated into MLscript, a nascent ML-inspired programming language with structural types and advanced type inference, based on the recently proposed MLstruct type system [26]. Using this approach, all the types can

[3] `Option[A]` is defined as the usual algebraic data type, with cases `Some[A](value: A)` and `None`.

[4] Such separation of concerns was already proposed by previous authors, such as Bettini et al. [2] and Damiani et al. [8], but the systems they developed did not support overriding and open recursion, which is the *raison d'être* of our approach.

[5] We see in Section 4.1 that the `Point` class inheritance example seen above can be desugared into our core $\lambda^{\mathsf{super}}$ calculus through interface inheritance and without requiring class inheritance.

be inferred automatically as long as they do not involve first-class polymorphism (which requires explicit annotations). For instance, in MLscript, the `CompareColored` mixin shown above could also be written as the more lightweight:

```
mixin CompareColored {
  fun compare(lhs, rhs) =
    super.compare(lhs, rhs) && lhs.color.equals(rhs.color) }
```

for which our compiler infers the following *mixin signature*:

```
mixin CompareColored: ∀ 'A1 'A2 'B . {
  super: { compare: ('A1, 'A2) → Bool }
  compare: ('A1 & {color: {equals: 'B → Bool}}, 'A2 & {color: 'B}) → Bool}
```

Our specific contributions are summarized as follows:

- We explain the general ideas of SuperOOP in the context of the structurally-typed MLscript programming language, and how it allows solving interesting problems simply and elegantly, including the Expression Problem and derivatives (Section 2). SuperOOP mixins improve on the state of the art by allowing precise typing of open recursion, which to the best of our knowledge was never proposed before.
- We formalize the core concepts of SuperOOP, including its precisely-typed mixin inheritance mechanism, in a declarative type system called $\lambda^{\mathsf{super}}$. We use big-step semantics to closely reflect a real implementation and prove the soundness of $\lambda^{\mathsf{super}}$ through the preservation and coverage properties (Section 3).
- We discuss the expressiveness and limitations of the presented design of SuperOOP as well as its implementation. We present several important approaches from previous literature on the topic of inheritance and the Expression Problem, and explain how these approaches compare to SuperOOP in detail (Section 4).
- We provide an implementation of MLscript/SuperOOP which demonstrates how type inference can be used to type check concise mixin and class definitions. Both the open-source version and the archived artifact with documentation are available. A demo of this implementation is included in the supplementary material of this paper.[6]

## 2 Motivation

In this section, we introduce a motivating example in MLscript "**`super`**-charged" by our OOP approach in more detail.

### The Expression Problem and Extensible Variants

In modular programming, the *Expression Problem* (EP) describes the dilemma posed by the modular extension for both data types and their operations in object-oriented and functional programming. There are many ways of tackling this problem, but one of the most straightforward is to rely on some notion of *extensible variants*, as done by Garrigue [16] with OCaml's polymorphic variants. The general idea of extensible variants is that they are similar to algebraic data types (a.k.a. variants) except that one is able to specify which data type cases are allowed in a given type, and moreover one is able to add new data type cases after the fact.

[6] The demo is also available at: `https://hkust-taco.github.io/superoop/`.

MLscript supports a simple form of extensible variants implemented through *subtyping and structural types*. In this section, we see how the combination of this feature and SuperOOP's precise typing of open recursion can achieve what we believe is one of the simplest and cleanest solutions to the expression problem so far.

## A Quick Look at MLscript

We first take a quick look at MLscript's basic language features that enable a form of extensible variants and serve as key ingredients in our solution to the Expression Problem.

**Basic data type classes.** Consider the following MLscript class definitions which encode a very minimal expression language that we will later extend in several directions.

```
class Lit(value: Int)
class Add[T](lhs: T, rhs: T)
```

The `Lit` class represents integer literals and the `Add` class represents addition. Note that the types of `Add`'s value parameter are polymorphic, meaning that they can be chosen arbitrarily. We will see that the ability to leave the types of subexpressions open is crucial to the extensibility of our approach.

**Union types.** Based on these class definitions, we can construct types such as:

```
type LitOrAddLit = Lit | Add[Lit]
```

where '`|`' is called a *union type* constructor. `LitOrAddLit` represents the type of an expression that is *either* an integer literal *or* an addition between two integer literals.

**Equirecursive types.** More interestingly, we can define the type of arbitrary expressions in our little language as:

```
type SimpleExpr = Lit | Add[SimpleExpr]
```

Notice that this type is *equirecursive*, meaning that `SimpleExpr` is *equivalent* to its unrolling `Lit | Add[SimpleExpr]`. This is quite convenient in the context of structural typing, and it allows us to have subtyping relationships (denoted as '`T1 <: T2`', meaning that `T1` is a subtype of `T2`) such as `LitOrAddLit <: SimpleExpr`. An equivalent way of specifying `SimpleExpr` without having to introduce a type declaration is through MLscript's '**`as`**' binder (similar to '**`as`**' in languages like OCaml), as in '`Lit | Add['a] as 'a`' (where '**`as`**' has least precedence).

**Evaluation.** To use values in our small expression language, we define an `eval` recursive function:

```
fun eval(e) = if e is
  Lit(n)        then n
  Add(lhs, rhs) then eval(lhs) + eval(rhs)
```

This function uses MLscript's syntax for pattern matching, which extends the traditional **`if-then-else`** syntactic form with multi-way-**`if`**-style functionality and destructuring through an '**`is`**' keyword [25]. The type of this function is inferred by MLscript to be:

```
eval: (Lit | Add['a] as 'a) → Int
```

**Default cases and constructor difference.** It is quite instructive to consider what happens when default cases are used in MLscript, as in:

```
fun eval2(e) =
  if e is
    Lit(n)        then n
    Add(lhs, rhs) then eval2(lhs) + eval2(rhs)
  else e
```

In this case, the type inferred is

```
eval2: (Lit | Add['a] | 'b\Lit\Add as 'a) → (Int | 'b)
```

Above, '\' is a *constructor difference* type operator,[7] which is used to *remove* concrete class type constructors from a given type (here `'b`). This type operator applies incrementally, as its left-hand side becomes concretely known upon type instantiation. For instance, after instantiating the type variable `'b` to, say, `Add[Int] | Bool` in the type above, `'b\Lit\Add` becomes `(Add[Int] | Bool)\Lit\Add`, which is equivalent to just `Bool`. Since all negative occurrences of `'b` (here there is only one) are subject to this constructor difference, passing values for `'b` which are of the `Lit` or `Add` forms is effectively prevented, which ensures type safety[8] [26]. On the other hand, any other type constructor is allowed, for example, we could call `eval2(true)`, with inferred result type `Int | Bool`.

## Open Recursion in MLscript with SuperOOP Mixins

Now let us consider putting our original evaluation function inside of a **`mixin`**, in order to enable future extensions. To make the recursion of evaluation *open*, we now recurse through method calls of the form '`this.eval`' (here '**`this`**' is the class instance to be late-bound) instead of a direct `eval` recursive function call:

```
mixin EvalBase {
  fun eval(e) = if e is
    Lit(n)        then n
    Add(lhs, rhs) then this.eval(lhs) + this.eval(rhs) }
```

The type signature inferred for that mixin definition is the following:

```
mixin EvalBase: ∀ 'A. {
  this: { eval: ('A) → Int }
  eval: (Lit | Add['A]) → Int
}
```

Above, `'A` is a *mixin-level* type variable,[9] meaning that it must be instantiated to a specific type each time the mixin is inherited as part of a class. Since mixins do *not* introduce types on their own, `EvalBase` cannot be used as a type. Using `EvalBase` as a type would be a

[7] Constructor difference is not a primitive construct of MLscript's underlying type system, MLstruct [26]. Type `A \ B` is encoded in that type system as `A & ~#B`, where `&` is the *type intersection* operator, `~` is the *type negation* operator, and `#B` represents the *nominal identity* of class `B`, i.e., its raw type constructor without any fields or type parameters attached.

[8] Perhaps counter-intuitively, we do not need to restrict the positive occurrences of `'b`, as they are always effectively unrestricted due to covariance. Consider a function of type `('b\Lit\Add) → 'b`. Substituting `Mul | Lit | Add` for `'b` results in `((Mul | Lit | Add)\Lit\Add) → (Mul | Lit | Add)`, which is equivalent to `Mul → (Mul | Lit | Add)`. This is a supertype of `Mul → Mul`, which we could have obtained from substituting `Mul` for `'b` in the first place, so this type would have been reachable even after a "properly restricted" substitution of `'b`. In other words, it does not make much sense to restrict the positive occurrence of `'b` and there is no practical need for it.

[9] We use uppercase names for *mixin-level* type variables and lowercase names for *function-level* ones.

problem because there would be no definite type to replace `'A` with in the signature of its `eval` method – so we would not know how to type expressions such as `x.eval` when `x` has type `EvalBase`. Note that `'A` can even be instantiated to *several incomparable types* within a single class, if `EvalBase` is inherited several times.

What is interesting here is that MLscript infers a **`this`** type refinement (also called *self type*), which specifies what the type of **`this`** should be for the mixin to be well-typed. Here, **`this`** represents the final object obtained from the future mixin composition. Crucially, notice that the type of `eval` is *no longer recursive* – indeed, it no longer contains a recursive '**`as`**' binder. This is because we have *opened* the recursion, and the type that is inferred for `eval` *precisely* specifies what this partial definition accomplishes: it examines the top level of an expression and when that expression is an `Add`, it calls `eval` open-recursively through **`this`** with the corresponding subexpressions, expecting integer results from that recursive call.

Opening recursion in this way allows us to adapt the interpretation of this partially-specified recursive function, as we shall see shortly.

**Closing back.** We can immediately tie the knot and obtain an equivalent implementation to the original recursive function `eval` by defining a class that only inherits from `EvalBase`:

```
class SimpleLang extends EvalBase
```

whose inferred type signature is:

```
class SimpleLang: {
  eval: (Lit | Add['a] as 'a) → Int
}
```

Something important happened here: by creating the class `SimpleLang` from the previous mixin, we effectively *tie the recursive knot* for the corresponding method. That is, to type check `SimpleLang`, MLscript constrains the "open" polymorphic type variable `'A` associated with `eval` in `EvalBase` and instantiates it to the correct type to make the overall mixin composition type check. More specifically, remember that `eval` as defined in `EvalBase` was given type `(Lit | Add['A]) → Int` *assuming* that **`this`** had type `{ eval: ('A) → Int }`. Here, we know that the type of **`this`** is `SimpleLang` and that `SimpleLang`'s `eval` implementation is the one inherited from `EvalBase`. So when constraining types to make the subtyping relation `SimpleLang <: { eval: ('A) → Int }` hold, this leads to constraining `(Lit | Add['A]) → Int <: ('A) → Int`, which in turn leads to the constraint `'A <: (Lit | Add['A])`. So MLscript instantiates the type variable `'A` to the principal solution, i.e the recursive type `(Lit | Add['a]) as 'a`, which satisfies this recursive constraint.

**Extending the operations.** Now consider extending our code for a new expression pretty-printing method:

```
mixin PrettyBase {
 fun print(e) = if e is
   Lit(n)        then toString(n)
   Add(lhs, rhs) then this.print(lhs) ++ "+" ++ this.print(rhs) }
```

Mixin `PrettyBase` defines a `print` method for `Lit` and `Add`. Its inferred type is analogous to that of `EvalBase`. This demonstrates that we can extend the operations performed on our simple language, which is one of the extensibility directions considered by the Expression Problem.

**Extending the data types.** Next, consider another direction of code extension – defining a *new expression constructor*. We here define a negation expression type `Neg`:

```
class Neg[T](expr: T)
```

Now, the obvious question is how to extend arbitrary existing operations to this new data type constructor in a way that is as general and modular as possible.

### super-charging OOP with Polymorphic Mixins

As noticed by Garrigue [16], it is often useful to define components that extend *yet unknown* base implementations, so that the same components can be applied to different base implementations, and so that in general we can merge independently-defined languages together. This is possible to do in MLscript by defining mixins that make use of **`this`** and **`super`**, as in the following example:

```
mixin EvalNeg {
  fun eval(e) =
    if e is Neg(d) then 0 - this.eval(d)
    else super.eval(e)
}
```

which can be written more concisely using the following syntax sugar:

```
mixin EvalNeg { fun eval(override Neg(d)) = 0 - this.eval(d) }
```

We can include this partial `Neg`-handling recursion step as part of any previously-defined base implementation, such as our previous `EvalBase`. We get the following inferred type for `EvalNeg`, which precisely describes this property:

```
mixin EvalNeg: ∀ 'A 'B 'R . {
  this:  { eval: 'A → Int }
  super: { eval: 'B → 'R }
  eval: (Neg['A] | 'B\Neg) → (Int | 'R)
}
```

We can see that the type signature of our mixin now includes a **`super`** refinement *in addition to* the **`this`** refinement. This is the key to enabling polymorphic extension: when composing such a mixin later on, MLscript will match up this **`super`** requirement with whatever implementation is provided by the previous mixin implementations in the chain of mixin composition. Recursive knots will only be tied when the mixin is composed as part of a class.

The `PrettyNeg` extension for pretty-printing is defined analogously.

**Tying the knot again.** Finally, we can compose everything together as part of a new class:

```
class Lang extends EvalBase, EvalNeg, PrettyBase, PrettyNeg
```

And here is the type signature inferred for this definition:

```
class Lang: {
  eval:  (Lit | Add['a] | Neg['a] as 'a) → Int
  print: (Lit | Add['a] | Neg['a] as 'a) → Str
}
```

Again, what happens here is important to consider. We are now tying the knot with respect to *both* **`this`** and **`super`** in all the mixins making up the mixin inheritance stack. More specifically, we start by making sure that the member types provided by the first mixin `EvalBase` satisfy the **`super`** requirement of the second mixin `EvalNeg`, then we compute new member types based on `EvalNeg`'s contributions, before checking that the resulting type

satisfies the `super` requirement of the next mixin in line, `PrettyBase`, etc. This results in the inferred recursive types above, which precisely characterize what shapes of data that `Lang`'s `eval` and `print` methods can handle.

**Polymorphic extensibility.** To demonstrate that our `EvalNeg` component is truly generic over the existing implementation it is to be merged upon, we can define yet another mixin that adds a new `Mul` language feature:

```
class Mul[T](lhs: T, rhs: T)
mixin EvalMul { fun eval(override Mul(l, r)) = this.eval(l) * this.eval(r) }
```

And then we compose all of these mixins together in two possible orders (the order determines which of `Neg` and `Mul` will be matched first):

```
class LangNegMul extends EvalBase, EvalNeg, EvalMul
class LangMulNeg extends EvalBase, EvalMul, EvalNeg
```

In both cases, the inferred signature is equivalent:

```
class LangNegMul: { eval: (Lit | Add['a] | Neg['a] | Mul['a] as 'a) → Int }
```

## Pattern-Matching All the Way

To conclude this motivating example, we exemplify a capability of our system that most solutions to the expression problem lack, with the notable exception of polymorphic variants (see Section 4.3): the ability of *pattern matching deeply inside subexpressions*, which enables the definition of optimization passes.

For instance, below we define an `EvalNegNeg` optimization which shortcuts the evaluation of double negations, directly evaluating the doubly-negated expression instead:

```
mixin EvalNegNeg { fun eval(override Neg(Neg(d))) = this.eval(d) }
```

of inferred type:

```
mixin EvalNegNeg: ∀ 'A 'B 'C 'D . {
  super: {eval: (Neg['A] | 'B) → 'C}
  this:  {eval: 'D → 'C}
  fun eval: (Neg[Neg['D] | 'A\Neg] | 'B\Neg) → 'C
}
```

This type deserves some explanation. The parameter type of `eval` is '`Neg[Neg['D] | 'A \ Neg] | 'B\Neg`', which describes the fact that:

- `eval` accepts either an instance of `Neg` or, failing that, a `'B` that is *not* a `Neg`;
- If the argument *is* a `Neg`, then its type argument must itself be either a `Neg` or an `'A` that is not a `Neg`;
- If that nested type argument is a `Neg`, then its type argument must be `'D`. Since this type argument is passed to `this.eval`, we get the `this` refinement `{eval: 'D → 'C}`.
- In case either the `eval` argument is not a `Neg` (so the argument is a `'B`) or the `eval` argument is a `Neg['A]` where `'A` is not a `Neg`, evaluation falls back to a `super` call, which is translated into the `super` refinement `{eval: (Neg['A] | 'B) → 'C}`.

This mixin can be merged onto any mixin stack to obtain the desired effect; for example:[10]

```
class Lang extends EvalBase, EvalNeg, EvalMul, EvalNegNeg
```

[10] In this case, it is important to mix in `EvalNegNeg` *after* `EvalNeg` in the inheritance stack, so that the optimization semantics override the base semantics, and not the other way around. This is a fundamental property of optimization passes: their composition order matters.

# 3 A Core Language for SuperOOP

In this section, we present an explicitly-typed core language that captures the core object-oriented concepts of SuperOOP, leaving type inference aside. We first informally present the key innovation of SuperOOP's object-oriented type system and then define $\lambda^{\mathsf{super}}$, a minimal declarative and explicitly-polymorphic calculus.

## 3.1 SuperOOP Core Concepts

The core concepts of SuperOOP can be summarized as follows.

**Interfaces, mixins, and classes.** Interfaces, mixins, and classes are three orthogonal building blocks that model OOP in our system. ***Interfaces*** define a set of method signatures. For an object conforming to an interface, it should support all the methods specified in that interface. Contrary to classes and mixins, which in our core language have no types, we associate each interface with its own type. ***Mixins*** provide *implementations* for methods. ***Classes***, finally, implement interfaces by a set of parameters, which represent the *state* of the object, and a linear composition of mixins.

**Interface inheritance.** As in most OOP languages, existing interfaces can be extended with additional methods through interface inheritance. A child interface may inherit from several parent interfaces (i.e., we support *multiple inheritance* of interfaces). Moreover, a child interface may override parent method signatures with *refined* signatures, as determined by the subtyping relation. As an example, consider the following interface composition:

```
interface I1 { a: S }; interface I2 { a: T }; interface I3 extends I1, I2
```

Method `a`'s signature in the composed interface `I3` is the *intersection* of the inherited signatures, i.e. `S & T`. Intersection types enable precise multiple interface inheritance, since they are used as greatest lower bounds of the inherited type signatures, which also makes the composed interface a subtype of all inherited interfaces.

**Mixin composition.** SuperOOP mixins are compositional and reusable building blocks to construct classes. They provide partial method implementations that, when composed together, are checked to satisfy the interface that the class is meant to conform to. A mixin composition is simply a list of mixins. Each mixin in a mixin composition *overrides* not only method implementations but also method *types* inherited from previous mixins. So the type of a method may change along the mixin composition, but the type system ensures that the typing assumptions made by each implementation (in the form of **`this`** and **`super`** refinements) are satisfied. This also explains why mixins are not considered types (unlike, e.g., Scala traits): the fact that a mixin is present in the inheritance clause of a class does *not* imply that the resulting object will offer methods with types comparable to the ones provided by the mixin.

**Precisely-Typed Open Recursion.** A crucial feature of OOP, *open recursion* is the ability for a method to invoke itself or another method via a late-bound **`this`** instance, which may lead to evaluating overriding implementations. In most OOP languages with inheritance, the type of **`this`** is the current class's type. In these languages, method invocations on **`this`** are safe because overriding implementations from subclasses can only refine the types of overridden methods. By contrast, in SuperOOP, methods are overridden regardless of types,

Names, types, and terms

| | |
|---|---|
| *Class name* | $C$ |
| *Mixin name* | $M, N$ |
| *Interface name* | $I, J$ |
| *Field name* | $m, p$ |
| *Type* | $S, T, U, V \ ::= \ X, Y \mid I[\overline{T}] \mid S \to T \mid \forall X.\ T \mid S \ \& \ T \mid \mathsf{Object}$ |
| *Term* | $e \ ::= \ x,\ y \mid \textbf{this} \mid \textbf{super} \mid \lambda x : T.\ e \mid \Lambda X.\ e$ |
| | $\mid e_1\ e_2 \mid e\ T \mid e.m \mid \textbf{new}\ C[\overline{T}](\overline{e})$ |

Interfaces, mixins, and classes

| | |
|---|---|
| *Structural type* | $\mathcal{R} \ ::= \ \{\ \overline{m : T}\ \}$ |
| *Implementation* | $\mathcal{I} \ ::= \ \{\ \overline{m : T = e}\ \}$ |
| *Top-level definition* | $\mathcal{D} \ ::=$ |
| *Interface* | $I[\overline{X}] \lhd \overline{J[\overline{T}]}\ \mathcal{R}$ |
| *Mixin* | $\mid M[\overline{X}]^{\mathcal{R}}_{T}\ \mathcal{I}$ |
| *Class* | $\mid C[\overline{X}](\overline{m : T}) \lhd I[\overline{T}], \overline{M[\overline{T}]}$ |
| *Program* | $\mathcal{P} \ ::= \ \overline{\mathcal{D}};\ e$ |

**Figure 1** Syntax of $\lambda^{\textbf{super}}$.

and the actual type of **this** is only decided when the mixin composition is finalized as part of a class definition. Therefore, a precise type specification for **this** is necessary for open recursive calls in mixin methods. Importantly, **this** type refinement can be polymorphic at the mixin level, being instantiated at mixin composition time (i.e., upon being used as part of a class definition). Such polymorphism allows for later extensions to the shapes of data types that a method may be made to work on, as described in Section 2.

## 3.2 Formal Syntax

We now introduce the $\lambda^{\textbf{super}}$ calculus, a formalization of SuperOOP. The design of this calculus is inspired by Featherweight Generic Java [19] and Pathless Scala [20]. Throughout our formalization, we use the notation $\overline{E_i}^{i \in n..m}$ to denote the repetition of syntax form $E_i$ with index $i$ from $n$ to $m$. We use $\overline{E}$ as a shorthand when $i$ is not necessary for disambiguation. Moreover, we use $[\overline{T/X}]$ to denote the conventional capture-avoiding substitution of a list of type parameters $\overline{X}$ (which can possibly be empty) to $\overline{T}$. In definitions of metafunctions, we use $\varnothing$ as a default vacuous result.

The syntax of $\lambda^{\textbf{super}}$ is presented in Figure 1. Meta-variables $S, T, U, V$ range over types, which include type variables, interfaces with a list of type arguments, arrow types, universally quantified types, intersection types, and the top type Object. For terms $e$, there are term variables $x$ and $y$. **this** and **super** are akin to term variables with special treatment. We have standard explicitly-typed lambda abstractions and term applications, as well as type abstraction and type application terms. Method invocation and access to object fields share a single syntax: we consider access to object fields as method invocation. Objects are created with a **new** keyword with term and type arguments supplied.

$$\boxed{S <: T}$$

$$\text{S-Refl}\ \frac{}{T <: T} \qquad \text{S-Top}\ \frac{}{T <: \mathsf{Object}}$$

$$\text{S-Interface}\ \frac{S \in \mathsf{parents}(I[\overline{T}])}{I[\overline{T}] <: S} \qquad \text{S-Inv}\ \frac{\overline{S <: T} \quad \overline{T <: S}}{I[\overline{S}] <: I[\overline{T}]}$$

$$\text{S-Andl}\ \frac{}{S_1 \ \&\ S_2 <: S_1} \qquad \text{S-Andr}\ \frac{}{S_1 \ \&\ S_2 <: S_2}$$

$$\text{S-And}\ \frac{S <: T_1 \quad S <: T_2}{S <: T_1 \ \&\ T_2} \qquad \text{S-Trans}\ \frac{S <: U \quad U <: T}{S <: T}$$

$$\text{S-Arrow}\ \frac{S_2 <: S_1 \quad T_1 <: T_2}{S_1 \to T_1 <: S_2 \to T_2} \qquad \text{S-Forall}\ \frac{S <: T}{\forall X.\ S <: \forall X.\ T}$$

**Figure 2** Declarative subtyping.

The top-level definitions of $\lambda^{\mathsf{super}}$ are interfaces, mixins, and classes. Every interface $I[\overline{X}]$ has a type parameter list $[\overline{X}]$, a structural refinement $\mathcal{R}$, and inherits multiple parent interfaces $\overline{J[\overline{T}]}$. A structural refinement $\mathcal{R}$ contains a list of method signatures $\overline{m : T}$ that specify methods' names and types. Mixins, parametrized by type parameters, provide method implementations $\mathcal{I}$. Crucially, each mixin has a structural refinement $\mathcal{R}$ attached to **super** and a type $T$ for **this** for precise typing of open recursion. Finally, a class has a class-level type parameter list, immutable object fields, an interface it implements, and a mixin composition $\overline{M[\overline{T}]}$ that provides method implementations. A program consists in a list of top-level definitions and a term that accesses them. For all top-level definitions, we require the standard well-formedness conditions that all names are uniquely defined and no class transitively inherits itself. In later rules, we assume terms' access to the underlying top-level definitions.

## 3.3 Static Semantics

We present the static semantics of $\lambda^{\mathsf{super}}$ which includes a declarative subtyping, term typing, and well-formedness check of top-level definitions.

**Declarative subtyping.** Figure 2 shows the declarative subtyping of $\lambda^{\mathsf{super}}$. Most rules are unsurprising. Rule S-Interface describes that an interface is a subtype of its parent interfaces. Auxiliary function $\mathsf{parents}(I[\overline{T}])$ (defined in the extended version) returns the list of parent interfaces. For simplicity, we consider that interfaces are *invariant* in their type parameters (rule S-Inv). A universally quantified type is a subtype of another universally quantified type only when they are quantifying the same type variable.

**Term typing.** Figure 3 lists the typing rule of terms. $\Gamma \vdash e : T$ is the term typing relation. A typing context $\Gamma$ maps term variables to types, **super** to a structural refinement, and **this** to a type. The typing rules for term variables (T-Var), lambda and type abstractions (T-Abs and T-TAbs), term and type applications (T-App and T-TApp), as well as the subsumption rule (T-Sub), are standard. Note that since **super** is not bound to a type (but to a structural refinement) in typing contexts, **super** itself will never be assigned a type, which matches the usual semantics of **super** that it should only receive method call messages but not be passed around. The typing of method invocations is separated into two cases. If the receiver is a term (other than **super**) that has a type, we look up the method signature

*Typing context* $\Gamma ::= \epsilon \mid \Gamma, x : T \mid \Gamma, \textbf{super} : \mathcal{R} \mid \Gamma, \textbf{this} : T$

$\boxed{\Gamma \vdash e : T}$

$$\textsc{T-Var}\ \frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \textsc{T-This}\ \frac{\Gamma(\textbf{this}) = T}{\Gamma \vdash \textbf{this} : T} \qquad \textsc{T-Abs}\ \frac{\Gamma, x : S \vdash e : T}{\Gamma \vdash \lambda x : S.\ e : S \to T} \qquad \textsc{T-TAbs}\ \frac{\Gamma \vdash e : T}{\Gamma \vdash \Lambda X.\ e : \forall X.\ T}$$

$$\textsc{T-App}\ \frac{\Gamma \vdash e_1 : S \to T \quad \Gamma \vdash e_2 : S}{\Gamma \vdash e_1\ e_2 : T} \qquad \textsc{T-TApp}\ \frac{\Gamma \vdash e : \forall X.\ S}{\Gamma \vdash e\ T : [T/X]S} \qquad \textsc{T-Access}\ \frac{\Gamma \vdash e : T \quad \mathsf{mtype}(m, T) = S}{\Gamma \vdash e.m : S}$$

$$\textsc{T-Super}\ \frac{\Gamma(\textbf{super}) = \mathcal{R} \quad \mathsf{mrefn}(m, \mathcal{R}) = S}{\Gamma \vdash \textbf{super}.m : S} \qquad \textsc{T-New}\ \frac{\mathsf{vparams}(C[\overline{T}]) = \overline{m_i : U_i}^{\,i \in 1..n} \quad \overline{\Gamma \vdash e_i : U_i}^{\,i \in 1..n} \quad \mathsf{ctype}(C[\overline{T}]) = V}{\Gamma \vdash \textbf{new}\ C[\overline{T}](\overline{e_i}^{\,i \in 1..n}) : V} \qquad \textsc{T-Sub}\ \frac{\Gamma \vdash e : S \quad S <: T}{\Gamma \vdash e : T}$$

Given that interface $I$ is defined as $I[\overline{X}] \lhd \overline{J[\overline{U}]}\ \mathcal{R}$:

$$\mathsf{mtype}(m, I[\overline{T}]) = \begin{cases} [\overline{T/X}]S & \text{if } (m : S) \in \mathcal{R} \\ S & \text{if } m \notin \mathcal{R} \text{ and } \mathsf{mtype}(m, \&\overline{[\overline{T/X}]J[\overline{U}]}) = S \end{cases}$$

$$\mathsf{mtype}(m, S\ \&\ T) = \begin{cases} U\ \&\ V & \text{if } \mathsf{mtype}(m, S) = U \text{ and } \mathsf{mtype}(m, T) = V \\ U & \text{if } \mathsf{mtype}(m, S) = U \text{ and } \mathsf{mtype}(m, T) = \varnothing \\ V & \text{if } \mathsf{mtype}(m, S) = \varnothing \text{ and } \mathsf{mtype}(m, T) = V \end{cases}$$

$$\mathsf{mtype}(m, T) = \varnothing \text{ otherwise}$$

**Figure 3** Term typing.

in the receiver's type. Function $\mathsf{mtype}(m, T)$ computes method $m$'s signature from type $T$. Otherwise, if the receiver is **super**, we directly read the method type from its associated structural refinement using function $\mathsf{mrefn}(m, \mathcal{R})$ (defined in the extended version). To type class instantiation (T-New), we check that all constructor arguments match the types of the class fields returned by function $\mathsf{vparams}(C[\overline{T}])$, and the object has interface type $\mathsf{ctype}(C[\overline{T}])$ of the class ($\mathsf{vparams}$ and $\mathsf{ctype}$ are defined in the extended version).

The design of $\mathsf{mtype}$ basically follows that of Pathless Scala [20]. When a method signature is present in an interface, we directly return it. Otherwise, we search parent interfaces by calling $\mathsf{mtype}$ with the *intersection* of all parent interfaces (denoted as $\&\overline{J[\overline{U}]}$). Note that nullary intersection is $\mathsf{Object}$. To compute a method signature from an intersection type, we recursively consider both sides of the intersection. When both types define the method, we take the intersection of corresponding results.

**Well-formedness of top-level definitions.** Figure 4 shows the well-formedness check of mixins, classes, and interfaces. We put name lookup results of those structures as premises in the rules. The first premises of rules in Figure 4 are the case.

$$\boxed{M\ \textbf{\textit{ok}}} \qquad \textsc{MixinCheck}\ \frac{M[\overline{X}]^{\mathcal{R}}_{T}\ \{\ \overline{m : S = e}\ \} \qquad \forall (m : S = e) \in M\,.\quad \textbf{this} : T,\ \textbf{super} : \mathcal{R} \vdash e : S}{M\ \textbf{\textit{ok}}}$$

$$\boxed{I\ \textbf{\textit{ok}}} \qquad \textsc{InterfaceCheck}\ \frac{I[\overline{X}] \lhd \overline{J[\overline{T}]}\ \{\ \overline{m : S}\ \} \qquad \overline{J\ \textbf{\textit{ok}}} \qquad \forall (m : S) \in I\,.\, \mathsf{mtype}(m, \&\overline{J[\overline{T}]}) = \varnothing \text{ or } \begin{cases} \mathsf{mtype}(m, \&\overline{J[\overline{T}]}) = U \\ S <: U \end{cases}}{I\ \textbf{\textit{ok}}}$$

$$\boxed{C\ \textbf{\textit{ok}}} \qquad \textsc{ClassCheck}\ \frac{C[\overline{X}](\overline{p : T}) \lhd I[\overline{U}], \overline{M_i[\overline{U'}]}^{i \in n..1} \qquad I\ \textbf{\textit{ok}} \qquad \overline{M_i\ \textbf{\textit{ok}}} \qquad \overline{M_i \Rightarrow C} \qquad \forall m \in \mathsf{mnames}(I[\overline{U}])\,. \begin{cases} \mathsf{mtype}(m, I[\overline{U}]) = S \\ \mathsf{search}(m, 0, C) = V \\ V <: S \end{cases}}{C\ \textbf{\textit{ok}}}$$

$$\boxed{M_i \Rightarrow C} \qquad \textsc{InheritCheck}\ \frac{C[\overline{X}](\overline{p : U'}) \lhd I[\overline{U}], \overline{M_i[\overline{V}]}^{i \in n..1} \qquad M_i[\overline{Y}]^{\mathcal{R}}_{T}\ \mathcal{I} \qquad I[\overline{U}] <: [\overline{V/Y}]T \qquad \forall (m : S) \in \mathcal{R}\,. \begin{cases} \mathsf{search}(m, (i+1), C) = S' \\ S' <: [\overline{V/Y}]S \end{cases}}{M_i \Rightarrow C}$$

**Figure 4** Well-formedness check of top-level definitions and mixin inheritance check.

**Well-formed mixins.** To check a mixin ($M$ ***ok***), we check that every method implementation can be typed at its signature with precise types of **this** and **super** in the context.[11]

**Well-formed interfaces.** An interface is well-formed ($I$ ***ok***) when its parent interfaces are all well-formed. A method signature should either be newly introduced (in this case, $\mathsf{mtype}(m, \&\overline{J[\overline{T}]}) = \varnothing$), or have a subtype of the intersection of all $m$'s signatures in parents (i.e., $\mathsf{mtype}(m, \&\overline{J[\overline{T}]}) = U$).

**Well-formed classes.** Class well-formedness check ($C$ ***ok***) considers the following aspects:

1. The implemented interface and each mixin in the mixin composition are well-formed.
2. Open-recursive calls via **this** in the mixin composition are safe: the class type is a subtype of each mixin's **this** type annotation.
3. The mixin composition is correct: each mixin's structural refinement on **super** is satisfied.
4. The interface is satisfied: the class has all methods (and fields, as we uniformly treat fields and methods) required, and their signatures conform to the interface.

[11] Note that we bind **this** to a type while **super** to a structural refinement in each mixin. For **super**, the parent mixin in the composition hierarchy does *not* define an object type. It is therefore enough to give **super** a structural method refinement to tell what types the overridden methods should have. On the other hand, **this** is late-bound to the receiver object that has a type, can be passed around, and receive method invocation messages. Hence **this** is annotated with a type, and the annotated type should be a supertype of the later defined class's type.

Given that class $C$ is defined as $C[\overline{X}](\overline{m_j : T_j}) \lhd I[\overline{S'}], \overline{M_i[\overline{S}]}^{i\in n..1}$,
and mixin $M_i$ is defined as $M_i[\overline{Y}]^{\mathcal{R}}_{V}\ \mathcal{I}$:

$$\mathsf{search}(m_j, 0, C) = \begin{cases} T_j & \text{if } m_j : T_j \in \overline{m_j : T_j} \\ U & \text{if } m_j \notin \overline{m_j : T_j} \text{ and } \mathsf{search}(m_j, 1, C) = U \end{cases}$$

$$\mathsf{search}(m, i, C) = \begin{cases} [\overline{S/Y}]U & \text{if } 0 < i \leq n \text{ and } (m : U = e) \in \mathcal{I} \\ U & \text{if } 0 < i \leq n \text{ and } m \notin \mathcal{I} \text{ and } \mathsf{search}(m, (i+1), C) = U \end{cases}$$

$$\mathsf{search}(m, i, C) = \varnothing \text{ otherwise}$$

**Figure 5** Method implementation type search function.

For **1.**, $I$ ***ok*** checks the interface, and $\overline{M\ \textbf{\textit{ok}}}$ checks each mixin. Relation $M_i \Rightarrow C$ implements mixin inheritance check which deals with **2.** and **3.**. It checks if the inheritance of the $i$-th mixin in class $C$'s mixin composition is correct. Note that the index $i$ here ranges in $n..1$ (as $\overline{M_i[\overline{S}]}^{i\in n..1}$), which means syntactically, the *rightmost* mixin in the mixin composition is the *first* one. Rule InheritCheck guarantees that, first, **this** type of the $i$-th mixin should be a *supertype* of the interface that the class conforms to, which satisfies **2.**. Second, for each method $m$'s signature in the structural refinement of **super**, the parent mixin composition provides a compatible implementation. Specifically, the type of $m$'s implementation provided by mixins ranging in $n..(i+1)$ (computed by $\mathsf{search}(m, (i+1), C)$, defined in Figure 5 and explained later) should be a *subtype* of the $i$-th mixin's **super** refinement on $m$, which satisfies **3.**. To satisfy **4.**, for each method name $m$ defined in the interface (computed by $\mathsf{mnames}$, defined in the extended version), its implementation type provided by the class fields or mixin composition (computed by $\mathsf{search}(m, 0, C)$) should be compatible with the signature specified by the interface (computed by $\mathsf{mtype}$).

**Method implementation type search.** Figure 5 defines function $\mathsf{search}(m, i, C)$ to search implementation type of $m$ provided by fields or mixins ranging in $n..i$. When $i = 0$, it searches class fields for the method name $m$. If $m$ is not implemented by fields, the search continues with the first mixin ($i = 1$). For the $i$-th mixin, the search directly returns the method signature if $m$ is implemented in this mixin. Otherwise, it continues with the parent mixin (indexed $(i+1)$). The search returns $\varnothing$ if $i$ exceeds the length of class $C$'s mixin composition ($i > n$), which means $m$ is not implemented in the class, and the search fails.

## 3.4 Dynamic Semantics

Figure 6 lists the syntax of values, results, and runtime contexts, and lists the evaluation rules that produce values (the rules that produce runtime errors are omitted and can be found in the extended version). The big-step evaluation judgment $\Xi \vdash e \Downarrow r$ denotes that term $e$ evaluates to result $r$ under runtime context $\Xi$. The result of evaluation may be a normal value or an error. Values are either *closures* or *objects*. A runtime context $\Xi$ binds values to term variables and a *configured object* to **this**. A configured object $\{i \star C[\overline{T}](\overline{v})\}$ is a pair of an object and a natural number $i$ called the *search index*. This index directs the search for method implementation in the object fields and mixin composition at runtime. The evaluation rules for variables and term applications are standard. For type applications, while we use type substitution in the semantics, this will be no-op at runtime, as all generic types are erasable – only class tags are used at runtime, which are concrete types that need

$$
\begin{array}{rrcl}
\textit{Value} & v, w & ::= & \langle \lambda x : T.\ e, \Xi \rangle \mid \langle \Lambda X.\ e, \Xi \rangle \mid C[\overline{T}](\overline{v}) \\
\textit{Runtime context} & \Xi & ::= & \epsilon \mid \Xi,\ x \mapsto v \mid \Xi,\ \textbf{this} \mapsto \{i \star C[\overline{T}](\overline{v})\} \\
\textit{Result} & r & ::= & \textbf{val}\ v \mid \textbf{err}
\end{array}
$$

$\boxed{\Xi \vdash e \Downarrow r}$

$$\textsc{E-Var}\ \frac{\Xi(x) = v}{\Xi \vdash x \Downarrow \textbf{val}\ v} \qquad \textsc{E-This}\ \frac{\Xi(\textbf{this}) = \{i \star C[\overline{T}](\overline{v})\}}{\Xi \vdash \textbf{this} \Downarrow \textbf{val}\ C[\overline{T}](\overline{v})}$$

$$\textsc{E-App}\ \frac{\Xi \vdash e_1 \Downarrow \textbf{val}\ \langle \lambda x : T.\ e, \Xi' \rangle \quad \Xi \vdash e_2 \Downarrow \textbf{val}\ v \quad \Xi', x \mapsto v \vdash e \Downarrow \textbf{val}\ v'}{\Xi \vdash e_1\ e_2 \Downarrow \textbf{val}\ v'} \qquad \textsc{E-TApp}\ \frac{\Xi \vdash e \Downarrow \textbf{val}\ \langle \Lambda X.\ e', \Xi' \rangle \quad [T/X]\Xi' \vdash [T/X]e' \Downarrow \textbf{val}\ v}{\Xi \vdash e\ T \Downarrow \textbf{val}\ v}$$

$$\textsc{E-Abs}\ \frac{}{\Xi \vdash \lambda x : T.\ e \Downarrow \textbf{val}\ \langle \lambda x : T.\ e, \Xi \rangle} \qquad \textsc{E-TAbs}\ \frac{}{\Xi \vdash \Lambda X.\ e \Downarrow \textbf{val}\ \langle \Lambda X.\ e, \Xi \rangle}$$

$$\textsc{E-New}\ \frac{\mathsf{vparams}(C[\overline{T}]) = \overline{m_i} \quad \overline{\Xi \vdash e_i \Downarrow \textbf{val}\ v_i}}{\Xi \vdash \textbf{new}\ C[\overline{T}](\overline{e_i}) \Downarrow \textbf{val}\ C[\overline{T}](\overline{v_i})} \qquad \textsc{E-Access}\ \frac{\Xi \vdash e \Downarrow \textbf{val}\ C[\overline{S}](\overline{v}) \quad (\textbf{this} \mapsto \{0 \star C[\overline{S}](\overline{v})\}) \vdash \textbf{super}.m \Downarrow \textbf{val}\ v'}{\Xi \vdash e.m \Downarrow \textbf{val}\ v'}$$

$$\textsc{E-ArgMiss}\ \frac{\Xi(\textbf{this}) = \{0 \star C[\overline{S}](\overline{v})\} \quad m \notin \mathsf{vparams}(C[\overline{S}]) \quad (\textbf{this} \mapsto \{1 \star C[\overline{S}](\overline{v})\}) \vdash \textbf{super}.m \Downarrow \textbf{val}\ v'}{\Xi \vdash \textbf{super}.m \Downarrow \textbf{val}\ v'} \qquad \textsc{E-ArgHit}\ \frac{\Xi(\textbf{this}) = \{0 \star C[\overline{S}](\overline{v_i})\} \quad \mathsf{vparams}(C[\overline{S}]) = \overline{m_i : U_i}}{\Xi \vdash \textbf{super}.m_i \Downarrow \textbf{val}\ v_i}$$

$$\textsc{E-SuperMiss}\ \frac{\Xi(\textbf{this}) = \{i \star C[\overline{S}](\overline{v})\} \quad i > 0 \quad m \notin \mathsf{methods}(i, C[\overline{S}](\overline{v})) \quad (\textbf{this} \mapsto \{(i+1) \star C[\overline{S}](\overline{v})\}) \vdash \textbf{super}.m \Downarrow \textbf{val}\ v'}{\Xi \vdash \textbf{super}.m \Downarrow \textbf{val}\ v'}$$

$$\textsc{E-SuperHit}\ \frac{\Xi(\textbf{this}) = \{i \star C[\overline{S}](\overline{v})\} \quad i > 0 \quad (m : U = e) \in \mathsf{methods}(i, C[\overline{S}](\overline{v})) \quad (\textbf{this} \mapsto \{(i+1) \star C[\overline{S}](\overline{v})\}) \vdash e \Downarrow \textbf{val}\ v'}{\Xi \vdash \textbf{super}.m \Downarrow \textbf{val}\ v'}$$

**Figure 6** Big-step operational semantics producing values.

no substitution. Note that evaluation of **this** is to simply read the configured object from the context and return a plain object (i.e., with no search index). Class instantiations produce objects. Lambda and type abstractions are evaluated to closures. Note that $\lambda^{\textbf{super}}$ would not need a value restriction [32] even if we added imperative effects to it, because it does not evaluate under polymorphic abstractions. This is different from the real MLscript language, which does need a value restriction as it uses ML-style polymorphism.

**Method invocation and access to fields.** Proper modeling of method invocation and access to fields are of our particular interest. The following procedure explains the overall idea:

1. When the receiver is a term (modulo **super**), we first evaluate the term to an object and search through the object's fields for the method implementation (E-Access).
2. If the invoking method is not provided by any object field, we traverse the mixin composition of the class (E-ArgMiss).

$$v : T$$

$$\text{VT-Abs1}\ \frac{\Gamma \vDash \Xi \quad \Xi(\textbf{this}) = \{i \star C[\overline{U}](\overline{v})\} \quad \mathcal{R} \vDash \{i \star C[\overline{U}](\overline{v})\} \quad \Gamma, x : S, \textbf{super} : \mathcal{R} \vdash e : T}{\langle \lambda x : S.\ e, \Xi \rangle : S \to T}$$

$$\text{VT-TAbs1}\ \frac{\Gamma \vDash \Xi \quad \Xi(\textbf{this}) = \{i \star C[\overline{S}](\overline{v})\} \quad \mathcal{R} \vDash \{i \star C[\overline{S}](\overline{v})\} \quad \Gamma, \textbf{super} : \mathcal{R} \vdash e : T}{\langle \Lambda X.\ e, \Xi \rangle : \forall X.\ T}$$

$$\text{VT-Sub}\ \frac{v : S \quad S <: T}{v : T}$$

$$\mathcal{R} \vDash \{i \star C[\overline{S}](\overline{v})\}$$

$$\frac{C[\overline{X}](\overline{n : T'}) \triangleleft I[...], \overline{M[...]} \qquad \forall\, (m : T) \in \mathcal{R}\,.\, \begin{cases} \mathsf{search}(m, i, C) = U \\ [\overline{S/X}]U <: T \end{cases}}{\mathcal{R} \vDash \{i \star C[\overline{S}](\overline{v})\}}$$

**Figure 7** Value typing of closures.

3. If the invoking method is provided as an object field, we return the value bound to the field (E-ArgHit).
4. If the invoking method is not implemented by the $i$-th mixin, we search the next mixin in the composition hierarchy (E-SuperMiss). Helper function $\mathsf{methods}(i, C[\overline{S}](\overline{v}))$ (defined in the extended version) returns all method implementations of the $i$-th mixin.
5. If the invoking method is implemented by the $i$-th mixin, we evaluate the method body with **this** bound to the configured object where the search index points to the parent mixin (E-SuperHit).

## 3.5 Metatheory

We now develop the metatheory of $\lambda^{\textbf{super}}$. We follow Ernst et al.'s approach to prove type soundness of our big-step style semantics.

**Value typing.** Our metatheory focuses on *strong* soundness, that is, we need to type values to ensure that the evaluation result keeps the type. Value typing rules of closures are listed in Figure 7. Rule VT-Abs1 types lambda abstraction body under a typing context $\Gamma$ with the term variable bound to the input type and **super** refined by a structural refinement $\mathcal{R}$. Here we perform two consistency checks. First, the typing context should be consistent with the runtime context ($\Gamma \vDash \Xi$, rules are listed in the extended version), i.e., each term variable is bound to a value that matches the variable's type in the typing context. Second, to guarantee that calls to super implementations are always safe, the structural refinement $\mathcal{R}$ giving precise types to calls on **super** in the closure body should be *consistent* with the configured object in the closure's context. Relation $\mathcal{R} \vDash \{i \star C[\overline{S}](\overline{v})\}$ implements the second consistency check, which examines each method signature's compatibility with the method implementation type provided by the configured object. The remaining rules (in the extended version) that type objects and closures with no binding to **this** in the context are non-surprising.

**Soundness.** We finally show the soundness results of our formal calculus. The complete proofs can be found in the extended version. For a program $\mathcal{P}$, we denote its top-level definitions as $\overline{\mathcal{D}_\mathcal{P}}$ and the associated term as $e_\mathcal{P}$. The preservation lemma is stated below:

▶ **Lemma 1** (Preservation). *If* $\overline{\mathcal{D}_\mathcal{P}}$ ***ok*** *and* $\epsilon \vdash e_\mathcal{P} : T$ *and* $\epsilon \vdash e_\mathcal{P} \Downarrow r$ *then* $r = \textbf{val}\ v$ *and* $v : T$.

We define the *finite evaluation* relation [9] here to augment our big-step semantics with fuel.

▶ **Definition 2** (Finite evaluation). *Define an evaluation relation* $\Xi \vdash e \Downarrow_k r^+$ *(where* $r^+ ::= r \mid \textbf{kill}$*, and* $k$ *is the step-counting index, i.e. fuel) with evaluation rules copied from* $\Xi \vdash e \Downarrow r$*. For each rule,* $\Downarrow$ *in the conclusion is replaced by* $\Downarrow_k$*, and* $\Downarrow$ *in premises is replaced by* $\Downarrow_{k-1}$*. Also, propagate timeout result of subderivations (the corresponding rules are listed in the extended version). Finally, add the following axiom:*

*E-Timeout*
$\Xi \vdash e \Downarrow_0 \textbf{kill}$

The soundness theorem of our calculus follows from the preservation lemma that rules out errors when evaluation terminates and the coverage lemma that ensures our evaluation rules with finite fuel always produce a result.

▶ **Lemma 3** (Coverage). *For all* $n$*,* $\Xi$*, and* $e$*, there exists an* $r^+$ *such that* $\Xi \vdash e \Downarrow_n r^+$*.*

▶ **Definition 4** (Expression divergence). $e$ *diverges* $\triangleq$ *For all* $n$*,* $\epsilon \vdash e \Downarrow_n \textbf{kill}$*.*

▶ **Theorem 5** (Soundness). *If* $\overline{\mathcal{D}_\mathcal{P}\ \textbf{ok}}$ *and* $\epsilon \vdash e_\mathcal{P} : T$ *then (1)* $\epsilon \vdash e_\mathcal{P} \Downarrow \textbf{val}\ v$ *and* $v : T$*, or (2)* $e_\mathcal{P}$ *diverges.*

# 4 Discussion and Related Work

We now discuss the expressiveness, limitations, and implementation of SuperOOP as presented in this paper, and we compare our approach to related work.

## 4.1 Expressiveness and Limitations

Thanks to the clear separation of concerns between the orthogonal concepts of interfaces, mixins, and classes, and thanks to the flexibility of mixins, SuperOOP not only captures standard OOP features but can also be used to explain existing advanced OOP models.

**Desugaring traditional classes.** A classic OOP class is desugared into three SuperOOP core language components: (a) a core-language class for its fields; (b) a core-language mixin for its implementations; and (c) a core-language interface for its method signatures. Although our core language does not directly support class inheritance, this feature can easily be desugared into SuperOOP. For example, recall `ColoredPoint` from Section 1, which inherited from class `Point`. This class hierarchy can be desugared to SuperOOP as:

```
interface IPoint { x: Int; y: Int }
class Point(x: Int, y: Int) implements IPoint

interface IColoredPoint extends IPoint, Colored
class ColoredPoint(x: Int, y: Int, color: Color) implements IColoredPoint
```

**Multiple inheritance and linearization.** Languages that support multiple inheritance usually have a *linearization* mechanism that determines the order of inherited parent classes, traits, or mixins. The underlying assumption is that each parent can only be inherited at most once, so if a parent transitively occurs more than once in an inheritance clause, the linearization mechanism removes all but its first occurrence. Consequently, linearization affects the semantics of method resolution and `super`-calls. For example, Scala uses linearization for its multiple trait inheritance system [22]. The linearization of a Scala class definition of the form `class C extends` $\mathtt{B}_0$`,` $\mathtt{B}_1$`, ...,` $\mathtt{B}_n$ starts with $\mathtt{B}_0$'s linearization and appends to it the

linearization of $\texttt{B}_1$ save for those traits that are already in the constructed linearization of $\texttt{B}_0$, etc. Several languages such as Python adopt the influential C3 linearization algorithm [1]. Although SuperOOP does not natively support multiple class inheritance, we can still apply any linearization algorithms used by existing languages and desugar the result using core SuperOOP classes, interfaces, and mixins. On the other hand, in SuperOOP, one can inherit a given mixin an arbitrary number of times at different positions in the mixin inheritance stack. The resolution of method invocations simply follows the order of inherited mixins, which do not necessarily need to be linearized. So SuperOOP's approach is more general.

We show an example encoding of Scala multiple trait inheritance in SuperOOP in the extended version of this paper.

**Mixin parameters.** Mixin parameters are a powerful extension to the core SuperOOP language presented in this paper. They for instance allow one to define flexible and efficient streaming processing abstractions that are composed through mixins, as in the following:

```
module MyPipeline extends
  Map(x => x + 1),
  Filter(x => x % 2 == 0),
  Map(x => x * 2)
```

We use *two* instances of `Map` in the mixin composition above, showing that using `this` refinements to encode mixin parameters would not be sufficient, as each of these two `Map` instances needs to be given a *different* argument. Mixin parameters are implemented in MLscript/SuperOOP, but we omitted this extension from $\lambda^{\textsf{super}}$ for simplicity.

**Member access control.** We have not yet modeled in the core language nor implemented any notions of encapsulation and visibility, such as the `private` and `protected` modifiers. We expect that modeling these features should be straightforward, as their design is mostly orthogonal to the features of SuperOOP.

## 4.2 Implementation of SuperOOP in MLscript

We now briefly describe our implementation and possible alternative implementation strategies.

**Compilation to JavaScript.** MLscript currently compiles to JavaScript, which supports classes as first-class entities. This means it is possible to define mixins directly, by using functions. For instance, the `EvalNeg` and `EvalMul` mixins and the `LangNegMul` class mentioned in Section 2 are essentially compiled into the following JavaScript code:

```
function mkEvalNeg(base) {
  return class EvalNeg extends base {
    eval(e) {
      if (e instanceof Neg) return 0 - this.eval(e.expr)
      else return super.eval(e) } }
}
function mkEvalMul(base) {
  return class EvalMul extends base {
    eval(e) {
      if (e instanceof Mul) return this.eval(e.lhs) * this.eval(e.rhs)
      else return super.eval(e) } }
}
class LangNegMul extends mkEvalMul(mkEvalNeg(EvalBase))
```

One side effect of this straightforward implementation is that mixins in MLscript can be inherited an arbitrary number of times and that no inheritance linearization is needed. MLscript *classes*, on the other hand, follow the usual single-inheritance hierarchy discipline, which is useful for type checking pattern matching and inferring simple types for it.

**Compilation to other targets.** We are also considering adding alternative compilation backends to MLscript, such as backend compilers targeting WebAssembly and the Java Virtual Machine. In that context, we can still follow the general JavaScript-based semantics described above, but we will make sure to evaluate the mixin functions at compilation time, to guarantee optimal performance and simple compilation. Super calls would then be resolved statically, allowing for efficient target code. Therefore, our approach to mixin composition should offer better performance than alternative solutions to the expression problem which rely on closure compositions and thus require virtual dispatch, like the approach of Garrigue [16]. However, we reserve a rigorous performance evaluation for future work.

**Separate compilation.** An aspect of the Expression Problem as originally stated is that it should be possible to compile each extension separately before putting them all together. We can essentially achieve this even in the static compiler scenario by separately compiling method *implementations* and composing classes whose methods simply forward to these pre-compiled implementations. This is more or less the approach used by Scala for traits, which was shown to be practical in real-world scenarios.

**Case studies.** In the extended version of this paper, we provide case studies of MLscript/SuperOOP that include a modular evaluator of extended lambda calculus, as described by Garrigue [16], and a simple "regions" DSL developed by Sun et al. [31]. These case studies showcase the flexibility of SuperOOP polymorphic mixins, the ability to handle mutually-recursive functions across different mixins, interpret complex data types, and optimize domain-specific languages via built-in nested pattern matching. Additionally, thanks to MLscript's powerful principal type inference [26], those case studies type check without the help of a single type annotation.

## 4.3 Solutions to the Expression Problem

There is a sea of work in extensible programming that address the Expression Problem, based on techniques such as polymorphic variants [15] in OCaml, recursive modules [21] in ML, and new programming paradigms [4, 23] like Compositional Programming [34]. We survey a few of them by showing their solutions to the Expression Problem and discuss various design tradeoffs with respect to the approach of SuperOOP.

**Polymorphic Variants.** The *polymorphic variant* (PV) solution [16] probably comes closest to our approach. Open recursion there is implemented by way of an explicit parameter for recursive calls, and by manually tying the recursive knots. For example, one defines an open-recursive base implementation of evaluation on two expression data types as follows:

```
let eval_base eval_rec = function
  | `Lit(n) → n
  | `Add(e1, e2) → eval_rec e1 + eval_rec e2
(* val eval_base :
  ('a → int) → [< `Add of 'a * 'a | `Lit of int ] → int *)
```

PVs differ from traditional variants or *algebraic data types* (ADTs) in that PVs allow the use of arbitrary constructors without a corresponding data type definition; they can be thought of as ADTs that are "not fully specified" and thus allow further extension. In the example above, two constructors `‘Lit` and `‘Add` are introduced. Function `eval_base` takes a first parameter `eval_rec` for open-recursive calls and the expression to evaluate as a second parameter. Parameter `eval_rec` accepts expressions with type `'a`, and the expression is required to have type `[< ‘Add of 'a * 'a | ‘Lit of int ]`, which allows either an `‘Add` expression containing nested subexpressions of type `'a`, or a `‘Lit` instance with an integer payload. Extending this base evaluator with new operations is done by composing it inside new functions. To extend the supported expression forms, one defines another evaluation implementation that works, e.g., on negations:

```
let eval_ext eval_rec = function
  ‘Neg(e) → 0 - eval_rec e
(* val eval_ext : ('a → int) → [< ‘Neg of 'a ] → int *)
```

Finally, one needs to tie both implementations together:

```
type 'a expr_base = [‘Lit of int | ‘Add of 'a * 'a]
type 'a expr_ext = [‘Neg of 'a]
let rec eval = function
  | #expr_base as x → eval_base eval x
  | #expr_ext as x → eval_ext eval x
(* val eval :
  ([< ‘Add of 'a * 'a | ‘Lit of int | ‘Neg of 'a ] as 'a) → int *)
```

Function `eval` dispatches the evaluation of the base and extended data types to the two evaluation sub-implementations, and it ties the recursive knots by passing itself as the entry point of the recursion. Note that `eval` has an inferred recursive type that accepts an expression recursively constructed by the three variants. Compared with our solution, from a programming style perspective, one programs with polymorphic variants in a functional way, while SuperOOP adopts a more object-oriented style. More importantly, polymorphic variants suffer from several practical drawbacks, including loss of polymorphism and approximated typing of pattern matching [5]. Those drawbacks can be fixed by embracing "proper" implicit subtyping as in MLscript [26]. In particular, we argue that union types are simpler than row polymorphism, which imperfectly emulates subtyping through unification [26].

**OCaml's Object System.** In OCaml class definitions, one can annotate "self" with a type signature and define "super" explicitly in a way that superficially looks similar to SuperOOP. One may be tempted to try and encode precise typing of open recursion in OCaml, to enable extensible programming with classes. However, this does not work due to OCaml's use of unification and its lack of subtyping: the self and super types are *unified* with the object type being defined, and thus all three must exactly coincide. By contrast, SuperOOP mixins allows *different* self and super types and allows overriding methods with *different* types, which is crucial for our technique. We discuss this in more detail in the extended version.

**Featherweight Generic Go.** Go is a popular programming language developed by Google. Featherweight Go (FG) and its generic version Featherweight Generic Go (FGG) proposed by Griesemer et al. [18] are formal developments of Go with the goal of helping "get polymorphism right". FGG provides a solution to the Expression Problem based on generics and covariant matching of method receiver type refinements, as in:

```
func (e Plus(type a Evaler)) Eval() int {
  return e.left.Eval() + e.right.Eval()
}
```

Method `Eval` is generic in type variable '`a`' which is upper-bounded by interface `Evaler`. Once dissociated from the quantification of `a`, the receiver type of the method is `Plus(a)`, the type of a `Plus` instance with subexpressions of type '`a`'. To extend the supported operations in the encoded language, one may define a similar pretty-printing method. Finally, one combines the interfaces for different interpretations together in a final expression type:

```
type Expr interface {
  Evaler
  Stringer
}
```

Type `Expr` composes two operations together, so it implements both of `Evaler` and `Stringer` (an interface for stringification). One can build and use such expressions as follows:

```
var e Expr = Plus(Expr){Lit{1}, Lit{2}}
var result Int = e.Eval()
var pretty string = e.String()
```

While this allows FGG to solve the Expression Problem, the features that enable this solution are not part of the Go team's current design for generics [18]. Moreover, the inspection of data structures only happens at the *outermost* level. If one wants to deeply transform an expression instance, that is, to inspect its inner structure and, for example, perform optimizations on it, one would have to make an additional method to delegate the inspection semantics itself. This approach, called *delegated method patterns* in Sun et al.'s work [31], is non-modular in FGG as it requires adding a new method for each inner structure inspection and to implement this method for each constructor of the data type, even those constructors that should otherwise fall into a default case of the encoded pattern matching.

**Object Algebras.** *Object Algebras* are a well-known object-oriented approach to solve the Expression Problem [23]. The key to this solution is an abstract factory called *object algebra interface*, which contains data type constructor signatures, leaving their interpretation unspecified. An object algebra interface for expressions could be, in Scala syntax:

```
trait ExpAlg[Exp] {
  def Lit: Int => Exp
  def Add: (Exp, Exp) => Exp
}
```

Trait `ExpAlg` specifies two data type constructors, and it is parameterized by type parameter `Exp` that indicates the interpretation of expression data types. We can implement evaluation on expressions by implementing the object algebra interface:

```
trait IEval { def eval: Int }
trait Eval extends ExpAlg[IEval] {
  def Lit = n => new IEval { def eval = n }
  def Add = (e1,e2) => new IEval { def eval = e1.eval + e2.eval }
}
```

Trait `Eval` is an object algebra which implements `ExpAlg` with the type parameter instantiated to `IEval`. Trait `IEval` indicates that expressions can be evaluated to integers. To extend the language with new operations, we may simply define a new interpretation type and a corresponding object algebra interface implementation. On the other hand, for new data type extensions, we inherit the object algebra interface and the old implementation:

```
trait NegAlg[Exp] extends ExpAlg[Exp] {
  def Neg: Exp => Exp
}
trait EvalNeg extends NegAlg[IEval] with Eval {
  def Neg = (e) => new IEval { def eval = 0 - e.eval }
}
```

We can now define an expression instance and instantiate the language:

```
trait exp[Exp](f: NegAlg[Exp]) {
  f.Add(f.Lit(1), f.Neg(f.Lit(-1)))
}
object eval extends EvalNeg
println(exp(eval).eval)
```

In trait `exp`, the data type constructors are accessed through the input object algebra `f`. With different implementations of the object algebra interface passed in, the expression will be interpreted in different ways. However, as noticed by Zhang et al. [34], one needs to create an expression instance for each data type interpretation, and there is no built-in approach to composing interpretations in different object algebras. Moreover, as data type constructors are specified through type *signatures* in object algebra interfaces, there is no way to have an inspectable representation of language instances without a complete definition of abstract syntax, blocking useful extensions such as modular transformations and optimizations.

**Compositional Programming.** *Compositional programming* [34] (implemented in the *CP* language) is a novel programming paradigm that features modularity. It supports a *merge operator* as the introduction term for intersection types. At the type level, the intersection type operator composes interfaces. At the term level, the merge operator composes *first-class traits* that contain data and operations. Similarly to Object Algebras, in Compositional Programming, a *compositional interface* specifies data type signatures, leaving their interpretation unspecified, and concrete interpretations are defined in first-class traits:

```
type ExpSig<Exp> = {
  Lit : Int → Exp;
  Add : Exp → Exp → Exp;
};
type Eval = { eval : Int };
evalNum = trait implements ExpSig<Eval> => {
  (Lit n).eval = n;
  (Add e1 e2).eval = e1.eval + e2.eval;
};
```

Trait `evalNum` implements the compositional interface `ExpSig<Eval>` which specifies that `Lit` and `Add` support an evaluation method. Similarly, one can implement a pretty-printing operation by adding another concrete interpretation. To extend the expression language with new data types, one extends the compositional interface and implements new operations in derived traits. Finally, everything is tied together with the merge operator as shown below:

```
type NegSig<Exp> extends ExpSig<Exp> = {
  Neg : Exp → Exp → Exp;
};
evalNeg = trait implements NegSig<Eval> inherits evalNum => {
  (Neg e).eval = 0 - e.eval;
};
```

```
exp Exp = trait [self : NegSig<Exp>] => {
  test = new Neg (new Add (new Lit 1) (new Lit 2));
};
// Assume pretty-printing of expression is analogously defined
e = new evalNeg ,, printNeg ,, exp @(Eval & Print);
```

Trait `exp` contains an example expression. The `self` type annotation in square brackets enables the trait body to access the three data type constructors. With the merge operator, trait instance `e` is composed of traits that contain different expression interpretations and the test trait. Note that trait `Exp` is passed with an intersection type argument `Eval & Print`, meaning the expression language supports both evaluation and pretty-printing.

In recent follow-up work on Compositional Programming by Sun et al. [31], different aspects of domain-specific language embedding are investigated, including the two-direction extensibility of language constructs and their interpretations, transformations and optimizations on language instances, etc. Since Compositional Programming does not natively support nested pattern matching (unlike our approach), deep inspection of data is only possible via the delegated method pattern (discussed above in the paragraph on Go), which is "not as convenient", as the authors put it. We also argue that this does *not* work well for defining *optimizations* in a modular way. Indeed, optimizations are fundamentally order-sensitive, and encoding them in terms of CP's unordered patterns requires non-local transformations of the involved pattern matching structures. For instance, one cannot *independently* define optimizations for evaluating `Neg(Neg(e))` as `e` and `Neg(Lit(n))` as `Lit(0 - n)`, whereas doing so in MLscript/SuperOOP is straightforward.

**Approaches lacking type safety.** It is much easier to solve the Expression Problem if one no longer cares about catching composition errors at compilation time. Zenger and Odersky [33] propose to use exception-throwing default cases in base implementations and to override these cases in further extensions, which relies on the programmer *remembering* to override all default cases and to pass only supported expression forms to the various methods in the program. Similar to SuperOOP, in a method that defines the interpretation of extended data types and overrides the base interpretation, they delegate the interpretation of base data types to the overridden method using **`super`**. While just as flexible as SuperOOP, this approach is fundamentally unsafe and error-prone. Going further, at the other end of the spectrum, approaches such as monkey-patching and Julia-style multiple dispatch allow completely dynamic updates of base implementations, which trivially supports extension but is anti-modular, as reasoning about the well-foundedness of method calls on given argument types requires global knowledge of all extension points in the program and libraries.

## 4.4 Modeling Inheritance and Reuse

In this subsection, we discuss previous work related to modeling inheritance and code reuse.

In their seminal *Inheritance Is Not Subtyping* paper, Cook et al. [7] introduced the crucial idea that inheritance could be unrestrained if it was decoupled from the subtyping relationship. However, they do not provide a specific source language in which to realize their ideas and only describe an imagined typed encoding of it, without an obvious way of connecting that encoding back to a hypothetical source language.

Bracha and Cook [3] describe both a Smalltalk-style approach and a CLOS-style multiple inheritance approach for modeling single inheritance and **`super`**. The paper uses a notion of implementation "deltas" $\Delta$, which are not first-class and only used for explanation. In our approach, this notion of deltas exists as a first-class entity which we call *mixins*. Bracha and

Cook describe mixins as a form of *abstraction* (over an unknown base class), and linearization as *application* (wiring in all the base classes), by analogy with the classical lambda calculus concepts. In our approach, abstraction is similarly done through `super` and application is done through `extends`, but we do not require linearization and allow mixins to be inherited an arbitrary number of times. While Bracha and Cook leverage the notion that subtyping is not inheritance and allow the types of methods to change, they do not support the idea of precise `this` and `super` annotations and thus cannot precisely type open recursion.

The concept of "mixin" described by Flatt et al. [13, 14, 12] is related to ours, but conceptually different. While they do model `super`, their mixins necessarily conform to interfaces and are thus constrained to specific method signatures, preventing SuperOOP-style modular programming. The authors discuss the possibility of solving the EP with modules and their mixins in later work [11], but without proposing a static typing model.

Schärli et al. [29] study and discuss many perceived problems with mixin composition. They suggest that *traits* are a better unit of abstraction. We agree that traits are useful for architecting OOP code in the large, but argue that mixins are independently useful: abstract (i.e., open-ended) base classes are specifically what unlocks the expressiveness of mixin inheritance and our new solution to the Expression Problem. We believe that mixins should be conceptualized as pure *whitebox implementation* bundles (the implementation itself being the API) by contrast with interfaces, which hide implementation detail, and traits, which enable a form of well-behaved (associative and commutative) multiple inheritance, and that all three could have a place in an OO programmer's toolkit.

The idea of separating reusable components from types was previously embraced by Bettini et al. [2], who argue that the role of *units of reuse* and the role of *types* are competing, as also observed by Cook et al. [7] and Snyder [30]. The semantics of Bettini et al.'s trait systems are similar to Schärli et al.'s but provide additional flexibility, in that traits are composed with explicit operations on methods such as renaming and exclusion to resolve conflict. A similar idea is used by Damiani et al. [8] in their design of a language enabling both trait reuse and *deltas* of classes, in the context of Software Product Line Engineering.

Type classes as in languages like Haskell [27] and Scala [24] also provide *data abstraction* and powerful parametrization and extensibility [6]. SuperOOP's `super` is a way of *nesting* interpretations the same way one can design dependent type class instances. Any class hierarchy encoded solely with `super` refinements in SuperOOP translates straightforwardly to classic type classes. However, type classes *per se* are not enough for modular code reuse with recursive data structures, as that requires open recursion. Explicit encodings of open recursion can be implemented in Haskell and Scala, but these would live outside of the type class definitions and are orthogonal to type classes. By contrast, SuperOOP directly provides precisely-typed open recursion via `this` refinements in mixins.

## 5 Conclusion and Future Work

We presented a new approach to OOP which cleanly separates the concerns of *state*, *implementations*, and *interfaces* into the orthogonal constructs of *classes*, *mixins*, and *interfaces*. We showed that a refined typing of mixins allows for a new and powerful solution to the expression problem. Finally, we presented an implementation in MLscript, leveraging its flexible type inference capabilities to enable annotation-free modular programming. The main item of future work we would like to look into is the *deep* composition of mixin *families*, reminiscent of Delta-Oriented Programming [28, 8] but with precisely-typed open recursion.

## References


1 Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for dylan. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 69–82, New York, NY, USA, 1996. Association for Computing Machinery. doi:10.1145/236337.236343.

2 Lorenzo Bettini, Ferruccio Damiani, Ina Schaefer, and Fabio Strocco. Traitrecordj: A programming language with traits and records. *Science of Computer Programming*, 78(5):521–541, 2013. Special section: Principles and Practice of Programming in Java 2009/2010 & Special section: Self-Organizing Coordination. doi:10.1016/j.scico.2011.06.007.

3 Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA/ECOOP '90, pages 303–311, New York, NY, USA, 1990. Association for Computing Machinery. doi:10.1145/97945.97982.

4 Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, September 2009. doi:10.1017/S0956796809007205.

5 Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyen. Set-theoretic types for polymorphic variants. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 378–391, Nara, Japan, September 2016. Association for Computing Machinery. doi:10.1145/2951913.2951928.

6 William R. Cook. On understanding data abstraction, revisited. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 557–572, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1640089.1640133.

7 William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 125–135, New York, NY, USA, 1989. Association for Computing Machinery. doi:10.1145/96709.96721.

8 Ferruccio Damiani, Reiner Hähnle, Eduard Kamburjan, and Michael Lienhardt. A unified and formal programming model for deltas and traits. In Marieke Huisman and Julia Rubin, editors, *Fundamental Approaches to Software Engineering*, pages 424–441, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

9 Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 270–282, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1111037.1111062.

10 Andong Fan. Simple extensible programming through precisely-typed open recursion. In *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2022, pages 54–56, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3563768.3563951.

11 Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 94–104, New York, NY, USA, 1998. Association for Computing Machinery. doi:10.1145/289423.289432.

12 Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In Naoki Kobayashi, editor, *Programming Languages and Systems*, pages 270–289, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

13 Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 171–183, New York, NY, USA, 1998. Association for Computing Machinery. doi:10.1145/268946.268961.

**14** Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. *A Programmer's Reduction Semantics for Classes and Mixins*, pages 241–269. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. doi:10.1007/3-540-48737-9_7.

**15** Jacques Garrigue. Programming with polymorphic variants. In *In ACM Workshop on ML*, 1998. URL: https://caml.inria.fr/pub/papers/garrigue-polymorphic_variants-ml98.pdf.

**16** Jacques Garrigue. Code reuse through polymorphic variants. In *In Workshop on Foundations of Software Engineering*, 2000. URL: https://www.math.nagoya-u.ac.jp/~garrigue/papers/variant-reuse.pdf.

**17** David S. Goldberg, Robert Bruce Findler, and Matthew Flatt. Super and inner: Together at last! In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 116–129, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/1028976.1028987.

**18** Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. Featherweight go. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi:10.1145/3428217.

**19** Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001. doi:10.1145/503502.503505.

**20** Guillaume Martres. Pathless scala: A calculus for the rest of scala. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Scala*, SCALA 2021, pages 12–21, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3486610.3486894.

**21** Keiko Nakata and Jacques Garrigue. Recursive modules for programming. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 74–86, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1159803.1159813.

**22** Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language, 2004.

**23** Bruno C. d. S. Oliveira and William R. Cook. Extensibility for the masses: Practical extensibility with object algebras. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 2–27, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-31057-7_2.

**24** Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 341–360, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1869459.1869489.

**25** Lionel Parreaux. The ultimate conditional syntax. *ML Family Workshop*, 2022. URL: https://icfp22.sigplan.org/details/mlfamilyworkshop-2022-papers/6/The-Ultimate-Conditional-Syntax.

**26** Lionel Parreaux and Chun Yin Chau. MLstruct: Principal type inference in a boolean algebra of structural types. *Proc. ACM Program. Lang.*, 6(OOPSLA2), October 2022. doi:10.1145/3563304.

**27** Simon Peyton Jones. Haskell 98 language and libraries: the revised report. *Journal of Functional Programming*, 13, January 2003.

**28** Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond*, pages 77–91, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

**29** Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In Luca Cardelli, editor, *ECOOP 2003 – Object-Oriented Programming*, pages 248–274, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

30 Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '86, pages 38–45, New York, NY, USA, 1986. Association for Computing Machinery. doi:10.1145/28697.28702.

31 Yaozhu Sun, Utkarsh Dhandhania, and Bruno C. d. S. Oliveira. Compositional embeddings of domain-specific languages. *Proc. ACM Program. Lang.*, 6(OOPSLA2), October 2022. doi:10.1145/3563294.

32 Andrew K. Wright. Simple imperative polymorphism. *LISP and Symbolic Computation*, 8(4):343–355, December 1995. doi:10.1007/BF01018828.

33 Matthias Zenger and Martin Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 241–252, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/507635.507665.

34 Weixin Zhang, Yaozhu Sun, and Bruno C. D. S. Oliveira. Compositional programming. *ACM Trans. Program. Lang. Syst.*, 43(3), September 2021. doi:10.1145/3460228.