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

Related Version *Short Version*: https://doi.org/10.4230/LIPIcs.ECOOP.2023.11

Supplementary Material Supplements can be found as follows:
Software (ECOOP 2023 approved artifact): https://doi.org/10.4230/DARTS.9.2.22
Software (Open-source implementation): https://github.com/hkust-taco/superoop
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 [14].

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 [21]. 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 [3]. 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)
```

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

[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.

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 the concept of 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: Str }

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

We define an interface specifying that a `Colored` object should contain a `color` method or field of type `Str`. 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: Str)
    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's type. 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.'s famous assertion that *inheritance is not subtyping* [8].

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] }
```

[2] A module declares a class with a singleton instance, similar to Scala's **`object`**.
[3] `Option[A]` is defined as the usual algebraic data type, with cases `Some[A](value: A)` and `None`.

```
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) and
      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: Str, 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 [32]. Using this approach, all the types can be inferred automatically as long as they do not involve first-class polymorphism (which may require 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) =
```

---

[4] Such separation of concerns was already proposed by previous authors, such as Bettini et al. [3] and Damiani et al. [10], 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 $\lambda^{\textsf{super}}$ calculus through interface inheritance and without requiring class inheritance.

```
    super.compare(lhs, rhs) and 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 $\lambda^{\textsf{super}}$. We use big-step semantics to closely reflect a real implementation and prove the soundness of $\lambda^{\textsf{super}}$ 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 discuss several important approaches from previous literature on the topic of inheritance and the Expression Problem, explaining 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 class and mixin definitions. Both the open-source repository and the archived artifact with documentation are available online. 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 "**super**-charged" MLscript.

The Expression Problem and Extensible Variants

In the field of modular programming, the *Expression Problem* (EP) [38] 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 [20] 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.

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.

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

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 [31]. 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] [32]. 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 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

---

[7] Constructor difference is not a primitive construct of MLscript's underlying type system, MLstruct [32]. 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.

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 this partially-specified recursive function to different contexts, as we shall see shortly.

Closing back. We can immediately tie the knot and obtain an implementation equivalent 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 some 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 [20], 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:

```
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 λ^{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 through a linear composition of mixins and a set of parameters which represents the *state* of the object.

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. Consider the following example with generics, where `Option` is the usual option type with constructors `Some` and `None` (here, **`implements`** `{ foo: ... }` is a shorthand for defining an unnamed interface and adding it to the **`implements`** clause):

```
mixin Foo { fun foo: Int = 1 }
mixin Bar[A] {
  super: { foo: A }
  fun foo: Option[A] = new Some[A](super.foo)
}
```

```
class ClsFoo extends Foo, Bar[Int], Bar[Option[Int]]
             implements { foo: Option[Option[Int]] }
```

Generic mixin `Bar[A]` overrides the implementation of `foo` by wrapping the parent implementation of `foo` (i.e. `super.foo`) with the `Some` constructor. Importantly, `Bar[A]` has the type of `super` refined as `{ foo: A }`, which gives `super.foo` type `A`. In this example, mixin `Bar[Int]` overrides method `foo` of type `Int` in mixin `Foo`, and `Bar[Option[Int]]` overrides `foo` of type `Option[Int]` provided by the inherited mixin composition `(Foo, Bar[Int])`.

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' 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, 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. Consider the following example:

```
mixin Mxn1 {
  this: { a: Str }
  fun a: Boolean = (this.a == "42")
}
mixin Mxn2 { fun a: Str = "42" }
class Cls extends Mxn1, Mxn2 implements { a: Str }
```

Method `a` has type `Boolean` in mixin `Mxn1`. The annotated precise type of `this` gives `this.a` type `Str` in `Mxn1`, allowing string comparison in `a`'s implementation. Mixin `Mx2` provides `a`'s implementation of type `Str`. Class `C` implements `a: Str` by the mixin composition `(Mxn1, Mxn2)`. The inheritance of `Mxn1` is allowed since the interface that class `Cls` implements matches the annotated `this` type in `Mxn1`. In `Mxn2`, we could still access the super implementation of `a` in `Mxn1` by refining the type of `super`.

3.2 Formal Syntax

We now introduce the $\lambda^{\textsf{super}}$ calculus, a formalization of SuperOOP. The design of this calculus is inspired by Featherweight Generic Java [23] and Pathless Scala [25]. 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 $\lambda^{\textsf{super}}$ 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 $\textsf{Object}$. 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

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 $\lambda^{\textsf{super}}$.

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.

The top-level definitions of $\lambda^{\textsf{super}}$ 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 $\lambda^{\textsf{super}}$ which includes a declarative subtyping, term typing, and well-formedness check of top-level definitions.

Declarative subtyping. Figure 2 shows the declarative subtyping of $\lambda^{\textsf{super}}$. 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 Figure 9 of Appendix A) returns the list of parent interfaces. For simplicity, we consider that interfaces are *invariant* in their type

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

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

$$\text{S-AndL}\quad \frac{}{S_1 \,\&\, S_2 <: S_1} \qquad \text{S-AndR}\quad \frac{}{S_1 \,\&\, S_2 <: S_2} \qquad \text{S-And}\quad \frac{S <: T_1 \quad S <: T_2}{S <: T_1 \,\&\, T_2} \qquad \text{S-Trans}\quad \frac{S <: U \quad U <: T}{S <: T}$$

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

Figure 2 Declarative subtyping.

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 Γ 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 calls 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 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 Figure 9 of Appendix A). 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 (vparams and ctype are defined in Figure 9).

The design of mtype basically follows that of Pathless Scala [25]. When a method signature is present in an interface, we directly return it. Otherwise, we search parent interfaces by calling mtype with the *intersection* of all parent interfaces (denoted as $\&\overline{J[\overline{U}]}$). Note that nullary intersection is 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.

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. Note that we bind **this** to a type while **super** to a structural refinement in each mixin. This syntactic difference is rooted in the semantic difference between them. People often call **super** a "pseudo-variable" because it is merely a reference to call methods inherited from the parent class or mixin. In SuperOOP, the parent mixin in the composition hierarchy does *not* define

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

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

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

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

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

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

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

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

Figure 3 Term typing.

an object type, and **super** should not be passed around. 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 an object that has a type, can be passed around, and receives method calls. Hence **this** is annotated with a type, and the annotated type should be a supertype of the later defined class's type.

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.

For **1.**, I ***ok*** checks the interface, and $\overline{M\ \textit{ok}}$ 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

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

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

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

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

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

$\overline{M_i[\overline{S}]}^{i \in n..1}$), 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 mnames, defined in Figure 9 of Appendix A), 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 mtype).

Method implementation type search. Figure 5 defines function $\mathsf{search}(m, i, C)$ to search the 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 the current 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 that m is not implemented in the class so the method implementation search fails.

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 $M_i[\overline{Y}]^{\mathcal{R}}_{\overline{V}}\ \mathcal{I}$:

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

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

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

Figure 5 Method implementation type search function.

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

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

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

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

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

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

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

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

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

Figure 6 Big-step operational semantics producing values.

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 Figure 10 in Appendix A). The big-step evaluation judgment $\Xi \vdash e \Downarrow r$ denotes that term e evaluates to result r under runtime context Ξ. The result of evaluation may be a normal value or an error. Values are either *closures* or *objects*. A runtime context Ξ 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, we use a type substitution in the semantics, which 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 no substitution. Note that the evaluation rule for **this** simply reads the configured object from the context and returns a plain object (i.e., with no search index). Class instantiations produce objects. Lambda and type abstractions are evaluated to closures. Note that $\lambda^{\textsf{super}}$ would not need a value restriction [40] 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).
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 Figure 9 of Appendix A) 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).

Configured object $\{i \star C[\overline{S}](\overline{v})\}$ is bound to **this** in the context; it specifies the object in which we search for method implementations as well as the current *level* of that search. When the configuring index is nil ($i = 0$), we search the object fields. Otherwise ($i > 0$), we search the i-th mixin, counting from the *rightmost* composed mixin of the object's class.

In rule E-Access, we evaluate the receiver to an object and trigger method implementation search by evaluating **super**.m with **this** bound to the object configured by 0, i.e., we start the search with the object fields. Two sets of Miss/Hit rules evaluate method invocations on **super**. Rules E-ArgMiss/Hit consider the object fields. If the method name is found in the constructor parameter list of the class (computed by $\mathsf{vparams}$), the corresponding value is returned. Otherwise, we search the mixin composition by incrementing the configuring index to 1 and recursively evaluating **super**.m. Rules E-SuperMiss/Hit deal with method calls on **super** when the configuring index is non-zero. If the method m is implemented in the i-th mixin, we evaluate the method body. If the implementation is missing, we search the next mixin by incrementing the configuring index to $i + 1$. Note that it is safe to drop

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

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

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

Figure 7 Value typing of closures.

the context (save for the binding of the configured object to **this**) in rules E-ArgMiss and E-SuperMiss since the method body is always evaluated under a context containing only a binding from **this** to the configured object. If the method search fails, an error is produced. We do not need the call-site runtime environment in either case.

3.5 Metatheory

We now develop the metatheory of $\lambda^{\textsf{super}}$. We follow Ernst et al.'s approach to prove type soundness of our big-step style semantics. Different from soundness proof for small-step semantics, runtime error and divergence both lead to the non-existence of evaluation derivation in a big-step semantics. Therefore, soundness proof of big-step semantics requires special treatment to model divergence of term evaluation, discriminating runtime error and divergence. To solve the problem, an evaluation result is first divided into two: a value or a runtime error. The static type system should guarantee that, if a well-typed term evaluates to a result, it is always a value, and the result value preserves the term's type. This is called *preservation*. To handle divergence, evaluation is indexed by *fuel*. Each step of evaluation consumes one unit of fuel. When the fuel runs out, the evaluation terminates and returns a timeout result. This means we may always construct a *finite* derivation when evaluating any term. When a term evaluates to a timeout result regardless of fuel amount, it is said to diverge. Now we can model soundness of big-step semantics: a well-typed term evaluates to a value or it diverges. Note that preservation itself does not lead to soundness. To guarantee that any term evaluates to a result, we need a *coverage* lemma to rule out the situation when a term cannot be evaluated because of missing evaluation rules (preservation is vacuously true in this case). With both preservation and coverage, we have type soundness for a big-step 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 Γ 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 Figure 8), 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 Figure 8 of Appendix A) 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 Appendix B. 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 [13] 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 Figure 11 of Appendix A). Finally, add the following axiom:*

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

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, Ξ, 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 $\overline{\mathcal{D}_\mathcal{P}}$ **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 [27]. The linearization of a Scala class definition of the form `class C extends` B_0, B_1, ..., B_n starts with B_0's linearization and appends to it the linearization of B_1 save for those traits that are already in the constructed linearization of B_0, etc. Several languages such as Python adopt the influential C3 linearization algorithm [2]. 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.

Example encoding of Scala trait inheritance. The following example shows the idea of encoding Scala multiple trait inheritance in SuperOOP. Consider the following simple Scala code:

```
trait A { def a = 0 }
trait L extends A { override def a = 42 }
trait R(foo: Int) extends A { override def a = foo }
class LR(foo: Int) extends L, R(2 * foo)
```

There is a base trait `A` and two derived traits `L` and `R` overriding the base implementation of `a`. Note that `R` accesses its trait parameter. In SuperOOP, we can equivalently have:

```
mixin A { fun a = 0 }
mixin L { fun a = 42 }
interface I$R { foo$R: Int }
// R accesses its local parameter via a unique name
mixin R { this: I$R; fun a = this.foo$R }
interface I extends I$R { foo$LR: Int }
// R's local parameter is finally provided by the class
mixin $R { this: I; fun foo$R = this.foo$LR * 2 }
// LR is composed by the linearization of LR in Scala
class LR(foo$LR: Int) extends I, A, L, R, $R
```

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.

Cached interpretation. Consider the following code adapted from the "linguistic reuse" example of Sun et al. [37]:

```
fun go(x) =
  if x is 0 then Lit(1)
  else
    let shared = go(x - 1)
    Add(shared, shared)
```

The execution of `eval(go(`n`))` requires 2^n computation time because, in SuperOOP, embedded language data types are represented by class definitions, that is, data structures are objects. The evaluation of the `shared` expression nodes is repeated when interpreting `Add`, even though the `shared` node itself is indeed shared on both sides of the `Add` node. Such representation is known as *initial embeddings* [5]. On the contrary, in some other extensible programming paradigms, such as object algebras [29], Compositional Programming [42], and tagless-final [5], expression definitions have an unspecified final interpretation type. Under such embeddings (known as *final embeddings* in the tagless-final work [5]), when interpreting an expression constructed by the `go` function above, the `shared` variable would hold the final interpretation result (in our example, `shared` would be an integer), so the computation time would be linear in n.

We could avoid repeated computations and reuse the interpreted results in SuperOOP by *caching* these results in the expression data types. With *family polymorphism* [12], we could modularly add new cached interpretations, as exemplified below:

```
mixin BaseMxn {
  class Base
}

mixin Example {
  virtual class Base
  class Lit(n: Int) extends Base
  class Add[A](lhs: A, rhs: A) extends Base

  fun go(x) = ... // as defined above
  val exp = go(10)
}

mixin CacheSize {
  fun sizeImpl(x) = if x is
    Lit(n) then 1
```

```
    Add(lhs, rhs) then lhs.size + rhs.size
  override class Base { lazy val size = this.sizeImpl(this) }
}

class LangBase extends BaseMxn, Example
module Lang extends LangBase, CacheSize
Lang.exp.size
```

The data classes in mixin `Example` extend a virtual class `Base`. `Base` can be overriden to incrementally extend expressions with new cached interpretations. For example, to modularly extend the language with pretty-printing, we could have:

```
mixin CachePrint {
  fun printImpl(x) = if x is
    Lit(n) then n.toString
    Add(lhs, rhs) then lhs.print ++ rhs.print
  override class Base { lazy val print = this.printImpl(this) }
}

module Lang' extends Lang, CachePrint
Lang'.exp.size
Lang'.exp.print
```

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 [20]. 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.

Type inference. Our proposed novel OOP model SuperOOP is the latest evolution of the MLscript programming language, which enables our elegant solution to the Expression Problem, as shown in Section 2. MLscript and its underlying core type system MLstruct [32] feature *principal* polymorphic type inference for structural types, i.e., types with functions, records, first-class unions and intersections, enabling extensible variants without needing row polymorphism. MLstruct itself evolved from a simplified take [30] on the groundbreaking *algebraic subtyping* approach [11], whereby an algebraic take on the semantics of types enables principality of type inference, notably without backtracking in the type checker, which helps with the scalability of our approach.

Case studies. In Appendix C, we provide case studies of MLscript/SuperOOP that include a modular evaluator of extended lambda calculus, as described by Garrigue [20], and a simple "regions" DSL developed by Sun et al. [37]. 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 [32], those case studies type check without the help of a single type annotation (except for class field definitions).

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 [19] in OCaml, recursive modules [26] in ML, new programming paradigms [5, 29] like Compositional Programming [42], and covariant class field type refinement in Scala [39]. 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 [20] 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 [6]. Those drawbacks can be fixed by embracing "proper" implicit subtyping as in MLscript [32]. In particular, we argue that union types are simpler than row polymorphism, which imperfectly emulates subtyping through unification [32].

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. For example, we first define a base class with the receiver's type refined:

```
class ['a] base = object (self: < eval: 'a → int; .. >)
method eval = function
  | ‘Lit n → n
  | ‘Add(a, b) → self#eval a + self#eval b
```

```
end
(*  class ['a] base :
    object
      constraint 'a = [< ‘Add of 'a * 'a | ‘Lit of int ]
      method eval : 'a → int
    end *)
```

Note that the recursive knot of the expression type has already been tied as OCaml generates a constraint that `'a = [< ‘Add of 'a * 'a | ‘Lit of int ]`. If we try to extend the base class with evaluation on a new data type variant, OCaml will raise a unification error:

```
class ['a] ext = object (self) inherit ['a] base as super
method eval = function
  | ‘Neg e → 0 - self#eval e
  | e → super#eval e
end
(* Error: This pattern matches values of type [> ‘Neg of 'a ]
        but a pattern was expected which matches values of type
          [< ‘Add of 'b * 'b | ‘Lit of int ] as 'b
        The second variant type does not allow tag(s) ‘Neg *)
```

Here OCaml tries to *unify* the type of `eval` in the base and extended implementation, i.e., `this#eval` and `super#eval`. The unification fails as `super#eval`'s type is already closed, that is, it only accepts expressions constructed by `Add` and `Lit`, but `this#eval` is trying to match expressions of type `[> ‘Neg of 'a ]`. One may fix this unification error by extending pattern matching of the base implementation with a default case, therefore opening the type of `eval` in the base class. However, statically this fix will make the base implementation accept any expression variant that may not be handled by derived implementations.

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. [22] 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 (i.e. covariant receiver type refinements) are not part of the Go team's current design for generics [22]. 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 [37], 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, like in the following example encoding:

```
type Normer(type a Normer(a)) interface {
  Norm() a
  NormDele() a
}

func (e Lit) Norm() Lit {
  return e
}

func (e Lit) NormDele() Neg(Lit) {
  return Neg(Lit){e}
}

func (e Neg(type a Normer(a))) Norm() a {
  return e.expr.NormDele()
}

func (e Neg(type a Normer(a))) NormDele() a {
  return e.expr
}

type Expr interface {
  Evaler
  Stringer
  Normer(Expr)
}

Neg(Expr){Neg(Expr){Lit{1}}}.Norm().Eval()
```

However, this encoding does not really work in the presented FGG system and its implementation as `Lit` is not considered to implement `Expr`, because its `Norm` implementation is not exactly returning `Expr`.

Object Algebras. *Object Algebras* are a well-known object-oriented approach to solve the Expression Problem [29]. 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. [42], 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* [42] (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. [37], 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 [41] 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. [8] 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 [4] 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" Δ, 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. [16, 17, 18] 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 [15], but without proposing a static typing model.

Schärli et al. [35] 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. [3], who argue that the role of *units of reuse* and the role of *types* are competing, as also observed by Cook et al. [8] and Snyder [36]. 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. [10] 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 [33] and Scala [28] also provide *data abstraction* and powerful parametrization and extensibility [7]. 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. As Oleg Kiselyov put it in his lecture on modular tagless-final interpreters [24]:

> *To be able to extend our de-serializer, we have to write it in the open recursion style.*

> *It is a bit unfortunate that we have to anticipate extensibility; alas, open recursion seems unavoidable for any extensible inductive de-serializer.*

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.

4.5 Big-Step Semantics and Its Soundness

Due to being close to an interpreter, big-step semantics have been proposed as a more natural and informative way of formalizing program semantics. We found that it is particularly appropriate once the language starts deviating significantly from simple variations of pure lambda calculus. In particular, the representation of **super** method lookup, which relies on having a *configured object* bound in the current runtime context, would be particularly easier to formalize using big-step semantics. Dagnino et al. [9] describes a great overview of modeling divergence and proof of type soundness of big-step semantics. Amin and Rompf's work [1] proves soundness of F-sub in a big-step semantics. Jeremy Siek's article is a good short introduction to prove type safety of big-step semantics using fuel.[11] It summarizes:

> *In general, the solutions to proving big-step soundness seem to fall into three categories: (1) introduce a separate co-inductive definition of divergence for the language in question; (2) develop a notion of partial derivations; and (3) a time counter that causes a time-out error when it gets to zero. — that is, make the semantics step-indexed.*

Our approach follows Ernst et al. [13], which uses fuel. Meanwhile, Ernst et al. provide a clear explanation of why fuel (or somehow to separate stuck terms from divergent terms) is necessary in their paper.

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 [10, 34] but with precisely-typed open recursion, as exemplified by the following code:

```
mixin Base {
  class Foo(x: Int)
}
mixin Derived1 {
  mixin Foo {
    fun get = this.x + 1
  }
}
mixin Derived2 {
```

[11] http://siek.blogspot.com/2012/07/big-step-diverging-or-stuck.html

```
  mixin Foo {
    fun get = super.get * 2
  }
}
```

whose inferred types would be:

```
class Base: {
  class Foo(x: Int): {}
}
mixin Derived1: {
  mixin Foo: {
    this: { x: Int }
    get: Int
  }
}
mixin Derived2 {
  mixin Foo: {
    super: { get: Int }
    get: Int
  }
}
```

An example usage of this could be, for instance:

```
class Final extends Base, Derived1, Derived2
Final.Foo(3).get // 8
```

References

1 Nada Amin and Tiark Rompf. 2017. Type Soundness Proofs with Definitional Interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 666–679. https://doi.org/10.1145/3009837.3009866

2 Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. 1996. A Monotonic Superclass Linearization for Dylan. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Jose, California, USA) *(OOPSLA '96)*. Association for Computing Machinery, New York, NY, USA, 69–82. https://doi.org/10.1145/236337.236343

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

4 Gilad Bracha and William Cook. 1990. Mixin-Based Inheritance. In *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications* (Ottawa, Canada) *(OOPSLA/ECOOP '90)*. Association for Computing Machinery, New York, NY, USA, 303–311. https://doi.org/10.1145/97945.97982

5 Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *J. Funct. Program.* 19, 5 (sep 2009), 509–543. https://doi.org/10.1017/S0956796809007205

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

7 William R. Cook. 2009. On Understanding Data Abstraction, Revisited. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 557–572. `https://doi.org/10.1145/1640089.1640133`

8 William R. Cook, Walter Hill, and Peter S. Canning. 1989. Inheritance is Not Subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '90)*. Association for Computing Machinery, New York, NY, USA, 125–135. `https://doi.org/10.1145/96709.96721`

9 Francesco Dagnino, Viviana Bono, Elena Zucca, and Mariangiola Dezani-Ciancaglini. 2020. Soundness Conditions for Big-Step Semantics. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 169–196.

10 Ferruccio Damiani, Reiner Hähnle, Eduard Kamburjan, and Michael Lienhardt. 2017. A Unified and Formal Programming Model for Deltas and Traits. In *Fundamental Approaches to Software Engineering*, Marieke Huisman and Julia Rubin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 424–441.

11 Stephen Dolan. 2017. *Algebraic subtyping.* Ph. D. Dissertation.

12 Erik Ernst. 2001. Family Polymorphism. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*. Springer-Verlag, Berlin, Heidelberg, 303–326.

13 Erik Ernst, Klaus Ostermann, and William R. Cook. 2006. A Virtual Class Calculus. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '06)*. Association for Computing Machinery, New York, NY, USA, 270–282. `https://doi.org/10.1145/1111037.1111062`

14 Andong Fan. 2022. 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* (Auckland, New Zealand) *(SPLASH Companion 2022)*. Association for Computing Machinery, New York, NY, USA, 54–56. `https://doi.org/10.1145/3563768.3563951`

15 Robert Bruce Findler and Matthew Flatt. 1998. Modular Object-Oriented Programming with Units and Mixins. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '98)*. Association for Computing Machinery, New York, NY, USA, 94–104. `https://doi.org/10.1145/289423.289432`

16 Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. 2006. Scheme with Classes, Mixins, and Traits. In *Programming Languages and Systems*, Naoki Kobayashi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 270–289.

17 Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1998. Classes and Mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '98)*. Association for Computing Machinery, New York, NY, USA, 171–183. `https://doi.org/10.1145/268946.268961`

18 Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1999. *A Program-*

mer's Reduction Semantics for Classes and Mixins. Springer Berlin Heidelberg, Berlin, Heidelberg, 241–269. https://doi.org/10.1007/3-540-48737-9_7

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

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

21 David S. Goldberg, Robert Bruce Findler, and Matthew Flatt. 2004. Super and Inner: Together at Last!. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, BC, Canada) *(OOPSLA '04)*. Association for Computing Machinery, New York, NY, USA, 116–129. https://doi.org/10.1145/1028976.1028987

22 Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. 2020. Featherweight Go. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 149 (nov 2020), 29 pages. https://doi.org/10.1145/3428217

23 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (May 2001), 396–450. https://doi.org/10.1145/503502.503505

24 Oleg Kiselyov. 2012. *Typed Tagless Final Interpreters*. Springer Berlin Heidelberg, Berlin, Heidelberg, 130–174. https://doi.org/10.1007/978-3-642-32202-0_3

25 Guillaume Martres. 2021. Pathless Scala: A Calculus for the Rest of Scala. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Scala* (Chicago, IL, USA) *(SCALA 2021)*. Association for Computing Machinery, New York, NY, USA, 12–21. https://doi.org/10.1145/3486610.3486894

26 Keiko Nakata and Jacques Garrigue. 2006. Recursive Modules for Programming. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming* (Portland, Oregon, USA) *(ICFP '06)*. Association for Computing Machinery, New York, NY, USA, 74–86. https://doi.org/10.1145/1159803.1159813

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

28 Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. 2010. Type Classes as Objects and Implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. Association for Computing Machinery, New York, NY, USA, 341–360. https://doi.org/10.1145/1869459.1869489

29 Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses: Practical Extensibility with Object Algebras. In *Proceedings of the 26th European Conference on Object-Oriented Programming* (Beijing, China) *(ECOOP'12)*. Springer-Verlag, Berlin, Heidelberg, 2–27. https://doi.org/10.1007/978-3-642-31057-7_2

30 Lionel Parreaux. 2020. The Simple Essence of Algebraic Subtyping: Principal Type Inference with Subtyping Made Easy (Functional Pearl). *Proc. ACM Program. Lang.* 4, ICFP, Article 124 (Aug. 2020), 28 pages. https://doi.org/10.1145/3409006

31 Lionel Parreaux. 2022. The Ultimate Conditional Syntax. *ML Family Workshop* (2022). https://icfp22.sigplan.org/details/mlfamilyworkshop-2022-papers/6/The-Ultimate-Conditional-Syntax

32 Lionel Parreaux and Chun Yin Chau. 2022. MLstruct: Principal Type Inference in a Boolean Algebra of Structural Types. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 141 (Oct 2022), 30 pages. https://doi.org/10.1145/3563304
33 Simon Peyton Jones. 2003. Haskell 98 Language and Libraries: the Revised Report. *Journal of Functional Programming* 13 (01 2003).
34 Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-Oriented Programming of Software Product Lines. In *Software Product Lines: Going Beyond*, Jan Bosch and Jaejoon Lee (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 77–91.
35 Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. 2003. Traits: Composable Units of Behaviour. In *ECOOP 2003 – Object-Oriented Programming*, Luca Cardelli (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 248–274.
36 Alan Snyder. 1986. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '86)*. Association for Computing Machinery, New York, NY, USA, 38–45. https://doi.org/10.1145/28697.28702
37 Yaozhu Sun, Utkarsh Dhandhania, and Bruno C. d. S. Oliveira. 2022. Compositional Embeddings of Domain-Specific Languages. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 131 (oct 2022), 29 pages. https://doi.org/10.1145/3563294
38 Philip Wadler. 1998. The expression problem. Posted on the Java Genericity mailing list. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt
39 Yanlin Wang and Bruno C. d. S. Oliveira. 2016. The Expression Problem, Trivially!. In *Proceedings of the 15th International Conference on Modularity* (Málaga, Spain) *(MODULARITY 2016)*. Association for Computing Machinery, New York, NY, USA, 37–41. https://doi.org/10.1145/2889443.2889448
40 Andrew K. Wright. 1995. Simple imperative polymorphism. *LISP and Symbolic Computation* 8, 4 (01 Dec 1995), 343–355. https://doi.org/10.1007/BF01018828
41 Matthias Zenger and Martin Odersky. 2001. Extensible Algebraic Datatypes with Defaults. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming* (Florence, Italy) *(ICFP '01)*. Association for Computing Machinery, New York, NY, USA, 241–252. https://doi.org/10.1145/507635.507665
42 Weixin Zhang, Yaozhu Sun, and Bruno C. D. S. Oliveira. 2021. Compositional Programming. *ACM Trans. Program. Lang. Syst.* 43, 3, Article 9 (sep 2021), 61 pages. https://doi.org/10.1145/3460228

$$\boxed{v : T} \qquad \frac{\text{vparams}(C[\overline{T}]) = \overline{m : U} \quad \overline{v : U} \quad \text{ctype}(C[\overline{T}]) = V}{C[\overline{T}](\overline{v}) : V}\ \text{VT-Object}$$

$$\frac{\Gamma \vDash \Xi \quad \textbf{this} \notin dom(\Gamma) \quad \Gamma, x : S \vdash e : T}{\langle \lambda x : S.\ e, \Xi \rangle : S \to T}\ \text{VT-Abs2} \qquad \frac{\Gamma \vDash \Xi \quad \textbf{this} \notin dom(\Gamma) \quad \Gamma \vdash e : T}{\langle \Lambda X.\ e, \Xi \rangle : \forall X.\ T}\ \text{VT-TAbs2}$$

$$\boxed{\Gamma \vDash \Xi} \qquad \frac{\Gamma \vDash \Xi \quad v : T}{\Gamma, x : T \vDash \Xi, x \mapsto v}\ \text{C-ConsVar} \qquad \frac{\Gamma \vDash \Xi \quad C[\overline{S}](\overline{v}) : T}{\Gamma, \textbf{this} : T \vDash \Xi, \textbf{this} \mapsto \{i \star C[\overline{S}](\overline{v})\}}\ \text{C-ConsThis} \qquad \frac{}{\epsilon \vDash \epsilon}\ \text{C-Nil}$$

Figure 8 Value typing (continued).

A Auxiliaries

Figure 8 shows the rest of the value typing rules. Rule VT-Abs2 deals with the case when **this** is unbound in closure's runtime context. In this case, no open-recursive calls are allowed in the lambda abstraction body, and closure typing is standard. Rules VT-TAbs1/2 work similarly on type abstraction closures. Object typing is non-surprising.

Figure 9 shows auxiliary definitions of $\lambda^{\textsf{super}}$. Figure 10 shows big-step evaluation rules which produce error (**err**) results. Figure 11 shows finite big-step evaluation rules (defined by Definition 2) which produce timeout (**kill**) results.

B Full Metatheory of $\lambda^{\textsf{super}}$

B.1 Preservation

Subtyping

▶ **Lemma 6.** *The following forms of subtyping derivation are impossible:*
- $I[\overline{V}] <: \{S \to T,\ \forall X. U,\ Y\}$
- $S \to T <: \{\forall X. U,\ I[\overline{V}],\ Y\}$
- $\forall X. U <: \{S \to T,\ I[\overline{V}],\ Y\}$
- $Y <: \{I[\overline{V}],\ S \to T,\ \forall X. U\}$
- $\text{Object} <: \{I[\overline{V}],\ S \to T,\ \forall X. U,\ Y\}$

Proof. By induction on each form of the impossible subtyping. For case S-Trans, we prove by induction on the type in the middle. ◀

▶ **Lemma 7.** *If $U <: S \,\&\, T$ then $U <: S$ and $U <: T$.*

Proof. By induction on the subtyping derivation.
Case S-Refl Immediate.
Case S-Interface Impossible as parents only returns a list of parent interfaces.
Case S-And Immediate.
Case S-Trans By IH and S-Trans.

◀

$$\boxed{\mathsf{mnames}(T) := \overline{m}} \qquad \frac{I[\overline{X}] \lhd \overline{J[\overline{U}]}\ \{\ \overline{m[...] : ...}\ \} \quad \mathsf{mnames}(\overline{\&J[\overline{U}]}) = \overline{n}}{\mathsf{mnames}(I[\overline{T}]) := \overline{n}, \overline{m}} \qquad \frac{\mathsf{mnames}(S) = \overline{n} \quad \mathsf{mnames}(T) = \overline{m}}{\mathsf{mnames}(S\ \&\ T) := \overline{n}, \overline{m}}$$

$$\boxed{\mathsf{vparams}(C[\overline{T}]) := \overline{m : S}} \qquad \frac{C[\overline{X}](\overline{m : S}) \lhd ..., ...}{\mathsf{vparams}(C[\overline{T}]) := [\overline{T/X}]\overline{m : S}}$$

$$\boxed{\mathsf{ctype}(C[\overline{T}]) := S} \qquad \frac{C[\overline{X}](...) \lhd I[\overline{S}], ...}{\mathsf{ctype}(C[\overline{T}]) := [\overline{T/X}]I[\overline{S}]}$$

$$\boxed{\mathsf{parents}(I[\overline{T}]) := \overline{J[\overline{S}]}} \qquad \frac{I[\overline{X}] \lhd \overline{J[\overline{S}]}\ \{\ ...\ \}}{\mathsf{parents}(I[\overline{T}]) := \overline{[\overline{T/X}]J[\overline{S}]}}$$

$$\boxed{\mathsf{mrefn}(m, \mathcal{R}) := T} \qquad \frac{\mathcal{R} = \{\overline{m : T}\} \qquad m : T \in \overline{m : T}}{\mathsf{mrefn}(m, \mathcal{R}) := T}$$

Given that class C is defined as $C[\overline{X}](...) \lhd I[\overline{U'}], \overline{M_i[\overline{U}]}^{i \in n..1}$,
and mixin M_i is defined as $M_i[\overline{Y}]^{\mathcal{R}}_{V}\ \{\ \overline{m : T = e}\ \}$:

$$\mathsf{methods}(i, C[\overline{S}](\overline{v})) = (\overline{\,m : [\overline{S/X}][\overline{U/Y}]T = [\overline{S/Y}][\overline{U/Y}]e\,})$$

Figure 9 Auxiliaries.

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

$$\textsc{E-ErrVar}\quad \frac{x \notin dom(\Xi)}{\Xi \vdash x \Downarrow \textbf{err}}$$

$$\textsc{E-ErrThis}\quad \frac{\textsf{this} \notin dom(\Xi)}{\Xi \vdash \textsf{this} \Downarrow \textbf{err}}$$

$$\textsc{E-ErrSuper}\quad \frac{}{\Xi \vdash \textsf{super} \Downarrow \textbf{err}}$$

$$\textsc{E-ErrAccess1}\quad \frac{\Xi \vdash e \Downarrow \textbf{val}\ \langle \lambda x : T.\ e', ...\rangle}{\Xi \vdash e.m \Downarrow \textbf{err}}$$

$$\textsc{E-ErrAccess2}\quad \frac{\Xi \vdash e \Downarrow \textbf{val}\ \langle \Lambda X.\ e', ...\rangle}{\Xi \vdash e.m \Downarrow \textbf{err}}$$

$$\textsc{E-ErrSuperAccess1}\quad \frac{\textsf{this} \notin dom(\Xi)}{\Xi \vdash \textsf{super}.m \Downarrow \textbf{err}}$$

$$\textsc{E-ErrSuperAccess2}\quad \frac{\Xi(\textsf{this}) = \{i \star C[\overline{S}](\overline{v})\} \qquad i \neq 0 \qquad \textsf{methods}(i, C[\overline{S}](\overline{v}))\ \text{undefined}}{\Xi \vdash \textsf{super}.m \Downarrow \textbf{err}}$$

$$\textsc{E-ErrSuperAccess3}\quad \frac{\Xi(\textsf{this}) = \{i \star C[\overline{S}](\overline{v})\} \qquad i = 0 \qquad \textsf{vparams}(C[\overline{S}])\ \text{undefined}}{\Xi \vdash \textsf{super}.m \Downarrow \textbf{err}}$$

$$\textsc{E-ErrApp1}\quad \frac{\Xi \vdash e_1 \Downarrow \textbf{val}\ C[\overline{T}](\overline{v})}{\Xi \vdash e_1\ e_2 \Downarrow \textbf{err}}$$

$$\textsc{E-ErrApp2}\quad \frac{\Xi \vdash e_1 \Downarrow \textbf{val}\ \langle \Lambda X.\ e', ...\rangle}{\Xi \vdash e_1\ e_2 \Downarrow \textbf{err}}$$

$$\textsc{E-ErrTApp1}\quad \frac{\Xi \vdash e \Downarrow \textbf{val}\ C[\overline{T}](\overline{v})}{\Xi \vdash e\ T \Downarrow \textbf{err}}$$

$$\textsc{E-ErrTApp2}\quad \frac{\Xi \vdash e \Downarrow \textbf{val}\ \langle \lambda x : T.\ e', ...\rangle}{\Xi \vdash e\ T \Downarrow \textbf{err}}$$

$$\textsc{E-ErrNew}\quad \frac{|\textsf{vparams}(C[\overline{T}])| \neq n}{\Xi \vdash \textbf{new}\ C[\overline{T}](\overline{e_i}^{i \in 1..n}) \Downarrow \textbf{err}}$$

$$\textsc{E-ErrPropTApp1}\quad \frac{\Xi \vdash e \Downarrow \textbf{err}}{\Xi \vdash e\ T \Downarrow \textbf{err}}$$

$$\textsc{E-ErrPropTApp2}\quad \frac{\Xi \vdash e \Downarrow \textbf{val}\ \langle \Lambda X.\ e',\ \Xi'\rangle \qquad [T/X]\Xi' \vdash [T/X]e' \Downarrow \textbf{err}}{\Xi \vdash e\ T \Downarrow \textbf{err}}$$

$$\textsc{E-ErrPropApp1}\quad \frac{\Xi \vdash e_1 \Downarrow \textbf{err}}{\Xi \vdash e_1\ e_2 \Downarrow \textbf{err}}$$

$$\textsc{E-ErrPropApp2}\quad \frac{\Xi \vdash e_2 \Downarrow \textbf{err}}{\Xi \vdash e_1\ e_2 \Downarrow \textbf{err}}$$

$$\textsc{E-ErrPropApp3}\quad \frac{\Xi \vdash e_1 \Downarrow \textbf{val}\ \langle \lambda x : T.\ e,\ \Xi'\rangle \qquad \Xi \vdash e_2 \Downarrow \textbf{val}\ v \qquad \Xi',\ x \mapsto v \vdash e \Downarrow \textbf{err}}{\Xi \vdash e_1\ e_2 \Downarrow \textbf{err}}$$

$$\textsc{E-ErrPropNew}\quad \frac{\Xi \vdash e_i \Downarrow \textbf{err}}{\Xi \vdash \textbf{new}\ C[\overline{T}](\overline{e_i}^{i \in 1..n}) \Downarrow \textbf{err}}$$

$$\textsc{E-ErrPropAccess1}\quad \frac{e \neq \textsf{super} \qquad \Xi \vdash e \Downarrow \textbf{err}}{\Xi \vdash e.m \Downarrow \textbf{err}}$$

$$\textsc{E-ErrPropAccess2}\quad \frac{\Xi \vdash e \Downarrow \textbf{val}\ C[\overline{S}](\overline{v}) \qquad (\textsf{this} \mapsto \{0 \star C[\overline{S}](\overline{v})\}) \vdash \textsf{super}.m \Downarrow \textbf{err}}{\Xi \vdash e.m \Downarrow \textbf{err}}$$

$$\textsc{E-ErrPropArgMiss}\quad \frac{\Xi(\textsf{this}) = \{0 \star C[\overline{S}](\overline{v})\} \qquad m \notin \textsf{vparams}(C[\overline{S}]) \qquad (\textsf{this} \mapsto \{1 \star C[\overline{S}](\overline{v})\}) \vdash \textsf{super}.m \Downarrow \textbf{err}}{\Xi \vdash \textsf{super}.m \Downarrow \textbf{err}}$$

$$\textsc{E-ErrPropSuperMiss}\quad \frac{\Xi(\textsf{this}) = \{i \star C[\overline{S}](\overline{v})\} \qquad i \neq 0 \qquad m \notin \textsf{methods}(i, C[\overline{S}](\overline{v})) \qquad (\textsf{this} \mapsto \{(i+1) \star C[\overline{S}](\overline{v})\}) \vdash \textsf{super}.m \Downarrow \textbf{err}}{\Xi \vdash \textsf{super}.m \Downarrow \textbf{err}}$$

$$\textsc{E-ErrPropSuperHit}\quad \frac{\Xi(\textsf{this}) = \{i \star C[\overline{S}](\overline{v})\} \qquad i \neq 0 \qquad (m : T = e) \in \textsf{methods}(i, C[\overline{S}](\overline{v})) \qquad (\textsf{this} \mapsto \{(i+1) \star C[\overline{S}](\overline{v})\}) \vdash e \Downarrow \textbf{err}}{\Xi \vdash \textsf{super}.m \Downarrow \textbf{err}}$$

Figure 10 Big-step semantics producing errors.

$$\boxed{\Xi \vdash e \Downarrow_k r^+}$$

$$\textsc{E-KillPropApp1}\ \frac{\Xi \vdash e_1 \Downarrow_{k-1} \textbf{kill}}{\Xi \vdash e_1\ e_2 \Downarrow_k \textbf{kill}} \qquad \textsc{E-KillPropApp2}\ \frac{\Xi \vdash e_2 \Downarrow_{k-1} \textbf{kill}}{\Xi \vdash e_1\ e_2 \Downarrow_k \textbf{kill}} \qquad \textsc{E-KillPropTApp1}\ \frac{\Xi \vdash e \Downarrow_{k-1} \textbf{kill}}{\Xi \vdash e\ T \Downarrow_k \textbf{kill}}$$

$$\textsc{E-KillPropTApp2}\ \frac{\Xi \vdash e \Downarrow_{k-1} \textbf{val}\ \langle \Lambda X.\ e',\ \Xi' \rangle \qquad [T/X]\Xi' \vdash [T/X]e' \Downarrow_{k-1} \textbf{kill}}{\Xi \vdash e\ T \Downarrow_k \textbf{kill}}$$

$$\textsc{E-KillPropApp3}\ \frac{\Xi \vdash e_1 \Downarrow_{k-1} \textbf{val}\ \langle \lambda x : T.\ e,\ \Xi' \rangle \qquad \Xi \vdash e_2 \Downarrow_{k-1} \textbf{val}\ v \qquad \Xi',\ x \mapsto v \vdash e \Downarrow_{k-1} \textbf{kill}}{\Xi \vdash e_1\ e_2 \Downarrow_k \textbf{kill}}$$

$$\textsc{E-KillPropNew}\ \frac{\Xi \vdash e_i \Downarrow_{k-1} \textbf{kill}}{\Xi \vdash \textbf{new}\ C[\overline{T}](\overline{e_i}^{i \in 1..n}) \Downarrow_k \textbf{kill}} \qquad \textsc{E-KillPropAccess1}\ \frac{\Xi \vdash e \Downarrow_{k-1} \textbf{kill}}{\Xi \vdash e.m \Downarrow_k \textbf{kill}}$$

$$\textsc{E-KillPropAccess2}\ \frac{\Xi \vdash e \Downarrow_{k-1} \textbf{val}\ C[\overline{S}](\overline{v}) \qquad (\textbf{this} \mapsto \{0 \star C[\overline{S}](\overline{v})\}) \vdash \textbf{super}.m \Downarrow_{k-1} \textbf{kill}}{\Xi \vdash e.m \Downarrow_k \textbf{kill}}$$

$$\textsc{E-KillPropArgMiss}\ \frac{\Xi(\textbf{this}) = \{0 \star C[\overline{S}](\overline{v})\} \qquad m \notin \mathsf{vparams}(C[\overline{S}]) \qquad (\textbf{this} \mapsto \{1 \star C[\overline{S}](\overline{v})\}) \vdash \textbf{super}.m \Downarrow_{k-1} \textbf{kill}}{\Xi \vdash \textbf{super}.m \Downarrow_k \textbf{kill}}$$

$$\textsc{E-KillPropSuperMiss}\ \frac{\Xi(\textbf{this}) = \{i \star C[\overline{S}](\overline{v})\} \qquad i \neq 0 \qquad m \notin \mathsf{methods}(i, C[\overline{S}](\overline{v})) \qquad (\textbf{this} \mapsto \{(i+1) \star C[\overline{S}](\overline{v})\}) \vdash \textbf{super}.m \Downarrow_{k-1} \textbf{kill}}{\Xi \vdash \textbf{super}.m \Downarrow_k \textbf{kill}}$$

$$\textsc{E-KillPropSuperHit}\ \frac{\Xi(\textbf{this}) = \{i \star C[\overline{S}](\overline{v})\} \qquad i \neq 0 \qquad (m : T = e) \in \mathsf{methods}(i, C[\overline{S}](\overline{v})) \qquad (\textbf{this} \mapsto \{(i+1) \star C[\overline{S}](\overline{v})\}) \vdash e \Downarrow_{k-1} \textbf{kill}}{\Xi \vdash \textbf{super}.m \Downarrow_k \textbf{kill}}$$

Figure 11 Finite big-step semantics propagating timeout results.

▶ **Lemma 8.** *If $S\ \&\ T <: U$ then $S <: U$ or $T <: U$.*

Proof. By induction on the subtyping derivation.
Case S-Refl Immediate.
Case S-Top Immediate.
Case S-AndL/R Immediate.
Case S-Trans By IH and S-Trans.

◀

▶ **Lemma 9.** *If $S_1 \to T_1 <: S_2 \to T_2$ then $S_2 <: S_1$ and $T_1 <: T_2$.*

Proof. By induction on the subtyping derivation (IH1).
Case S-Refl Immediate.
Case S-Arrow Immediate.

Case S-Trans $S_1 \to T_1 <: U$ and $U <: S_2 \to T_2$.

By induction on U (IH2), we have the following cases to consider after we rule out impossible forms of U with Lemma 6:

- $U = S_3 \to T_3$. By IH1, $S_3 <: S_1$ and $T_1 <: T_3$ and $S_2 <: S_3$ and $T_3 <: T_2$. We conclude with S-Trans.
- $U = S_3 \,\&\, T_3$. By Lemma 7, $S_1 \to T_1 <: S_3$ and $S_1 \to T_1 <: T_3$. By Lemma 8, we have two cases to consider:
  - $S_3 <: S_2 \to T_2$. We conclude with IH2 and S-Trans.
  - $T_3 <: S_2 \to T_2$. We conclude with IH2 and S-Trans.

◀

▶ **Lemma 10.** *If* $\forall Y. S <: \forall X. T$ *then* $X = Y$ *and* $S <: T$*.*

Proof. By induction on the subtyping derivation (IH1).
Case S-Refl Immediate.
Case S-Forall Immediate.
Case S-Trans $\forall Y. S <: U$ and $U <: \forall X. T$.

By induction on U (IH2), we have the following cases to consider after we rule out impossible forms of U with Lemma 6:

- $U = \forall Z. V$. By IH1 and S-Trans.
- $U = S_3 \,\&\, T_3$. By Lemmas 7 and 8, IH2, and S-Trans.

◀

▶ **Lemma 11.** *If* $S <: T$ *then* $[U/X]S <: [U/X]T$*.*

Proof. By induction on the subtyping derivation. ◀

Method Type Calculation (mtype)

▶ **Lemma 12.** *For a list of types* $\overline{T}$*, for any* $T_i \in \overline{T}$*, if* $\mathsf{mtype}(m, T_i) = S$ *then* $\mathsf{mtype}(m, \&\overline{T}) = S'$ *and* $S' <: S$*.*

Proof. By induction on the list of types.

- The conclusion is vacuously true if the list is empty.
- If T_i is at the head of the list, by definition of mtype, $S' = \mathsf{mtype}(m, \&(T_i, \overline{T})) = S$ if $\mathsf{mtype}(m, \&\overline{T}) = \varnothing$ or $S' = S \,\&\, S''$ if $\mathsf{mtype}(m, \&\overline{T}) = S''$. In both cases, $S' <: S$. We conclude by IH if T_i is in the rest of the list.

◀

▶ **Lemma 13.** $\mathsf{mtype}(m, T) = \varnothing$ *or* $\mathsf{mtype}(m, T) = S$*.*

Proof. By induction on the type T. When $T = I[\overline{U}]$, we conclude if $m \in \mathcal{R}$. Otherwise, by induction on the tree of interface inheritance (note we assume non-cyclic inheritance), we know m's type can be calculated for all parent interfaces. By definition of mtype, the result is some type S if mtype of any super interface is not $\varnothing$, or $\varnothing$ otherwise. ◀

▶ **Lemma 14.** *Define* $\diamond ::= T \mid \varnothing$*. If* $\mathsf{mtype}(m, T) = \diamond_1$ *and* $\mathsf{mtype}(m, T) = \diamond_2$ *then* $\diamond_1 = \diamond_2$*.*

Proof. By induction on T. By Lemma 13 and IH, mtype is computable for every subtree of T. For an interface, as we assume methods are always uniquely declared, mtype of m in the interface is $\varnothing$ if undefined or a unique type, and the mtype result is therefore unique. ◀

▶ **Lemma 15.** *If $\overline{\mathcal{D}\ \textbf{ok}}$ and* $\mathsf{mtype}(m, T) = S$ *and* $T' <: T$ *then* $\mathsf{mtype}(m, T') = S'$ *and* $S' <: S$.

Proof. By induction on the subtyping derivation.
Case S-Refl Immediate.
Case S-Top Impossible as $\mathsf{mtype}(m, \mathsf{Object}) = \varnothing$.
Case S-Interface $I[\overline{U}] <: T$ and $T \in \mathsf{parents}(I[\overline{U}])$

If I is defined as $I[\overline{X}] \triangleleft \overline{J[\overline{U'}]}\{\ ...\ \}$, by definition of $\mathsf{parents}$, $T = [\overline{U/X}]J^*[\overline{U^*}]$ and $J^*[\overline{U^*}] \in \overline{J[\overline{U'}]}$, then $\mathsf{mtype}(m, [\overline{U/X}]J^*[\overline{U^*}]) = S$. By Lemma 12, $\mathsf{mtype}(m, \&\overline{[\overline{U/X}]J[\overline{U'}]}) = S'$ and $S' <: S$. Consider the following cases:

- $m \notin I$. By definition of mtypes, $\mathsf{mtype}(m, I[\overline{U}]) = S'$ and $S' <: S$.
- $m : S^* \in I$, then $\mathsf{mtype}(m, I[\overline{U}]) = [\overline{U/X}]S^*$.
  By $\overline{\mathcal{D}\ \textbf{ok}}$, $I\ \textbf{ok}$. By $I\ \textbf{ok}$, $\mathsf{mtype}(m, \&\overline{J[\overline{U'}]}) = S''$ and $S^* <: S''$. By Lemma 11, $\mathsf{mtype}(m, \&\overline{[\overline{U/X}]J[\overline{U'}]}) = [\overline{U/X}]S'' = S'$ and $[\overline{U/X}]S^* <: [\overline{U/X}]S'' = S'$. By S-Trans, $[\overline{U/X}]S^* <: S$.

Case S-Inv $I[\overline{U}] <: I[\overline{U'}]$ and $\overline{U <: U'}$ and $\overline{U' <: U}$

By Lemma 11, it is immediate to have equivalent results of mtype with equivalent type arguments for the input interface type.

Case S-AndI $S_1 \ \&\ S_2 <: S_1$ and $\mathsf{mtype}(m, S_1) = T$

By definition and Lemma 13, $\mathsf{mtype}(m, S_1 \ \&\ S_2) = T$ if $\mathsf{mtype}(m, S_2) = \varnothing$ or $\mathsf{mtype}(m, S_1 \ \&\ S_2) = T \ \&\ T'$ if $\mathsf{mtype}(m, S_2) = T'$. In both cases $\mathsf{mtype}(m, S_1 \ \&\ S_2) <: T$.

Case S-Andr Symmetric to the former case.
Case S-And $T <: T_1 \ \&\ T_2$ and $\mathsf{mtype}(m, T_1 \ \&\ T_2) = S$

By Lemma 13, we consider the following cases:

- $\mathsf{mtype}(m, T_1) = \mathsf{mtype}(m, T_2) = \varnothing$. Contradiction by Lemma 14.
- $\mathsf{mtype}(m, T_1) = S_1$ and $\mathsf{mtype}(m, T_2) = \varnothing$, then $\mathsf{mtype}(m, T_1 \ \&\ T_2) = S_1$. By IH, $\mathsf{mtype}(m, T) = S'$ and $S' <: S_1$.
- $\mathsf{mtype}(m, T_1) = \varnothing$ and $\mathsf{mtype}(m, T_2) = S_2$. Symmetric to the former case.
- $\mathsf{mtype}(m, T_1) = S_1$ and $\mathsf{mtype}(m, T_2) = S_2$, then $\mathsf{mtype}(m, T_1 \ \&\ T_2) = S_1 \ \&\ S_2$. By IH and Lemma 14 and S-And, $\mathsf{mtype}(m, T) = S'$ and $S' <: S_1 \ \&\ S_2$.

Case S-Trans $T' <: U$ and $U <: T$ and $\mathsf{mtype}(m, T) = S$

By IH, $\mathsf{mtype}(m, U) = S'' <: S$. By IH and Lemma 14, $\mathsf{mtype}(m, T') = S' <: S''$. By S-Trans, $\mathsf{mtype}(m, T') = S'$ and $S' <: S$.

◀

Determinism

▶ **Lemma 16** (Determinism). *If* $\Xi \vdash e \Downarrow r_1$ *and* $\Xi \vdash e \Downarrow r_2$ *then* $r_1 = r_2$.

Proof. By induction on the first and inversion on the second evaluation derivation. ◀

Inversion of Value Typing

▶ **Lemma 17.** *If* $\langle \lambda x : S.\ e, \Xi \rangle : T$ *then there exists* Γ, $\mathcal{R}$, *and* $\{i \star C[\overline{U}](\overline{v})\}$ *such that:*

1. $\Gamma \vDash \Xi$
2. $\Xi(\textbf{this}) = \{i \star C[\overline{U}](\overline{v})\}$
3. $\mathcal{R} \vDash \{i \star C[\overline{U}](\overline{v})\}$
4. $\Gamma, x : S, \textbf{super} : \mathcal{R} \vdash e : T'$
5. $S \to T' <: T$

or:

1. $\Gamma \vDash \Xi$
2. **this** $\notin dom(\Gamma)$
3. $\Gamma, x : S \vdash e : T'$
4. $S \to T' <: T$

Proof. By induction on the closure typing judgment. We conclude immediately for cases VT-ABS1 and VT-ABS2. For case VT-SUB, we conclude by IH and S-TRANS. ◀

▶ **Lemma 18.** *If* $\langle \Lambda X.\ e, \Xi \rangle : T$ *then there exists* Γ, $\mathcal{R}$, *and* $\{i \star C[\overline{S}](\overline{v})\}$ *such that:*

1. $\Gamma \vDash \Xi$
2. $\Xi(\textbf{this}) = \{i \star C[\overline{S}](\overline{v})\}$
3. $\mathcal{R} \vDash \{i \star C[\overline{S}](\overline{v})\}$
4. $\Gamma, \textbf{super} : \mathcal{R} \vdash e : T'$
5. $\forall X.\ T' <: T$

or:

1. $\Gamma \vDash \Xi$
2. **this** $\notin dom(\Gamma)$
3. $\Gamma \vdash e : T'$
4. $\forall X.\ T' <: T$

Proof. By induction on the closure typing judgment. We conclude immediately for cases VT-TABS1 and VT-TABS2. For case VT-SUB, we conclude by IH and S-TRANS. ◀

▶ **Lemma 19.** *If* $C[\overline{T}](\overline{v}) : V$ *then there exists* $\overline{m : U}$ *and* V' *such that:*

1. $\mathsf{vparams}(C[\overline{T}]) = \overline{m : U}$
2. $\overline{v : U}$
3. $\mathsf{ctype}(C[\overline{T}]) = V'$
4. $V' <: V$

Proof. By induction on the object typing judgment. We conclude immediately for the case VT-OBJECT. For case VT-SUB, we conclude by IH and S-TRANS. ◀

Type Substitution

▶ **Lemma 20** (Type substitution of typing)**.** *If* $\Gamma \vdash e : T$ *then* $[S/X]\Gamma \vdash [S/X]e : [S/X]T$.

Proof. By induction on the typing judgment. Note that we assume capture-avoiding type substitution. ◀

▶ **Lemma 21** (Type substitution of mtype)**.** *If* $\mathsf{mtype}(m, T) = S$ *then* $\mathsf{mtype}(m, [U/X]T) = [U/X]S$.

Proof. By induction on T. For the case where T is an interface, we prove by induction on the tree of interface inheritance. ◀

▶ **Lemma 22** (Type substitution of refinement consistency)**.** *If* $\mathcal{R} \vDash \{i \star C[\overline{T}](\overline{v})\}$ *then* $[S/X]\mathcal{R} \vDash \{i \star [S/X](C[\overline{T}](\overline{v}))\}$.

Proof. By inversion of the refinement consistency judgment and Lemma 11. ◀

▶ **Lemma 23** (Type substitution of value typing)**.** *If* $v : T$ *then* $[S/X]v : [S/X]T$.

▶ **Lemma 24** (Type substitution of context consistency)**.** *If* $\Gamma \vDash \Xi$ *then* $[T/X]\Gamma \vDash [T/X]\Xi$

Proof of Lemmas 23 and 24. We prove two lemmas above together.

Goal - Lemma 24 By induction on the consistency judgment and Lemma 23 in IH.

Goal - Lemma 23 By induction on the value typing judgment and Lemma 24 in IH and Lemma 20.

◀

Preservation Main Theorem

▶ **Lemma 25** (General preservation). *If* $\overline{\mathcal{D}\ \textit{ok}}$ *and* Γ, **super** $: \mathcal{R} \vdash e : T$ *and* $\Gamma \models \Xi$ *and* $\Xi(\textbf{this}) = \{i \star C[\overline{U}](\overline{v})\}$ *and* $\mathcal{R} \models \{i \star C[\overline{U}](\overline{v})\}$ *and* $\Xi \vdash e \Downarrow r$ *then* $r = \textbf{val}\ v'$ *and* $v' : T$.

▶ **Lemma 26** (Simple general preservation). *If* $\overline{\mathcal{D}\ \textit{ok}}$ *and* $\Gamma \vdash e : T$ *and* $\Gamma \models \Xi$ *and* **this** $\notin dom(\Gamma)$ *and* $\Xi \vdash e \Downarrow r$ *then* $r = \textbf{val}\ v'$ *and* $v' : T$.

Proof of Lemmas 25 and 26. We prove that both lemmas hold together.

Goal - Lemma 25 By induction on the typing derivation (IH1).

Case T-Var Γ, **super** $: \mathcal{R} \vdash x : T$

By the premise of typing, $\Gamma(x) = T$. We invert the evaluation derivation.

- $x \notin dom(\Xi)$ and $r = \textbf{err}$. As $\Gamma \models \Xi$, $x \in dom(\Xi)$. Contradiction.
- $\Xi(x) = v$ and $r = \textbf{val}\ v$. As $\Gamma \models \Xi$, $v : T$.

Case T-This Γ, **super** $: \mathcal{R} \vdash$ **this** $: T$

Analogous to the former case.

Case T-Abs Γ, **super** $: \mathcal{R} \vdash \lambda x : S.\ e : S \to T$

We invert the evaluation derivation. $r = \textbf{val}\ \langle \lambda x : S.\ e, \Xi\rangle$. We conclude with VT-Abs1.

Case T-TAbs Γ, **super** $: \mathcal{R} \vdash \Lambda X.\ e : \forall X.\ T$

Analogous to the former case.

Case T-App Γ, **super** $: \mathcal{R} \vdash e_1\ e_2 : T$

By induction on the evaluation derivation (IH2), there are several cases to consider:

- $\Xi \vdash e_1\ e_2 \Downarrow \textbf{err}$ and at least one of e_1 and e_2 evaluates to **err** under Ξ. By IH1, e_1 and e_2 should both be evaluated to values, which leads to a contradiction.
- $\Xi \vdash e_1\ e_2 \Downarrow \textbf{err}$ and $\Xi \vdash e_1 \Downarrow \textbf{val}\ D[\overline{U'}](\overline{w})$. By IH1, $D[\overline{U'}](\overline{w}) : S \to T$. By inversion lemma of object typing (Lemma 19), $\mathsf{ctype}(D[\overline{U'}]) = V$ and $V <: S \to T$. Syntactically, ctype only yields an interface type $I[\overline{U''}]$, and interfaces types are not subtypes of function types (Lemma 6). Contradiction.
- $\Xi \vdash e_1\ e_2 \Downarrow \textbf{err}$ and $\Xi \vdash e_1 \Downarrow \textbf{val}\ \langle \Lambda X.\ e, \Xi'\rangle$. By IH1, $\langle \Lambda X.\ e, \Xi'\rangle : S \to T$. By inversion lemma of type abstraction closure typing (Lemma 18), $\langle \Lambda X.\ e, \Xi'\rangle : \forall X.\ T'$ and $\forall X.\ T' <: S \to T$, which is impossible by Lemma 6. Contradiction.
- $\Xi \vdash e_1\ e_2 \Downarrow \textbf{val}\ v'$ and $\Xi \vdash e_1 \Downarrow \textbf{val}\ \langle \lambda x : S'.\ e, \Xi'\rangle$ and $\Xi \vdash e_2 \Downarrow \textbf{val}\ v$ and $\Xi', x \mapsto v \vdash e \Downarrow \textbf{val}\ v'$. By IH1, $\langle \lambda x : S'.\ e, \Xi'\rangle : S \to T$ and $v : S$. By the inversion lemma of typing of closures (Lemma 17), we have two cases to consider:
  - $\Gamma' \models \Xi'$ and $\Xi'(\textbf{this}) = \{i \star D[\overline{U'}](\overline{w})\}$ and $\mathcal{R}' \models \{i \star D[\overline{U'}](\overline{w})\}$, Γ', $x : S'$, **super** $: \mathcal{R}' \vdash e : T'$, and $S' \to T' <: S \to T$. By the inversion lemma of subtyping of function types (Lemma 9), $S <: S'$ and $T' <: T$. By VT-Sub, $v : S'$. By C-ConsVar, $\Gamma', x : S' \models \Xi', x \mapsto v$. By IH2 and VT-Sub, $v' : T$.
  - $\Gamma' \models \Xi'$ and **this** $\notin dom(\Gamma)$ and $\Gamma', x : S' \vdash e : T'$, and $S' \to T' <: S \to T$. By the inversion lemma of subtyping of function types (Lemma 9), $S <: S'$ and $T' <: T$. By VT-Sub, $v : S'$. By C-ConsVar, $\Gamma', x : S' \models \Xi', x \mapsto v$. By Lemma 26 in IH2 and VT-Sub, $v' : T$.

- $\Xi \vdash e_1\ e_2 \Downarrow$ **err** and $\Xi \vdash e_1 \Downarrow$ **val** $\langle \lambda x : S.\ e, \Xi' \rangle$ and $\Xi \vdash e_2 \Downarrow$ **val** v and $\Xi', x \mapsto v \vdash e \Downarrow$ **err**. By the same reasoning for the prior case, $\Xi', x \mapsto v \vdash e \Downarrow$ **val** v', which leads to a contradiction by Lemma 16.

Case T-TApp Γ, **super** : $\mathcal{R} \vdash e\ T : [T/X]S$

By induction on the evaluation derivation (IH2), there are several cases to consider:

- $\Xi \vdash e\ T \Downarrow$ **err** and e evaluates to **err** under Ξ. By IH1, e should be evaluated to a value, which leads to a contradiction.
- $\Xi \vdash e\ T \Downarrow$ **err** and $\Xi \vdash e \Downarrow$ **val** $D[\overline{U'}](\overline{w})$. By IH1, $D[\overline{U'}](\overline{w}) : \forall X.\, S$. By inversion lemma of object typing (Lemma 19), $\mathsf{ctype}(C[\overline{U}]) = V$ and $V <: \forall X.\, S$. Syntactically, ctype only yields an interface type $I[\overline{U'}]$, and interfaces types are not subtypes of universal types, which leads to a contradiction by Lemma 6.
- $\Xi \vdash e\ T \Downarrow$ **err** and $\Xi \vdash e \Downarrow$ **val** $\langle \lambda x : S'.\ e', \Xi' \rangle$. By IH1, $\langle \lambda x : S'.\ e', \Xi' \rangle : \forall X.\, S$. By inversion lemma of type abstraction closure typing (Lemma 18), $\langle \lambda x : S'.\ e', \Xi' \rangle : S' \to T'$ and $S' \to T' <: \forall X.\, S$, which is impossible by Lemma 6.
- $\Xi \vdash e\ T \Downarrow$ **val** v' and $\Xi \vdash e \Downarrow$ **val** $\langle \Lambda X.\ e', \Xi' \rangle$ and $[T/X]\Xi' \vdash [T/X]e' \Downarrow$ **val** v'. By IH1, $\langle \Lambda X.\ e', \Xi' \rangle : \forall X.\, S$. By the inversion lemma of typing of type abstraction closures (Lemma 18), we have two cases to consider:
  - $\Gamma' \vDash \Xi'$ and $\Xi'(\textbf{this}) = \{i \star D[\overline{U'}](\overline{w})\}$ and $\mathcal{R}' \vDash \{i \star D[\overline{U'}](\overline{w})\}$ and Γ, **super** : $\mathcal{R}' \vdash e' : S'$ and $\forall Y.\, S' <: \forall X.\, S$. By the inversion lemma of subtyping of universal types (Lemma 10), $X = Y$ and $S' <: S$. By Lemma 24, $[T/X]\Gamma' \vDash [T/X]\Xi'$. By Lemma 20, $[T/X]\Gamma'$, **super** : $[T/X]\mathcal{R}' \vdash [T/X]e' : [T/X]S'$. By Lemma 22, $[T/X]\mathcal{R}' \vDash \{i \star [T/X](D[\overline{U'}](\overline{w}))\}$. By IH2, $v' : [T/X]S'$. By Lemma 11, $[T/X]S' <: [T/X]S$. By VT-Sub, $v' : [T/X]S$.
  - $\Gamma \vDash \Xi'$ and **this** $\notin dom(\Gamma)$ and $\Gamma \vdash e : S'$, and $\forall Y.\, S' <: \forall X.\, S$. By the inversion lemma of subtyping of universal types (Lemma 10), $X = Y$ and $S' <: S$. By Lemma 24, $[T/X]\Gamma' \vDash [T/X]\Xi'$. By Lemma 20, $[T/X]\Gamma' \vdash [T/X]e' : [T/X]S'$. By Lemma 26 in IH2, $v' : [T/X]S'$. By Lemma 11, $[T/X]S' <: [T/X]S$. By VT-Sub, $v' : [T/X]S$.
- $\Xi \vdash e\ T \Downarrow$ **err** and $\Xi \vdash e \Downarrow$ **val** $\langle \Lambda X.\ e', \Xi' \rangle$ and $[T/X]\Xi' \vdash [T/X]e' \Downarrow$ **err**. Using the same reasoning for the prior case, $[T/X]\Xi' \vdash [T/X]e' \Downarrow$ **val** v', which leads to a contradiction by Lemma 16.

Case T-Access Γ, **super** : $\mathcal{R} \vdash e.m : S$

By induction on reduction derivation (IH2), there are several cases to consider:

- Γ, **super** : $\mathcal{R} \vdash e \Downarrow$ **err**. By premises and IH1, we know e will reduce to a value. Contradiction.
- Γ, **super** : $\mathcal{R} \vdash e \Downarrow$ **val** $\langle \lambda x : U.\ e', ... \rangle$. By IH1, we know $\langle \lambda x : U.\ e', ... \rangle : T$ and $\mathsf{mtype}(m, T) = S$. By Lemma 17, $\langle \lambda x : U.\ e', ... \rangle : S' \to T'$ and $S' \to T' <: T$. By Lemma 15, $\mathsf{mtype}(m, S' \to T') = V$, which is impossible.
- Γ, **super** : $\mathcal{R} \vdash e \Downarrow$ **val** $\langle \Lambda X.\ e', \Xi' \rangle$. By IH1, we know $\langle \Lambda X.\ e', \Xi' \rangle : T$ and $\mathsf{mtype}(m, T) = S$. By Lemma 18, $\langle \Lambda X.\ e', \Xi' \rangle : \forall X.\, S'$ and $\forall X.\, S' <: T$. By Lemma 15, $\mathsf{mtype}(m, \forall X.\, S') = V$, which is impossible.
- $e =$ **super**, then Γ, **super** : $\mathcal{R} \vdash$ **super** : T, which is impossible.
- $\Xi \vdash e \Downarrow$ **val** $C[\overline{U}](\overline{v})$ and $(\textbf{this} \mapsto \{0 \star C[\overline{U}](\overline{v})\}) \vdash \textbf{super}.m \Downarrow$ **val** v' and $C[\overline{U}](\overline{v}) : T$ and $\mathsf{mtype}(m, T) = S$. By IH1, $C[\overline{U}](\overline{v}) : T$. By Lemma 19, $\mathsf{ctype}(C[\overline{U}]) = T'$ and $T' <: T$. By Lemma 15, $\mathsf{mtype}(m, T') = S'$ and $S' <: S$. We lookup class C's definition as $C[\overline{X}] \triangleleft I[\overline{U'}], ...$ By definition of ctype, $T' = [\overline{U/X}]I[\overline{U'}]$, therefore $\mathsf{mtype}(m, [\overline{U/X}]I[\overline{U'}]) = S'$. By C ***ok***, $\mathsf{search}(m, 0, C) = S^*$ and $\mathsf{mtype}(m, I[\overline{U'}]) = S^{**}$ and $S^* <: S^{**}$. By Lemma 21, $S' = [\overline{U/X}]S^{**}$. By Lemma 11, $[\overline{U/X}]S^* <:$

$\overline{[U/X]}S^{**}$. By S-Trans, $\overline{[U/X]}S^* <: S$. We pick a structural refinement $\mathcal{R}^* = \{\ m : S\ \}$ and a typing context $\Gamma^* = (\textsf{this} : T)$. By T-Super, Γ^*, $\textsf{super} : \mathcal{R}^* \vdash \textsf{super}.m : S$. By definition of **super** refinement consistency, $\mathcal{R}^* \vDash \{0 \star C[\overline{U}](\overline{v})\}$. By MC-ConsThis, $\Gamma^* \vDash (\textsf{this} \mapsto \{0 \star C[\overline{U}](\overline{v})\})$. By IH2, $v' : S$.

- $\Xi \vdash e \Downarrow \textbf{val}\ C[\overline{U}](\overline{v})$ and $(\textsf{this} \mapsto \{0 \star C[\overline{U}](\overline{v})\}) \vdash \textsf{super}.m \Downarrow \textbf{err}$. Using the same reasoning for the prior case, $(\textsf{this} \mapsto \{0 \star C[\overline{U}](\overline{v})\}) \vdash \textsf{super}.m \Downarrow \textbf{val}\ v'$, which leads to a contradiction by Lemma 16.

Case T-Super Γ, $\textsf{super} : \mathcal{R} \vdash \textsf{super}.m : S$

$\textsf{mrefn}(m, \mathcal{R}) = S$. $\overline{X}$ is class C's type parameter list. By induction on the evaluation derivation (IH2), there are several cases to consider:

- **super** evaluates to a value under Ξ. These cases are impossible by ErrSuper and Lemma 16.
- $\textsf{this} \notin dom(\Xi)$ and $\Xi \vdash \textsf{super}.m \Downarrow \textbf{err}$. Contradiction.
- $\textsf{methods}(i, C[\overline{U}](\overline{v}))$ undefined and $i \neq 0$. As $\mathcal{R} \vDash \{i{\star}C[\overline{U}](\overline{v})\}$ and $\textsf{mrefn}(m, \mathcal{R}) = S$, there must be a method implementation of m in part of the mixin composition from index i to the end. Therefore, i cannot reach beyond the length of mixin composition and methods is defined. Contradiction.
- $\textsf{vparams}(C[\overline{U}])$ undefined and $i = 0$. As $\mathcal{R} \vDash \{0 \star C[\overline{U}](\overline{v})\}$ and $\textsf{mrefn}(m, \mathcal{R}) = S$, there must be a method implementation of m as an object field or part of the mixin composition. Therefore, C must be defined and $\textsf{vparams}(C[\overline{U}])$ is therefore defined. Contradiction.
- $m \notin \textsf{vparams}(C[\overline{U}])$ and $(\textsf{this} \mapsto \{1 \star C[\overline{U}](\overline{v})\}) \vdash \textsf{super}.m \Downarrow \textbf{val}\ v$ and $\mathcal{R} \vDash \{0 \star C[\overline{U}](\overline{v})\}$ and $\Gamma \vDash \Xi$. By $\Gamma \vDash \Xi$, $C[\overline{U}](\overline{v}) : V$. By inverting the **super** refinement consistency judgment, $\textsf{search}(m, 0, C) = S'$ and $\overline{[U/X]}S' <: S$ As $m \notin \textsf{vparams}(C[\overline{U}])$, $\textsf{search}(m, 1, C) = S'$. We pick a structural refinement $\mathcal{R}^* = \{\ m : S\ \}$ and a typing context $\Gamma^* = (\textsf{this} : V)$. By definition of **super** refinement consistency, $\mathcal{R}^* \vDash \{1 \star C[\overline{U}](\overline{v})\}$. By MC-ConsThis, $\Gamma^* \vDash (\textsf{this} \mapsto \{1 \star C[\overline{U}](\overline{v})\})$. By T-Super, Γ^*, $\textsf{super} : \mathcal{R}^* \vdash \textsf{super}.m : S$. By IH2, $v' : S$.
- $\textsf{vparams}(C[\overline{U}]) = \overline{m_i : S_i}$ and $m = m_i$ and $(\textsf{this} \mapsto \{0 \star C[\overline{U}](\overline{v_i})\}) \vdash \textsf{super}.m \Downarrow \textbf{val}\ v_i$. By assumption, $\mathcal{R} \vDash \{0{\star}C[\overline{U}](\overline{v})\}$. By premise of evaluation, $\textsf{vparams}(C[\overline{U}]) = \overline{m_i : S_i}$. By inverting the **super** refinement consistency judgment, $\textsf{search}(m, 0, C) = S'_i$ and $\overline{[U/X]}S'_i <: S$. As $\textsf{vparams}(C[\overline{U}]) = \overline{m_i : S_i}$, $\overline{[U/X]}S'_i = S_i$. As $\Gamma \vDash \Xi$, $C[\overline{U}](\overline{v}) : V$. By Lemma 19, $v_i : S_i$. By rule VT-Sub, $v_i : S$.
- Given $\mathcal{R} \vDash \{i \star C[\overline{U}](\overline{v})\}$ and $\Gamma \vDash \Xi$, $\textsf{search}(m, i, C) = S'$ and $\overline{[U/X]}S' <: S$. By premises, $m \notin \textsf{methods}(i, C[\overline{U}](\overline{v}))$, therefore $\textsf{search}(m, (i+1), C) = S'$. By $\Gamma \vDash \Xi$, $C[\overline{U}](\overline{v}) : V$. We pick a structural refinement $\mathcal{R}^* = \{\ m : S\ \}$ and a typing context $\Gamma^* = (\textsf{this} : V)$. By definition of **super** refinement consistency, $\mathcal{R}^* \vDash \{(i+1) \star C[\overline{U}](\overline{v})\}$. By MC-ConsThis, $\Gamma^* \vDash (\textsf{this} \mapsto \{(i+1) \star C[\overline{U}](\overline{v})\})$. By T-Super, Γ^*, $\textsf{super} : \mathcal{R}^* \vdash \textsf{super}.m : S$. By IH2, $v' : S$.
- $\mathcal{R} \vDash \{i{\star}C[\overline{U}](\overline{v})\}$ and $\Gamma \vDash \Xi$. Since $\Xi(\textsf{this}) = \{i{\star}C[\overline{U}](\overline{v})\}$, there exists some T such that $C[\overline{U}](\overline{v}) : T$. By premises of evaluation, we have the method implementation $(m : S' = e) \in \textsf{methods}(i, C[\overline{U}](\overline{v}))$ and $(\textsf{this} \mapsto \{(i+1) \star C[\overline{U}](\overline{v})\}) \vdash e \Downarrow \textbf{val}\ v'$. Therefore, we know that m is defined in mixin M_i. We lookup several definitions: mixin M_i as $M_i[\overline{Y}]^{\mathcal{R}'}_{T'}\mathcal{I}$, the class C as $C[\overline{X}] \lhd I[\overline{U'}], \overline{M_i[\overline{U''}]}$, and the original definition of m in $\mathcal{I}$ as $m : S'' = e'$. We denote the type substitution $\sigma = \overline{[U/X]}\overline{[U''/Y]}$. By $\overline{\mathcal{D}}$ ***ok***, C ***ok***. By C ***ok***, M_i ***ok***. By M_i ***ok***, $(\textsf{this} : T', \textsf{super} : \mathcal{R}') \vdash e' : S''$. By Lemma 20, $(\textsf{this} : \sigma T', \textsf{super} : \sigma\mathcal{R}') \vdash \sigma e' : \sigma S''$. By definition of methods, $S' = \sigma S''$ and $e = \sigma e'$. Therefore, $(\textsf{this} : \sigma T', \textsf{super} : \sigma\mathcal{R}') \vdash e : S'$.

To apply IH2, we are to prove (1) $(\textsf{this} : \sigma T') \vDash (\textsf{this} \mapsto \{(i+1) \star C[\overline{U}](\overline{v})\})$, (2) $\sigma\mathcal{R}' \vDash \{(i+1) \star C[\overline{U}](\overline{v})\}$. We now prove both aspects:

1. By C ***ok***, $M_i \Rightarrow C$. By $M_i \Rightarrow C$, $I[\overline{U'}] <: [\overline{U''/Y}]T'$. By Lemma 11, $[\overline{U/X}]I[\overline{U'}] <: \sigma T'$. By $C[\overline{U}](\overline{v}) : T$ and Lemma 19, $C[\overline{U}](\overline{v}) : T''$ and $T'' <: T$ and $\mathsf{ctype}(C[\overline{U}]) = T'' = [\overline{U/X}]I[\overline{U'}]$. By VT-Sub, $C[\overline{U}](\overline{v}) : \sigma T'$. By C-ConsThis, we have (1).
2. By C ***ok***, $M_i \Rightarrow C$. By $M_i \Rightarrow C$, for all $(m : S^*) \in \mathcal{R}'$, $\mathsf{search}(m, i+1, C) = S^{**}$ and $S^{**} <: [\overline{U''/Y}]S^*$. By Lemma 11, for all $(m : \sigma S^*) \in \sigma\mathcal{R}'$, $\mathsf{search}(m, i+1, C) = S^{**}$ and $[\overline{U/X}]S^{**} <: [\overline{U/X}][\overline{U''/Y}]S^* = \sigma S^*$. By definition of **super** refinement consistency, we have (2).

By IH2, $v' : S'$. By $\mathcal{R} \vDash \{i \star C[\overline{U}](\overline{v})\}$ and definition of search, $\mathsf{search}(m, i, C) = [\overline{U''/Y}]S''$ and $[\overline{U/X}][\overline{U''/Y}]S'' = \sigma S'' = S' <: S$. By VT-Sub, $v' : S$.

Case T-New $\Gamma, \textsf{super} : \mathcal{R} \vdash \textsf{new}\ C[\overline{T}](\overline{e}) : V$

By inverting the reduction derivation, there are several cases to consider:

- Some $e \in \overline{e}$ evaluates to **err** under Ξ. By IH and premises, we know all arguments e should evaluate to values. Contradiction.
- It is impossible that the number of arguments provided does not match the constructor given that the object instantiation is well-typed.
- $\Xi \vdash \textsf{new}\ C[\overline{T}](\overline{e}) \Downarrow \textsf{val}\ C[\overline{T}](\overline{v})$ and $\overline{\Xi \vdash e \Downarrow \textsf{val}\ v}$ and $\mathsf{vparams}(C[\overline{T}]) = \overline{m : U}$. By IH and premises, $\overline{v : U}$. By VT-Object, $C[\overline{T}](\overline{v}) : V$.

Case T-Sub By IH1 and VT-Sub.

Goal - Lemma 26 By induction on the typing derivation and the evaluation derivation. The proof is mostly analogous and symmetric to the proof of Lemma 25 by using the two lemmas in the induction hypothesis alternately. Note that for this proof, case T-Super is impossible as Γ contains no **super** structural refinement by $\Gamma \vDash \Xi$.

◀

Proof of Lemma 1. Corollary of Lemma 26. ◀

B.2 Coverage and Soundness

Proof of Lemma 3. By induction on n and case analysis on the shape of e. When $n = 0$, the theorem immediately follows by E-Timeout. We now prove the case when $n > 0$.

- $e = x$. E-Var handles the case when $\Xi(x) = v$. E-ErrVar handles the case when x is unbound in Ξ.
- $e = \textsf{this}$. E-This handles the case when $\Xi(\textsf{this}) = \{i \star C[\overline{T}](\overline{v})\}$. E-ErrThis handles the case when **this** is unbound in Ξ.
- $e = \textsf{super}$. By E-ErrSuper.
- $e = \lambda x : T.\ e'$. By E-Abs.
- $e = \Lambda X.\ e'$. By E-TAbs.
- $e = e_1\ e_2$. By IH, $\Xi \vdash e_1 \Downarrow_{n-1} r_1^+$ and $\Xi \vdash e_2 \Downarrow_{n-1} r_2^+$. We do case analysis on r_1^+ and r_2^+. The cases when at least one of r_1^+ and r_2^+ is **kill** or **err** are handled by E-ErrPropApp1, E-ErrPropApp2, E-KillPropApp1 and E-KillPropApp2. When $r_1^+ = \textbf{val}\ v_1$ and $r_2^+ = \textbf{val}\ v_2$, we do case analysis on v_1. The cases when v_1 is a type abstraction closure or an object are handled by E-ErrApp1 or E-ErrApp2. When v_1 is a lambda abstraction closure, we conclude by IH, E-App or E-ErrPropApp3 or E-KillPropApp3.
- $e = e'\ T$. By IH, $\Xi \vdash e' \Downarrow_{n-1} r^+$. We do case analysis on r^+. The cases when r^+ is **kill** or **err** are handled by E-ErrPropTApp1 and E-KillPropTApp1. When $r^+ = \textbf{val}\ v$, we do case analysis on v. The cases when v is a lambda abstraction closure or an object

are handled by E-ErrTApp1 or E-ErrTApp2. When v is a type abstraction closure, we conclude by IH, E-TApp or E-ErrPropTApp2 or E-KillPropTApp2.

- $e = e'.m$. By IH, $\Xi \vdash e' \Downarrow_{n-1} r^+$. We discuss cases when $e' \neq \textsf{super}$ or $e' = \textsf{super}$ and do case analysis on r^+.
  - $e' \neq \textsf{super}$. The cases when r^+ is **err** or **kill** are handled by E-ErrPropAccess1 and E-KillPropAccess1. When r^+ is **val** v, we do case analysis on v. The cases when v is a lambda or type abstraction closure are handled by E-ErrAccess1 or E-ErrAccess2. When v is an object, we conclude by IH, E-Access or E-ErrPropAccess2 or E-KillPropAccess2.
  - $e' = \textsf{super}$. The case when r^+ is **kill** is handled by E-KillPropAccess1. When r^+ is **err** or **val** v, we first discuss if **this** is bound in Ξ. If it is not, we conclude by E-ErrSuperAccess1. If it is $(\Xi(\textbf{this}) = \{i \star C[\overline{T}](\overline{v})\})$, we discuss $i = 0$ or $i > 0$.
    * $i = 0$. We discuss if $\mathsf{vparams}(C[\overline{T}])$ is defined or undefined, and m is in or not in $\mathsf{vparams}(C[\overline{T}])$. If $\mathsf{vparams}(C[\overline{T}])$ is undefined, we conclude by E-ErrSuperAccess3. Otherwise, if m is in $\mathsf{vparams}(C[\overline{T}])$, we conclude by E-ArgHit. If it is not, we conclude by IH, E-ArgMiss or E-ErrPropArgMiss or E-KillPropArgMiss.
    * $i > 0$. We discuss if $\mathsf{methods}(i, C[\overline{T}](\overline{v}))$ is defined or undefined, and m is in or not in $\mathsf{methods}(i, C[\overline{T}](\overline{v}))$. If it is undefined, we conclude by E-ErrSuperAccess2. Otherwise, if m is in $\mathsf{methods}(i, C[\overline{T}](\overline{v}))$, we conclude by IH, E-SuperHit or E-ErrPropSuperHit or E-KillPropSuperHit. If m is not in, we conclude by IH, E-SuperMiss or E-ErrPropSuperMiss or E-KillPropSuperMiss.
- $e = \textsf{new}\ C[\overline{T}](\overline{e_i}^{i \in 1..k})$. By IH, $\overline{\Xi \vdash e_i \Downarrow_{n-1} r_i^+}$. If $|\mathsf{vparams}(C[\overline{T}])| \neq k$, we conclude by E-ErrNew. Otherwise, we do case analysis on $\overline{r_i^+}$. If at least one of $\overline{r_i^+}$ is **err** or **kill**, we conclude by E-ErrPropNew or E-KillPropNew. Otherwise (i.e. $\overline{r_i^+ = \textbf{val}\ v_i}$), we conclude by E-New.

◀

▶ **Lemma 27.** *If* $\Xi \vdash e \Downarrow_n \textbf{err}$ *then* $\Xi \vdash e \Downarrow \textbf{err}$ *or* $\Xi \vdash e \Downarrow_n \textbf{kill}$.

Proof. By induction on the evaluation derivation. By IH, subderivation of $\epsilon \vdash e \Downarrow_n \textbf{err}$ with an **err** result has a **kill** result by finite evaluation or **err** result by the original evaluation. In the former case, we conclude by deriving $\epsilon \vdash e \Downarrow_n \textbf{kill}$ using timeout propagation rules. Otherwise, we derive $\Xi \vdash e \Downarrow \textbf{err}$. ◀

▶ **Lemma 28.** *If* $\Xi \vdash e \Downarrow_n \textbf{val}\ v$ *then* $\Xi \vdash e \Downarrow \textbf{val}\ v$.

Proof. By induction on the finite evaluation derivation. ◀

▶ **Lemma 29.** *If* $\mathcal{P} : T$ *then for all* n, $\epsilon \vdash e \Downarrow \textbf{val}\ v$ *and* $v : T$, *or* $\epsilon \vdash e \Downarrow_n \textbf{kill}$.

Proof of Lemma 29. By Lemma 3, $\epsilon \vdash e \Downarrow_n r^+$. If r^+ is **kill**, the lemma immediately follows. If r^+ is **err**, by Lemma 27, $\Xi \vdash e \Downarrow \textbf{err}$ or $\Xi \vdash e \Downarrow_n \textbf{kill}$. We conclude immediately in the latter case. In the former case, we reach a contradiction by Lemma 1. If r^+ is **val** v, we conclude by Lemmas 1 and 28. ◀

Proof of Theorem 5. Corollary of Lemma 29. ◀

C Examples from the Literature Implemented in MLscript

This appendix provides MLscript/SuperOOP implementations of the examples discussed in Section 4.3. The contents of these files were extracted directly from our test suite. All the lines that begin with `//|` were inserted by our testing infrastructure automatically, displaying the inferred types and evaluated results.

We have also made use of *modules* in our examples, but they are not a core concept of SuperOOP. Indeed, a module simply desugars to a parameterless class along with a **`let`** binding of the same name whose body is an instance of the class. More concretely, the declaration '`module M extends Ms implements Is`' desugars to:

```
class M() extends Ms implements Is
let M = M()
```

C.1 Polymorphic Variants

From Garrigue [20].

```
class Cons[out A](head: A, tail: Cons[A] | Nil)
module Nil
//| class Cons[A](head: A, tail: Cons[A] | Nil)
//| module Nil

let l = Cons(1, Nil)
//| let l: Cons[1]
//| l
//|   = Cons {}

class NotFound()
class Success[out A](result: A)
//| class NotFound()
//| class Success[A](result: A)

fun list_assoc(s, l) =
  if l is
    Cons(h, t) then
      if s === h._1 then Success(h._2)
      else list_assoc(s, t)
    Nil then NotFound()
//| fun list_assoc: ∀ 'a 'A. (Eql['a], Cons[{_1: 'a, _2: 'A}] | Nil,) → (
    NotFound | Success['A])

// fun list_assoc(s: Str, l: Cons[{ _1: Str, _2: 'b }] | Nil): NotFound |
    Success['b]

class Var(s: Str)
//| class Var(s: Str)

mixin EvalVar {
  fun eval(sub, v) =
    if v is Var(s) then
      if list_assoc(s, sub) is
        NotFound    then v
        Success(r) then r
```

```
}
//| mixin EvalVar() {
//|   fun eval: (Cons[{_1: anything, _2: 'result}] | Nil, Var,) → (Var | '
    result)
//| }

class Abs[A](x: Str, t: A)
class App[A](s: A, t: A)
//| class Abs[A](x: Str, t: A)
//| class App[A](s: A, t: A)

fun gensym(): Str = "fun"
//| fun gensym: () → Str

fun int_to_string(x: Int): Str = "0"
//| fun int_to_string: (x: Int,) → Str

mixin EvalLambda {
  fun eval(sub, v) =
    if v is
      App(t1, t2) then
        let l1 = this.eval(sub, t1)
        let l2 = this.eval(sub, t2)
        if t1 is Abs(x, t) then
          this.eval(Cons((x, l2), Nil), t)
        else
          App(l1, l2)
      Abs(x, t) then
        let s = gensym()
        Abs(s, this.eval(Cons((x, Var(s)), sub), t))
    else
      super.eval(sub, v)
}
//| mixin EvalLambda() {
//|   super: {eval: ('a, 'b,) → 'c}
//|   this: {eval: ('a, 's,) → ('A & 'd) & (Cons[(Str, 'd,)], 't,) → 'c &
     (Cons[(Str, Var,) | 'A0], 't0,) → 'A1}
//|   fun eval: ('a & (Cons['A0] | Nil), Abs['t0] | App['s & (Abs['t] |
    Object & ~#Abs)] | Object & 'b & ~#Abs & ~#App,) → (Abs['A1] | App['A]
     | 'c)
//| }

module Test1 extends EvalVar, EvalLambda
//| module Test1 {
//|   fun eval: ∀ 'a. (Cons[{_1: anything, _2: 'result}] | Nil, Abs['b] |
    App['A] | Var,) → ('result | 'a)
//| }
//| where
//|   'b <: Abs['b] | App['A] | Var
//|   'A <: 'b & (Abs['b] | Object & ~#Abs)
//|   'result :> Var | 'a
//|   'a :> App['result] | Abs['result]

Test1.eval(Nil, Var("a"))
//| ∀ 'a. 'A | 'a
```

```
//|   where
//|     'A :> 'a | Var
//|     'a :> App['A] | Abs['A]
//| res
//|     = Var {}

Test1.eval(Nil, Abs("b", Var("a")))
//| ∀ 'a. 'A | 'a
//|   where
//|     'A :> Var | 'a
//|     'a :> App['A] | Abs['A]
//| res
//|     = Abs {}

Test1.eval(Cons(("c", Var("d")), Nil), App(Abs("b", Var("b")), Var("c")))
//| ∀ 'a. 'A | 'a
//|   where
//|     'A :> 'a | Var
//|     'a :> App['A] | Abs['A]
//| res
//|     = Var {}

Test1.eval(Cons(("c", Abs("d", Var("d"))), Nil), App(Abs("b", Var("b")),
    Var("c")))
//| ∀ 'a. 'A | 'a
//|   where
//|     'A :> 'a | Abs[Var] | Var
//|     'a :> App['A] | Abs['A]
//| res
//|     = Var {}

class Numb(n: Int)
class Add[A](l: A, r: A)
class Mul[A](l: A, r: A)
//| class Numb(n: Int)
//| class Add[A](l: A, r: A)
//| class Mul[A](l: A, r: A)

fun map_expr(f, v) =
  if v is
    Var        then v
    Numb       then v
    Add(l, r) then Add(f(l), f(r))
    Mul(l, r) then Mul(f(l), f(r))
//| fun map_expr: ∀ 'l 'A 'l0 'A0. ('l → 'A & 'l0 → 'A0, Add['l] | Mul['
    l0] | Numb | Var,) → (Add['A] | Mul['A0] | Numb | Var)

mixin EvalExpr {
  fun eval(sub, v) =
    fun eta(e) = this.eval(sub, e)
    let vv = map_expr(eta, v)
    if vv is
      Var                      then super.eval(sub, vv)
      Add(Numb(l), Numb(r)) then Numb(l + r)
      Mul(Numb(l), Numb(r)) then Numb(l * r)
```

```
    else v
}
//| mixin EvalExpr() {
//|   super: {eval: ('a, Var,) → 'b}
//|   this: {eval: ('a, 'c,) → Object}
//|   fun eval: ('a, 'b & (Add['c] | Mul['c] | Numb | Var),) → (Numb | 'b)
//| }

module Test2 extends EvalVar, EvalExpr
//| module Test2 {
//|   fun eval: ∀ 'a. (Cons[{_1: anything, _2: Object & 'result}] | Nil, 'a
     & (Add['b] | Mul['b] | Numb | Var),) → (Numb | Var | 'result | 'a | '
    b)
//| }
//| where
//|   'b <: Add['b] | Mul['b] | Numb | Var

Test2.eval(Nil, Var("a"))
//| Numb | Var
//| res
//|     = Var {}

Test2.eval(Cons(("c", Abs("d", Var("d"))), Nil), Var("a"))
//| Abs[Var] | Numb | Var
//| res
//|     = Var {}

Test2.eval(Cons(("a", Numb(1)), Nil), Var("a"))
//| Numb | Var
//| res
//|     = Var {}

Test2.eval(Cons(("a", Abs("d", Var("d"))), Nil), Add(Numb(1), Var("a")))
//| Abs[Var] | Add[Numb | Var] | Numb | Var
//| res
//|     = Add {}

module Test3 extends EvalVar, EvalExpr, EvalLambda
//| module Test3 {
//|   fun eval: ∀ 'A 'a. (Cons[{_1: anything, _2: 'result}] | Nil, Abs['b]
    | App['A] | Object & 'c & ~#Abs & ~#App,) → ('A0 | 'a)
//| }
//| where
//|   'result :> 'A0
//|           <: Object
//|   'A0 :> Numb | Var | 'result | 'c | 'a
//|   'a :> App['A0] | Abs['A0]
//|   'c <: Add['b] | Mul['b] | Numb | Var
//|   'b <: Abs['b] | App['A] | Object & 'c & ~#Abs & ~#App
//|   'A <: 'b & (Abs['b] | Object & ~#Abs)

Test3.eval(Cons(("c", Abs("d", Var("d"))), Nil), Abs("a", Var("a")))
//| ∀ 'a. 'A | 'a
//|   where
//|     'A :> 'a | Abs[Var] | Numb | Var
```

```
//|      'a :> App['A] | Abs['A]
//| res
//|      = Abs {}

Test3.eval(Cons(("c", Abs("d", Var("d"))), Nil), App(Abs("a", Var("a")),
    Add(Numb(1), Var("c"))))
//| ∀ 'a. 'A | 'a
//|    where
//|      'A :> 'a | Abs[Var] | Add[Numb | Var] | Numb | Var
//|      'a :> App['A] | Abs['A]
//| res
//|      = Var {}
```

C.2 A Simple "Regions" DSL

From Sun et al. [37]. Note that for better illustration of class/module method types, we provide several type synonyms to represent the variant types of the eDSL. The inferred type signatures are checked as subtypes of the method type annotations. For the unannotated raw version of this example, please refer to our open-source implementation repository or our artifact.

```
// ******************* Initial System *******************

class Vector(val x: Int, val y: Int)
//| class Vector(x: Int, y: Int)

class Circle(radius: Int)
class Outside[out Region](a: Region)
class Union[out Region](a: Region, b: Region)
class Intersect[out Region](a: Region, b: Region)
class Translate[out Region](v: Vector, a: Region)
//| class Circle(radius: Int)
//| class Outside[Region](a: Region)
//| class Union[Region](a: Region, b: Region)
//| class Intersect[Region](a: Region, b: Region)
//| class Translate[Region](v: Vector, a: Region)

type BaseLang[T] = Circle | Intersect[T] | Union[T] | Outside[T] |
    Translate[T]
//| type BaseLang[T] = Circle | Intersect[T] | Outside[T] | Translate[T] |
    Union[T]

mixin SizeBase {
  fun size(r) =
    if r is
      Circle(_)        then 1
      Outside(a)       then this.size(a) + 1
      Union(a, b)      then this.size(a) + this.size(b) + 1
      Intersect(a, b) then this.size(a) + this.size(b) + 1
      Translate(_, a) then this.size(a) + 1
}
//| mixin SizeBase() {
//|    this: {size: 'a → Int}
//|    fun size: (Circle | Intersect['a] | Outside['a] | Translate['a] |
```

```
    Union['a]) → Int
//| }

// ******************* Linguistic Reuse and Meta-Language Optimizations
    *******************

fun round(n: Num): Int = 0
//| fun round: (n: Num) → Int

fun go(x, offset) =
  if x is 0 then Circle(1)
  else
    let shared = go(x - 1, round(offset / 2))
    Union(Translate(Vector(0 - offset, 0), shared),
          Translate(Vector(offset, 0), shared))
//| fun go: ∀ 'Region. (0 | Int & ~0, Int) → 'Region
//|   where
//|     'Region :> Circle | Union[Translate['Region]]

// * Note that first-class polymorphism manages (correctly) to preserve the
     universal quantification
let circles = go(2, 1024)
//| let circles: ∀ 'Region. 'Region
//|   where
//|     'Region :> Circle | Union[Translate['Region]]
//| circles
//|         = Union {}

// ******************* Adding More Language Constructs *******************

class Univ()
class Empty()
class Scale[out Region](v: Vector, a: Region)
//| class Univ()
//| class Empty()
//| class Scale[Region](v: Vector, a: Region)

type ExtLang[T] = Univ | Empty | Scale[T]
//| type ExtLang[T] = Empty | Scale[T] | Univ

mixin SizeExt {
  fun size(a) =
    if a is
      Univ        then 1
      Empty       then 1
      Scale(_, b) then this.size(b) + 1
    else super.size(a)
}
//| mixin SizeExt() {
//|   super: {size: 'a → 'b}
//|   this: {size: 'c → Int}
//|   fun size: (Empty | Object & 'a & ~#Empty & ~#Scale & ~#Univ | Scale['
    c] | Univ) → (Int | 'b)
//| }
```

```
type RegionLang = BaseLang[RegionLang] | ExtLang[RegionLang]
//| type RegionLang = BaseLang[RegionLang] | ExtLang[RegionLang]

module TestSize extends SizeBase, SizeExt {
  fun size: RegionLang → Int
}
//| module TestSize {
//|   fun size: RegionLang → Int
//| }

TestSize.size(Empty())
//| Int
//| res
//|     = 1

TestSize.size(circles)
//| Int
//| res
//|     = 13

TestSize.size(Scale(Vector(1, 1), circles))
//| Int
//| res
//|     = 14

// ****************** Adding a New Interpretation *******************
// a stupid power (Int ** Int) implementation
fun pow(x, a) =
  if a is 0 then 1
  else x * pow(x, a - 1)
//| fun pow: (Int, 0 | Int & ~0) → Int

mixin Contains {
  fun contains(a, p) =
    if a is
      Circle(r)           then pow(p.x, 2) + pow(p.y, 2) <= pow(r, 2)
      Outside(a)          then not (this.contains(a, p))
      Union(lhs, rhs)     then
        this.contains(lhs, p) or this.contains(rhs, p)
      Intersect(lhs, rhs) then
        this.contains(lhs, p) and this.contains(rhs, p)
      Translate(v, a)     then
        this.contains(a, Vector(p.x - v.x, p.y - v.y))
}
//| mixin Contains() {
//|   this: {contains: ('a, 'b) → Bool & ('c, Vector) → 'd}
//|   fun contains: (Circle | Intersect['a] | Outside['a] | Translate['c] |
     Union['a], {x: Int, y: Int} & 'b) → (Bool | 'd)
//| }

type BaseRegionLang = BaseLang[BaseRegionLang]
//| type BaseRegionLang = BaseLang[BaseRegionLang]

module TestContains extends Contains {
  fun contains: (BaseRegionLang, Vector) → Bool
```

```
}
//| module TestContains {
//|   fun contains: (BaseRegionLang, Vector) → Bool
//| }

TestContains.contains(Translate(Vector(0, 0), Circle(1)), Vector(0, 0))
//| Bool
//| res
//|     = true

TestContains.contains(Intersect(Translate(Vector(0, 0), Circle(1)),
                                Circle(1)), Vector(0, 0))
//| Bool
//| res
//|     = true

TestContains.contains(circles, Vector(0, 0))
//| Bool
//| res
//|     = false

// ******************* Dependencies, Complex Interpretations, and Domain-
    Specific Optimizations *******************

fun toString(a: Int): Str = "foo"
fun concat(a: Str, b: Str): Str = a
//| fun toString: (a: Int) → Str
//| fun concat: (a: Str, b: Str) → Str

mixin Text {
  fun text(e) =
    if e is
      Circle(r)  then
        concat("a circular region of radius ", toString(r))
      Outside(a) then
        concat("outside a region of size ", toString(this.size(a)))
      Union      then
        concat("the union of two regions of size ", toString(this.size(e)))
      Intersect  then
        concat("the intersection of two regions of size ",
          toString(this.size(e)))
      Translate  then
        concat("a translated region of size ", toString(this.size(e)))
}
//| mixin Text() {
//|   this: {size: (Intersect[nothing] | Translate['Region] | Union[nothing
    ] | 'a) → Int}
//|   fun text: (Circle | Intersect[anything] | Outside['a] | Translate['
    Region] | Union[anything]) → Str
//| }


module SizeText extends SizeBase, Text {
  fun size: BaseRegionLang → Int
  fun text: BaseRegionLang → Str
```

```
}
//| module SizeText {
//|   fun size: BaseRegionLang -> Int
//|   fun text: BaseRegionLang -> Str
//| }

SizeText.text(circles)
//| Str
//| res
//|     = 'the union of two regions of size '

SizeText.size(circles)
//| Int
//| res
//|     = 13

SizeText.text(Intersect(Translate(Vector(0, 0), Circle(1)), Circle(1)))
//| Str
//| res
//|     = 'the intersection of two regions of size '

SizeText.size(Intersect(Translate(Vector(0, 0), Circle(1)), Circle(1)))
//| Int
//| res
//|     = 4

mixin IsUniv {
  fun isUniv(e) =
    if e is
      Univ            then true
      Outside(a)      then this.isEmpty(a)
      Union(a, b)     then this.isUniv(a) or this.isUniv(b)
      Intersect(a, b) then this.isUniv(a) and this.isUniv(b)
      Translate(_, a) then this.isUniv(a)
      Scale(_, a)     then this.isUniv(a)
    else false
}
//| mixin IsUniv() {
//|   this: {isEmpty: 'a -> 'b, isUniv: 'c -> Bool & 'd -> 'b}
//|   fun isUniv: (Intersect['c] | Object & ~#Intersect & ~#Outside & ~#
    Scale & ~#Translate & ~#Union & ~#Univ | Outside['a] | Scale['d] |
    Translate['d] | Union['c] | Univ) -> (Bool | 'b)
//| }

mixin IsEmpty {
  fun isEmpty(e) =
    if e is
      Univ            then true
      Outside(a)      then this.isUniv(a)
      Union(a, b)     then this.isEmpty(a) or this.isEmpty(b)
      Intersect(a, b) then this.isEmpty(a) and this.isEmpty(b)
      Translate(_, a) then this.isEmpty(a)
      Scale(_, a)     then this.isEmpty(a)
    else false
}
```

```
//| mixin IsEmpty() {
//|   this: {isEmpty: 'a → Bool & 'b → 'c, isUniv: 'd → 'c}
//|   fun isEmpty: (Intersect['a] | Object & ~#Intersect & ~#Outside & ~#
    Scale & ~#Translate & ~#Union & ~#Univ | Outside['d] | Scale['b] |
    Translate['b] | Union['a] | Univ) → (Bool | 'c)
//| }

module IsUnivIsEmpty extends IsUniv, IsEmpty {
  fun isEmpty: RegionLang → Bool
  fun isUniv:  RegionLang → Bool
}
//| module IsUnivIsEmpty {
//|   fun isEmpty: RegionLang → Bool
//|   fun isUniv: RegionLang → Bool
//| }

module IsUnivIsEmpty extends IsEmpty, IsUniv {
  fun isEmpty: RegionLang → Bool
  fun isUniv:  RegionLang → Bool
}
//| module IsUnivIsEmpty {
//|   fun isEmpty: RegionLang → Bool
//|   fun isUniv: RegionLang → Bool
//| }

IsUnivIsEmpty.isUniv(circles)
//| Bool
//| res
//|     = false

IsUnivIsEmpty.isEmpty(circles)
//| Bool
//| res
//|     = false



mixin Eliminate {
  fun eliminate(e) =
    if e is
      Outside(Outside(a)) then this.eliminate(a)
      Outside(a)          then Outside(this.eliminate(a))
      Union(a, b)         then
        Union(this.eliminate(a), this.eliminate(b))
      Intersect(a, b)     then
        Intersect(this.eliminate(a), this.eliminate(b))
      Translate(v, a)     then Translate(v, this.eliminate(a))
      Scale(v, a)         then Scale(v, this.eliminate(a))
    else e
}
//| mixin Eliminate() {
//|   this: {
//|     eliminate: 'a → 'b & 'c → 'Region & 'd → 'Region0 & 'e → '
    Region1 & 'f → 'Region2 & 'g → 'Region3
//|   }
```

```
//|   fun eliminate: (Intersect['e] | Object & 'b & ~#Intersect & ~#Outside
     & ~#Scale & ~#Translate & ~#Union | Outside['c & (Object & ~#Outside |
     Outside['a])] | Scale['g] | Translate['f] | Union['d]) -> (Intersect['
    Region1] | Outside['Region] | Scale['Region3] | Translate['Region2] |
    Union['Region0] | 'b)
//| }

module TestElim extends Eliminate {
  fun eliminate: RegionLang -> RegionLang
}
//| module TestElim {
//|   fun eliminate: RegionLang -> RegionLang
//| }

TestElim.eliminate(Outside(Outside(Univ())))
//| RegionLang
//| res
//|     = Univ {}

TestElim.eliminate(circles)
//| RegionLang
//| res
//|     = Union {}


// ***********************************************************

module Lang extends SizeBase, SizeExt, Contains, Text, IsUniv, IsEmpty,
    Eliminate {
  fun contains:  (BaseRegionLang, Vector) -> Bool
  fun eliminate: RegionLang -> RegionLang
  fun isEmpty:   RegionLang -> Bool
  fun isUniv:    RegionLang -> Bool
  fun size:      RegionLang -> Int
  fun text:      BaseRegionLang -> Str
}
//| module Lang {
//|   fun contains: (BaseRegionLang, Vector) -> Bool
//|   fun eliminate: RegionLang -> RegionLang
//|   fun isEmpty: RegionLang -> Bool
//|   fun isUniv: RegionLang -> Bool
//|   fun size: RegionLang -> Int
//|   fun text: BaseRegionLang -> Str
//| }

Lang.size(circles)
//| Int
//| res
//|     = 13

Lang.contains(circles, Vector(0, 0))
//| Bool
//| res
//|     = false
```

```
Lang.text(circles)
//| Str
//| res
//|     = 'the union of two regions of size '

Lang.isUniv(circles)
//| Bool
//| res
//|     = false

Lang.isEmpty(circles)
//| Bool
//| res
//|     = false

Lang.size(Lang.eliminate(circles))
//| Int
//| res
//|     = 13
```