Modular Type Safety for Traits with Extensible Variants and Deep Pattern Matching

ANDONG FAN✉, University of Toronto, Canada
LIONEL PARREAUX, HKUST, Hong Kong, China
NINGNING XIE, University of Toronto, Canada

Traits provide a powerful mechanism for code reuse, as they allow the definition of shared behaviors that can be composed into classes. Scala traits in particular have been used extensively in both academia and industry to help define reusable components, especially in the context of domain-specific language (DSL) compilers. Pattern matching on the extensible data types representing a DSL's constructs plays a key role in these applications. However, guaranteeing static type safety in this context is challenging: in Scala, a program using traits may successfully type check but then throw a runtime exception due to non-exhaustive pattern matching. This paper proposes a novel trait language which, for the first time, combines several important features: extensible data types, deep pattern matching, method overriding, exhaustiveness guarantees, and separate type checking. The former three are crucial to supporting DSL analysis and optimization use cases, while the latter two are important for reliable and scalable software development in the large. We formalize our approach in the framework of Boolean-algebraic subtyping, but its core ideas could be adapted to other type systems; thanks to it, languages like Scala that feature traits and extensible variants can finally become type safe, improving the experience of developers working with DSL compilation and related use cases.

CCS Concepts: • **Theory of computation → Type structures**; • **Software and its engineering → Object oriented languages**.

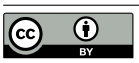
Additional Key Words and Phrases: traits, extensible variants, pattern matching

ACM Reference Format:
Andong Fan, Lionel Parreaux, and Ningning Xie. 2026. Modular Type Safety for Traits with Extensible Variants and Deep Pattern Matching. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 382 (October 2026), 28 pages. https://doi.org/10.1145/3839514

1 Introduction

Traits [39] offer a way of defining shared behaviors that can be composed into classes, enabling powerful patterns of code reuse. Among language designs inspired by traits [1, 3, 13, 40], one of the most successful and widely used is that of the Scala programming language [24, 25, 29]. Scala traits support the composition of reusable components via multiple inheritance, also referred to as *mixin composition*, a capability that has been widely adopted in deeply embedded domain-specific language (DSL) implementations [16, 21, 26, 38]. In each trait, one can define and *extend* selected syntax forms and operations of the DSL, and a complete DSL can then be built by composing a selection of such traits into a single class. This pattern is well known in the Scala community and has been used pervasively in, for example, the LMS [37, 41] and Squid [32, 35] frameworks. The essence of the pattern can be illustrated in the following example, written in Scala syntax:

Authors' Contact Information: Andong Fan, University of Toronto, Toronto, Canada, andong.fan@mail.utoronto.ca; Lionel Parreaux, HKUST, Hong Kong, China, parreaux@cse.ust.hk; Ningning Xie, University of Toronto, Toronto, Canada, ningningxie@cs.toronto.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/10-ART382
https://doi.org/10.1145/3839514

```
trait Base:
  abstract class Exp
  case class Lit(n: Int) extends Exp
  case class Add(l: Exp, r: Exp) extends Exp
```

```
trait Eval extends Base:
  def eval(e: Exp): Int = e match
    case Lit(n) => n
    case Add(l,r) => eval(l) + eval(r)
```

To define new operations on arithmetic expressions, such as pretty-printing, one can create a new trait, say `Print`, that extends `Base` with the new pretty-printing method, and then compose everything together through multiple inheritance:

```
object Arith extends Base, Eval, Print
val e = Arith.Add(Arith.Lit(1), Arith.Lit(2))
Arith.eval(e) // returns 3
Arith.print(e) // returns "1 + 2"
```

We defined `Exp` and its operations in traits to make them *modularly extensible*. Direct updates to the `Exp` hierarchy as defined in the `Base` trait would require recompilation of all downstream traits and modules, compromising code reuse and modularity. Instead, thanks to the *open* nature of abstract classes like `Exp` in Scala[1], we can simply define another trait with new subclasses:

```
trait EvalExt extends Eval:
  case class Neg(e: Exp) extends Exp
  override def eval(e: Exp): Int = e match
    case Neg(e) => 0 - eval(e)
    case _ => super.eval(e)
```

This new trait extends `Eval` and overrides the `eval` method to handle the new `Neg` variant of `Exp` and delegate the handling of other variants to the overridden method. We can now construct an extended language that incorporates the new trait, and call the two existing methods on an instance of the new variant:

```
object BadArith1 extends Base, Eval, Print, EvalExt
BadArith1.eval(BadArith1.Neg(e)) // -3
BadArith1.print(BadArith1.Neg(e)) // ???
```

While the evaluator works as expected, the pretty-printer triggers a runtime error on some inputs: indeed, we forgot to provide an extension that handles the pretty-printing of the new `Neg` case. The situation is even more insidious when variants are introduced in independently developed traits, with the downstream traits unaware of these newly mixed-in variants:

```
trait Foo extends Base:
  case class Unknown() extends Exp
object BadArith2 extends Base, Foo, Eval
BadArith2.eval(BadArith2.Unknown()) // ???
```

The type checker should reject both `BadArith1` and `BadArith2`; otherwise, one would have to *always remember* to extend all places that involve `Exp`, adding support for new cases that could be introduced *anywhere* in the hierarchy. Unfortunately, Scala has no way of statically tracking this; it throws a `scala.MatchError` at runtime instead[2]. Without properly enforcing the exhaustiveness of pattern matching on extensible variants, this design pattern is fundamentally error-prone, despite its widespread adoption and successful applications in the wild. The challenge is reminiscent of the classic Expression Problem [44], which highlights the difficulties of extending both data types and operations in a modular way while maintaining static type safety and separate compilation. While there has been much work on extensible variants with exhaustiveness checking [4, 15, 27, 36, 45] and

[1]Abstract classes extended by case classes are similar to open/extensible variants in, e.g., OCaml [23].
[2]One could add a "sealed" modifier to the base class to enable exhaustiveness checks, but that would prohibit extensions of the base class outside of the current source file, defeating all modularity and reuse potential.

there exist modular programming approaches that support traits with extensible variants [22, 47], to the best of our knowledge, no system simultaneously supports all of: *separate type checking*, *method overriding*, straightforward *deep pattern matching* (i.e. pattern matching with patterns containing nested constructors, a fundamental feature required for modular DSL optimization passes), and principled *exhaustive pattern matching guarantees*.

In this paper, we present a novel approach to modular programming with traits and extensible variants in the framework of *Boolean-algebraic subtyping* [5, 14, 34]. When a variant type `A` is defined in a trait `T`, it is open to retroactive extensions by downstream traits that inherit `T`. In trait method signatures, `A` may be referenced in two ways: `now/A`, which refers to all cases of `A` defined *up until **now***, according to the local knowledge of the current trait; and `this/A`, which refers to the sum of all cases that will eventually be defined when the trait is finally mixed into a class; these may include additional variants that are introduced later, possibly in independently developed traits. When a class is composed from separate traits, the type system makes sure that at the end of the day, in all inherited signatures, `now/A` matches `this/A`, and that the assumptions made in each trait method about the overridden implementation are satisfied. These two aspects together ensure type safety, a property that we justify through the use of *precise typing of open recursion* [12] in our underlying core language. Thanks to Boolean-algebraic subtyping, our approach supports modular type checking of partial pattern matches and their compositions. In particular, our type system guarantees the absence of runtime match errors, no matter how deeply nested the patterns are.

All in all, our contributions are the following:

- We present a new language design that generalizes Scala's powerful notion of traits with extensible variants by allowing principled type checking with exhaustiveness guarantees while retaining full modularity (Section 2). Our design improves the state of the art in modular and extensible programming, particularly for DSL development.
- We formalize this trait system design in a *surface language*, drawing from our new perspective on extensible variants and from the formalism of Scala traits [24, 25] adapted to the framework of Boolean-algebraic subtyping [34] (Section 3).
- We show the elaboration of our trait language into a *core language* that extends an existing lower-level calculus of classes, interfaces, and mixins [12] with Boolean-algebraic constructs and class-based pattern matching (Section 4), giving a runtime semantics to the surface language.
- We prove the soundness of the surface trait language by showing the soundness of elaboration together with the soundness of the core language, taking inspiration from and synthesizing the respective approaches of Chau and Parreaux [5] and Fan and Parreaux [12].

We conclude by discussing related work (Section 5). The proofs of the theorems, the full definitions of our formalism, including the auxiliary functions and judgments, further discussions on possible design choices, and additional examples can be found in the appendix.

2 Overview

This section presents programming examples using our trait system with extensible variants in the context of a modular DSL adapted from the example of Hofer et al. [16]. We present the examples in the syntax of the MLscript programming language [33, 34], which features a conditional syntax for pattern matching that is both expressive and intuitive [6, 7].

2.1 Base Language

We start with a simple base language that defines a few basic regions in the 2D plane:

```
type Vector = [Double, Double]
trait BasicRegions {
```

```
type Region = Univ | Empty | Circle
class { Univ, Empty, Circle }
fun show(x: now/Region): String = if x is
  Univ   then "univ"
  Empty  then "empty"
  Circle then "circle" }
```

Type synonym `Vector` represents points with real-valued coordinates. Traits are the building blocks in our system. Inside trait `BasicRegions`, we define inner classes that represent the basic region data types: `Univ` as the universe, `Empty` as the empty region, and `Circle` as a circle with radius 1 at the origin[3]. Type `Region` is an extensible variant type that includes the three basic region variants. Definitions within a trait are mutually in scope, so `Region` may refer to inner classes declared after it. Method `show` pretty-prints a region to a string. The input `x` has type `now/Region`, referring to the `Region` extensible variant type with a `now` path prefix, which stands for the variants of `Region` defined up until *now*. Currently, `Region` only includes the three basic variants, so we have the *subtyping constraint* that `now/Region` is equivalent to the *union type* of those variants, where ≡ means the two types are subtypes of each other:

```
now/Region ≡ Univ | Empty | Circle
```

and the pattern matching expression `if x is ...` handles these cases.

2.2 Modular DSL via Trait Composition

Next, we extend the base language with a new operation on regions in a new trait `MyRegions`:

```
trait MyRegions requires BasicRegions {
  fun contains(v: Vector, e: now/Region): Boolean = if [v, e] is
    [_, Univ] then true
    [_, Empty] then false
    [[vx, vy], Circle] then vx * vx + vy * vy < 1 }
```

The `requires` clause of the trait, which plays a similar role to `extends` in Scala, indicates that `MyRegions` requires `BasicRegions`, so the `Region` variant type and the data classes in `BasicRegions` are available in `MyRegions`. This allows us to define a new method `contains` that checks if the point represented by the vector `v` is contained in region `e`. The pattern matching simply handles the variants of `Region` defined up until now, namely the three basic variants in `BasicRegions`.

Now we compose the traits into a *module* `SimpleRegions` that extends both `BasicRegions` and `MyRegions`. The module is a singleton instance of a class that composes both traits by multiple inheritance:

```
module SimpleRegions extends BasicRegions, MyRegions
SimpleRegions.contains([1, 0], new SimpleRegions/Univ) // true
SimpleRegions.show(new SimpleRegions/Circle)           // "circle"
```

The module now supports two operations on the region data type, and the super trait requirement of `MyRegions` is satisfied by `BasicRegions` appearing before it in the trait composition. We apply the methods to concrete instances of the inner data types. Note that the path of the inner classes is the name of the module, and that the method signatures of the two operations are:

```
SimpleRegions.contains: Vector → SimpleRegions/Region → Boolean
SimpleRegions.show:     SimpleRegions/Region → String
```

Unlike in the trait where the variant type is referenced as `now/Region`, in the module it has a *concrete* path prefix of the module name and the following subtyping constraint:

[3]The actual syntax of inner classes in traits has `this` as a path prefix, e.g., `this/Circle`, which distinguishes them from top-level classes (with an implicit `prog` prefix) and inner classes composed into other modules.

```
SimpleRegions/Region ≡ SimpleRegions/Univ | SimpleRegions/Empty
                                          | SimpleRegions/Circle
```

When viewed from outside of the trait, the variant type is *finalized* to include the variants defined in the module (or class) and is no longer extensible. However, one can still implement arbitrary extensions of the operations by defining new traits and composing them into modules. Next, we will show how to modularly extend `Region` and its associated operations with new variants in a robust way.

2.3 Type-Safe and Modular Variant Extension

Consider the following trait `ScaleMyRegions` that extends the region data type with a new variant type, representing a region scaled by a vector:

```
trait ScaleMyRegions requires BasicRegions, MyRegions {
  Region += Scale
  class Scale(v: Vector, x: this/Region) // assuming v has non-zero components
  override fun contains(v: Vector, e: now/Region): Boolean = if [v, e] is
    [[vx, vy], Scale([svx, svy], se)] then
      this.contains([vx / svx, vy / svy], se)
    else super.contains(v, e) }
```

The `+=` operation adds a new variant `Scale` to `Region` required from `MyRegions`. Inner class `Scale` has two fields: a vector and a region of type `this/Region`. The `this` path prefix allows us to refer to all variants of `Region` in the *class* that the trait is mixed into, which may include other variants only introduced in the future, reflecting the "open" nature of extensible variant types. Here, we want to scale a region that may involve unknown future variants; thus, the scaled region is given type `this/Region`. Of course, the variants in the object should include those defined up to now, so we have the subtyping relationship `now/Region <: this/Region`, which allows us to write, e.g., `this.Scale([2, 1], this.Circle)`.

As we have extended the region data type, we need to extend the operations with support for the new variant. Here, `contains` is annotated with **`override`**, indicating that it overrides the `contains` implementation inherited from `MyRegions`. The condition pattern extracts both components of `v` as `vx` and `vy`, the scaling vector in `Scale` as `[svx, svy]`, and the scaled region as `se`. The type of `se` is `this/Region`, as specified in the class constructor. To decide if point `v` is contained in the scaled region, we can multiply `v` by the reciprocal of the scaling vector and recursively check if the scaled point is contained in the region `se`. Here, we call `contains` on `this`, so this method call is *open-recursive*. The call is dispatched to the actual implementation in the object that `this` stands for. In the object, the variants defined *now* include all variants mixed in. Therefore, we have the type signature of `this.contains` in the trait as:

```
this.contains: Vector → this/Region → Boolean
```

where the path prefix `now` in `contains`'s signature in the trait is substituted with `this`, reflecting the crucial fact that `now/Region` and `this/Region` match in the end and allowing us to pass `se` in.

In the else-branch of the conditional where `e` is not `Scale`, we call `contains` on `super`, delegating the handling of other variants (such as `Univ`) to the inherited implementation. Now, what should the types of `e` in the else-branch and `super.contains` be? At the beginning, `e` has type `now/Region`, which comprises the variants of `Region` defined up to now, including the extension we add to `Region` in this trait, i.e. `Scale`, *and* the variants of `Region` inherited from the traits that will appear before `ScaleMyRegions` in the trait composition of the class (where `MyRegions` is required). Note that more variants can be defined and mixed into the class before `ScaleMyRegions` is mixed in, in

addition to the ones introduced in the required super traits. We use `super/Region` to represent those inherited base variants of `Region`, and have the following subtyping constraint:

```
now/Region ≡ super/Region | Scale
```

Note that the actual syntax of extensible variant types with the `super` path prefix additionally carries the method name in the prefix, i.e. `super.contains/Region`. We adopt a simplified presentation in the examples by writing `super/Region`, and discuss this design further in Section 2.5.

In the else-branch, we know that `e` cannot be `Scale`, so it should now have type `super/Region`. The refinement of `e`'s type is achieved by taking the *intersection* of `now/Region` and `~Scale`, the *negation* of `Scale`. Thanks to the Boolean-algebraic subtyping [34] upon which our system builds, we have the following subtyping relationship:

```
(now/Region & ~Scale) ≡ (super/Region & ~Scale) <: super/Region
```

As we are calling the inherited implementation of `contains` by `super.contains`, it should be able to handle those inherited base variants of `Region`, thus we have its signature as:

```
super.contains: Vector → super/Region → Boolean
```

where the path prefix `now` in the trait is substituted with `super`, and then we can pass `e` in.

However, as the reader may have noticed, we missed extending `show` to handle `Scale`, which could lead to the same issue as `BadArith1` in Section 1. Fortunately, our system rejects the current definition of `ScaleMyRegions`. In general, when the signature of an inherited concrete method in a trait is overridden to have a proper subtype, the trait is obligated to implement that new signature, as also enforced by the Scala trait system [24, 25]. Here, `ScaleMyRegions` is obligated to implement `show` with signature `now/Region → String`, which is a subtype of the overridden signature `super/Region → String`, as `now/Region` includes `Scale`. Importantly, this mechanism only requires us to type check each trait separately, to type check each method implementation only once, and to perform subtyping checks between type signatures, without any repeated type checking of expressions, which would compromise separate compilation. We fix the issue in `ScaleMyRegions` by extending `show` to handle `Scale`:

```
// inside `ScaleMyRegions`, add the following lines
override fun show(e: now/Region): String = if e is
  Scale(sv, sx) then "scale(" ++ sv.toString ++ ", " ++ this.show(sx) ++ ")"
  else super.show(e)
```

Independent of `ScaleMyRegions`, we also define a new variant `Union` that represents the union of two regions as an extension:

```
trait UnionMyRegions requires BasicRegions, MyRegions {
  Region += Union
  class Union(x: this/Region, y: this/Region)
  override fun contains(v: Vector, e: now/Region): Boolean = if e is
    Union(ux, uy) then this.contains(v, ux) or this.contains(v, uy)
    else super.contains(v, e)
  override fun show(e: now/Region): String = ... }
```

Finally, module `ExtendedRegions` composes the new traits with their required super traits:

```
module ExtendedRegions extends BasicRegions, MyRegions,
                               ScaleMyRegions, UnionMyRegions
```

Now `contains` and `show` in the module are able to handle `ExtendedRegions/Region`, a concrete variant type that encompasses all variants of `Region` introduced in the traits, including `Scale` and `Union`. Note that there is no dependency between `ScaleMyRegions` and `UnionMyRegions`, so these two can be composed in any order in the trait composition of the module.

2.4 Modular Optimization Passes with Nested Patterns

A powerful aspect of our system is that it allows arbitrarily many definitions of domain-specific optimization passes, by simply defining new traits with overriding optimization behaviors that potentially involve pattern matching with nested patterns, and composing them into modules. For example, we can define an optimization pass that reduces nested scaling constructs into a single scaling by the product of the nested vectors, and eliminates scaling by the identity vector:

```
trait OptimizeScale requires BasicRegions, MyRegions, ScaleMyRegions {
  fun normalize(e: now/Region): this/Region = if e is
    Empty | Univ | Circle then e
    Scale([vx1, vy1], Scale([vx2, vy2], e0)) then
      this.normalize(new Scale([vx1 * vx2, vy1 * vy2], e0))
    Scale([1, 1], e0) then this.normalize(e0)
    Scale(v, e0) then new Scale(v, this.normalize(e0)) }
```

It is important to notice that `normalize` is *not* an overriding method. That is, no `super.normalize` exists to be called in the required super traits. Based on this observation, `now/Region` is interpreted as equivalent to the union of all variants introduced in the required traits, namely:

```
OptimizeScale/normalize ⊢ Empty | Univ | Circle | Scale ≡ now/Region
```

so the pattern matching in `normalize` is exhaustive. This subtyping constraint is specific to the trait method `normalize` in `OptimizeScale` which is declared as non-overriding, while it may not be the case for other methods even in the same trait. Notice that `Scale` is a subtype of `now/Region`, which in turn is a subtype of `this/Region`. This allows us to supply `Scale` as `this/Region` in the open-recursive call and as the return value. Thanks to pattern matching being a first-class term construct in our language, we can easily desugar deep pattern matching into nested if-then-else expressions, with exhaustiveness guaranteed by the type system. Now we define a module:

```
module OptimizedRegions extends BasicRegions, MyRegions,
                                ScaleMyRegions, OptimizeScale
val exp = new OptimizedRegions/Scale([0.5, 0.5],
            new OptimizedRegions/Scale([2, 2], new OptimizedRegions/Circle))
OptimizedRegions.normalize(exp) // returns an OptimizedRegions/Circle object
```

Of course, one may mix in more traits overriding `normalize` with more complex behaviors or define other optimization passes. On the other hand, it is natural to ask how to modularly extend `normalize` to the `Union` variant, which is defined in `UnionMyRegions` independently of `OptimizeScale`, without breaking any existing trait definition.

Retroactive handling of mixed-in variants. Consider the following module definition:

```
module BadRegions extends BasicRegions, MyRegions, UnionMyRegions,
                          ScaleMyRegions, OptimizeScale // rejected!
```

The trait composition seemingly makes sense, as the requirements of each trait are indeed satisfied. This is also a desirable pattern, since a user may wish to modularly extend an existing language with new optimization passes without modifying the original language. However, accepting this module would lead to the same issue as `BadArith2` in Section 1: `Union` is never handled by `normalize`, while the signature of `BadRegions.normalize` would allow `Union` to be passed in as one of the `Region` variants in the object. Recall that when defining `normalize` in `OptimizeScale`, it was declared as *non-overriding*, and then it was typed with the assumption that `now/Region` is equivalent to the union of all variants *only* introduced in the required super traits. Our type system correctly rejects this module by interpreting `now/Region` in the actually implemented type of `normalize` as `Empty | Univ | Circle | Scale`, which does not

match the desired `Empty | Univ | Circle | Scale | Union` in the object, where `now/Region` and `this/Region` should meet.

To retroactively handle `Union` without modifying `OptimizeScale`, one can define an override implementation of `normalize`:

```
module FixedRegions extends BasicRegions, MyRegions, UnionMyRegions,
                            ScaleMyRegions, OptimizeScale {
  override fun normalize(e: now/Region | Union): this/Region = if e is
    Union(ux, uy) then new Union(this.normalize(ux), this.normalize(uy))
    else super.normalize(e) }
```

When type checking `FixedRegions.normalize`, `now/Region` is interpreted as the union of variants in `ScaleMyRegions` and its required super traits, with `UnionMyRegions` excluded. Now that `Union` is explicitly added, the implementation signature of `normalize` matches the desired signature that allows all variants including `Union`.

2.5 Design Overview

Below we give an overview of our approach and summarize key concepts.

Exhaustive pattern matching guarantees. As shown in previous examples, after each pattern match case, Boolean-algebraic subtyping refines the scrutinee type by intersecting it with the negation of the tested class, effectively ruling out the matched variant from the remainder scrutinee type. Our type system ensures that, when pattern matches in a source program contain no wildcard case, well-typed source programs never reach the implicit "catch-all" branch inserted at the end of a pattern match. In Scala, a `MatchError` exception is thrown in such implicit catch-all branches. In our design, all source pattern matches, including those scrutinizing extensible variants, are also desugared to conditional expressions with else-branches, but the implicit catch-all branches are encoded as `x: Nothing` type ascriptions where `x` is bound to the scrutinee in the innermost else-branch. This type ascription forces the type checker to prove that the remainder scrutinee in the final catch-all branch refines to `Nothing` after intersection with the negated class types of the preceding cases, and therefore that this branch must be unreachable to preserve type soundness.

In the modular case, a method over `now/A` may delegate the final remainder to `super` when that remainder type is below the assumed type of the inherited method, i.e. the method type substituted with the `super` path prefix. To ensure soundness, we check that upon class formation, the local assumptions made in each trait about the final object and inherited implementations are satisfied by the actual class composition.

Program structure and type forms. In our design, a program is composed of traits, classes, and a main expression at the top level. Program behaviors are implemented as methods in traits, and these methods can be partial and open to overriding with extensions. Top-level classes specify object states in their fields, while the object behaviors are obtained by trait composition. Modules, similar to Scala's **`object`**, are singleton class instances. Traits use **`requires`** clauses to extend their functionalities. These required super traits are to be composed into the class before the current trait. Inside each trait, one may define extensible variant types, inner classes, and methods. Inner classes are data types carrying fields as object states. Unlike top-level classes, inner classes do not contain any methods or inherit traits. Traits are always at the top level, i.e., cannot be defined in other traits.

Traits and classes are types in our system. The **`requires`** clause of traits and the **`extends`** clause of classes introduce nominal subtyping relationships between traits, and between classes and their super traits. Different *path* forms in the prefixes of class types distinguish top-level classes (with a `prog` prefix) and inner classes in different contexts. As inner classes only nest in top-level traits,

their path prefixes can only be a class name or `this`. Inner classes defined in a trait are referenced with the `this` path prefix in that trait and its downstream subtraits. When a trait is composed into a class, its inner classes become *concrete*: the `this` prefixes in their paths are substituted with the class name. Classes with different concrete path prefixes are considered distinct types [10].

Extensible variant types are defined in a trait with a collection of inner class names in the current trait's scope, and they can be later extended with more variants (i.e. inner classes) in the downstream traits. Inside a trait, an extensible variant type `A` is referenced with the `now` or the `this` path prefix in method signatures. These references are abstract types that represent different versions of the extensible variant type. In particular, the subtyping assumptions on `this/A` depend on `this` as the method call receiver. `now/A` is adapted to `this/A` for open-recursive calls via `this`, and is adapted to `super.f/A` for calls to an overridden implementation via `super`, which is specific to the current trait method `f`. Thus, one may consider our extensible variant types as a form of *trait*-dependent types. When a class is put together, both `this/A` and `now/A` are concretized: their path prefixes are substituted with the concrete class name, and they are interpreted as the union of all concrete variants collected from the trait composition.

Choice between `now/` and `this/`. In method signatures, one may choose between the `now` and the `this` path prefixes for extensible variant types. For an extensible variant type `A`, we identify the following usage patterns where `A` is referenced with different paths in various positions: `now/A` stands for variants defined up to now. It is used to define partial method implementations that pattern match on those variants, where a variant is either handled or delegated to the overridden implementation. These methods are to be fused together via open recursion when the class is formed. On the other hand, `this/A` represents all variants that will ever be introduced. It typically appears in the return types of method calls on `this`. `this/A` can also be examined by a pattern match that necessarily implements a default case. This is useful for implementing "one-off" methods that do not have to evolve with `A`, such as methods that determine whether the input is a specific variant. When `A` is extended later, no overriding obligation is introduced for such one-off methods.

Local assumptions of `super` methods. When `super` method calls involve extensible variant types, an important question arises: for an overriding method, which variants is the `super` call expected to handle? In our design, the overridden implementation is assumed to handle all variants inherited from the super traits composed *after* the method's non-overriding base implementation, together with those handled by the base implementation itself. Since different methods may have different base implementations that each handle a distinct set of variants (such as the base definitions of `show` in `BasicRegions` and `normalize` in `OptimizeScale` in Section 2.4), the assumption on which variants a `super` call handles is *per-method*. Therefore, the `super` path prefix necessarily carries the method name. For example, in `ScaleMyRegions`, the actual type of `super.contains` is `Vector → super.contains/Region → Boolean`. We treat the `super` path prefix as a type checker internal detail used for super call typing and method signature checks, rather than a type form for users to write. Further discussion of the design space of local assumptions of inherited methods can be found in the appendix.

3 Formalization

This section presents the formalism of our surface language, which supports traits with extensible variants and builds upon Boolean-algebraic subtyping [34]. Our formalism is also inspired by Featherweight Java [18] and Pathless Scala [24, 25].

$$
\begin{array}{rrcl}
\textit{Variant type name} & & & \mathtt{A}, \mathtt{B} \\
\textit{Top-level name} & \mathscr{N} & ::= & \mathtt{T}, \mathtt{U} \mid \mathtt{prog/C} \mid \mathtt{Object} \\
\textit{Class path} & \mathscr{P} & ::= & \mathtt{prog} \mid \mathbf{this} \mid \mathtt{C} \\
\textit{Type} & \mathtt{t} & ::= & \mathtt{T} \mid \mathscr{P}/\mathtt{C} \mid \mathtt{t} \to \mathtt{t} \mid \mathtt{Anything} \mid \mathtt{Nothing} \mid (\mathtt{t} \,\&\, \mathtt{t}) \mid (\mathtt{t} \mid \mathtt{t}) \mid \mathtt{\sim t} \mid \mathscr{A} \\
\textit{Variant type} & \mathscr{A} & ::= & \mathbf{this}/\mathtt{A} \mid \mathbf{now}/\mathtt{A} \mid \mathtt{C}/\mathtt{A} \mid \mathbf{super}.\mathtt{f}/\mathtt{A} \\
\textit{Expression} & \mathtt{e} & ::= & \mathtt{x}, \mathtt{y} \mid \mathbf{this} \mid \mathbf{super}.\mathtt{f} \mid \mathtt{e} : \mathtt{t} \mid \mathtt{e}.\mathtt{f} \\
& & & \mid (\mathtt{x} : \mathtt{t}) \Rightarrow \mathtt{e} \mid \mathtt{e}(\mathtt{e}) \mid \mathbf{new}\ \mathscr{P}/\mathtt{C}(\overline{\mathtt{e}}) \mid \mathbf{if}\ \mathtt{e}\ \mathbf{is}\ \mathscr{P}/\mathtt{C}\ \mathbf{then}\ \mathtt{e}\ \mathbf{else}\ \mathtt{e} \\
\textit{Top-level definition} & \mathscr{D} & ::= & \mathbf{class}\ \mathtt{prog/C}(\overline{\mathtt{f} : \mathtt{t}})\ \mathbf{extends}\ \overline{\mathtt{T}} \\
& & & \mid \mathbf{trait}\ \mathtt{T}\ \mathbf{requires}\ \overline{\mathtt{T}}\ \{\overline{\mathscr{T}};\ \overline{\mathbf{class}\ \mathbf{this}/\mathtt{C}(\overline{\mathtt{f} : \mathtt{t}})};\ \overline{\mathscr{M}}\} \\
\textit{Type definition} & \mathscr{T} & ::= & \mathbf{type}\ \mathtt{A} = \overline{\mathbf{this}/\mathtt{C}} \mid \mathtt{B}\ \mathtt{+=}\ \overline{\mathbf{this}/\mathtt{C}} \\
\textit{Method} & \mathscr{M} & ::= & \mathtt{f} : \mathtt{t} \mid \mathtt{f} : \mathtt{t} = \mathtt{e} \\
\textit{Program} & \mathtt{P} & ::= & \overline{\mathscr{D}};\ \mathtt{e} \\
\textit{Context} & \mathtt{G} & ::= & \epsilon \mid \mathtt{G} \cdot (\mathtt{x} : \mathtt{t}) \qquad \mathtt{H} ::= \epsilon \mid \mathtt{T}/\mathtt{f} \qquad \mathtt{T}_? ::= \epsilon \mid \mathtt{T} \\
\textit{Shorthand} & & & \mathbf{if}\ \mathtt{x}\ \mathbf{is}\ \mathscr{P}/\mathtt{C}\ \mathbf{then}\ \mathtt{e} \triangleq \mathbf{if}\ \mathtt{x}\ \mathbf{is}\ \mathscr{P}/\mathtt{C}\ \mathbf{then}\ \mathtt{e}\ \mathbf{else}\ (\mathtt{x} : \mathtt{Nothing}) \\
& & & \mathtt{G} \cdot (\mathtt{e} : \mathtt{t}) \triangleq \mathtt{G} \cdot (\mathtt{x} : \mathtt{t}) \text{ if } \mathtt{e} = \mathtt{x}, \text{ and } \mathtt{G} \text{ otherwise}
\end{array}
$$

Fig. 1. Trait language syntax.

3.1 Syntax

Figure 1 presents the syntax of the surface language, which mostly formalizes the concepts discussed in Section 2.5. Top-level names $\mathscr{N}$ encompass both traits `T` and top-level classes `prog/C`, together with `Object`, a special trait that all objects are subtypes of. Types include trait types and class types with path prefixes, Boolean-algebraic types constructed by union (`t | t`), intersection (`t & t`), and negation (`~t`) of types, as well as standard function types, top (`Anything`), and bottom (`Nothing`) types. For an extensible variant type `A`, **`super`**`.f/A` refers to the variants handled by method `f` in the super traits; here **`super`**`.f` is a path prefix rather than a method call, and this type is not allowed in user method signatures. Expressions include term variables, **`this`** references, calls to inherited methods via **`super`**, method accesses, and type ascriptions. Function expressions use lambda syntax `(x : t) ⇒ e` with application `e(e)`. Object construction follows the pattern **`new`** $\mathscr{P}/\mathtt{C}(\overline{\mathtt{e}})$. The conditional construct **`if`** `e` **`is`** $\mathscr{P}/\mathtt{C}$ **`then`** `e` **`else`** `e` is for class-based pattern matching that tests whether the scrutinee is an instance of a specific class. When a conditional branching on a term variable `x` has no else-branch, we desugar it to having `(x : Nothing)` as the else-branch, which is never reachable as discussed in Section 2.5.

The language supports two kinds of top-level definitions: *top-level classes*, which define concrete data types that have fields and may extend multiple traits, as well as *traits* themselves. Note that class field selections have the same syntax as method calls, and field definitions override any method implementation inherited from the traits. In a trait, type definitions either define new extensible variant types or extend existing ones with the `+=` operator followed by a list of inner class names. Inner classes are defined within traits using the special **`this`** prefix and do not inherit from any trait. Methods can be either abstract (`f : t`) or concrete (`f : t = e`). As override annotations can be enforced by a simple external check, in our formalism by default, a method implementation always overrides the inherited implementation of the same method if there is one. Typing context `G` tracks variable bindings. As a shorthand, when pushing a binding of a term `e` into the typing context, it will be ignored if `e` is not a term variable. `H` indicates the current trait method if we are typing a method body.

$$\textsc{T-Method}\ \frac{\mathtt{G} \vdash_{\mathtt{H}} \mathtt{e} : \mathtt{t}_0 \qquad \mathsf{mtype}_{\mathtt{trait(H)}}(\mathtt{f}, \mathtt{t}_0) = \mathtt{t} \qquad (\mathtt{t}_0 = \mathtt{this}/\mathtt{C}) \vee (\epsilon \vdash \mathtt{t})}{\mathtt{G} \vdash_{\mathtt{H}} \mathtt{e.f} : \mathtt{t}}$$

$$\textsc{T-ThisMethod}\ \frac{\mathsf{mtype}_{\mathtt{T}}(\mathtt{f}, \mathtt{T}) = \mathtt{t}}{\mathtt{G} \vdash_{\mathtt{T/g}} \mathtt{this.f} : \mathtt{t}\,[\mathtt{now} \mapsto \mathtt{this}]}$$

$$\textsc{T-Super}\ \frac{\mathtt{super}(\mathtt{f}, \mathtt{T}) = \mathtt{t}}{\mathtt{G} \vdash_{\mathtt{T/g}} \mathtt{super.f} : \mathtt{t}} \qquad \textsc{T-Var}\ \frac{\mathtt{G}(\mathtt{x}) = \mathtt{t}}{\mathtt{G} \vdash_{\mathtt{H}} \mathtt{x} : \mathtt{t}} \qquad \textsc{T-This}\ \frac{}{\mathtt{G} \vdash_{\mathtt{T/f}} \mathtt{this} : \mathtt{T}}$$

$$\textsc{T-Abs}\ \frac{\mathtt{G} \cdot (\mathtt{x} : \mathtt{t}_1) \vdash_{\mathtt{H}} \mathtt{e} : \mathtt{t}_2}{\mathtt{G} \vdash_{\mathtt{H}} (\mathtt{x} : \mathtt{t}_1) \Rightarrow \mathtt{e} : \mathtt{t}_1 \to \mathtt{t}_2}$$

$$\textsc{T-App}\ \frac{\mathtt{G} \vdash_{\mathtt{H}} \mathtt{e}_1 : \mathtt{t}_1 \to \mathtt{t}_2 \qquad \mathtt{G} \vdash_{\mathtt{H}} \mathtt{e}_2 : \mathtt{t}_1}{\mathtt{G} \vdash_{\mathtt{H}} \mathtt{e}_1(\mathtt{e}_2) : \mathtt{t}_2} \qquad \textsc{T-Asc}\ \frac{\mathtt{G} \vdash_{\mathtt{H}} \mathtt{e} : \mathtt{t}}{\mathtt{G} \vdash_{\mathtt{H}} (\mathtt{e} : \mathtt{t}) : \mathtt{t}}$$

$$\textsc{T-New}\ \frac{\mathsf{ctor}_{\mathtt{trait(H)}}(\mathscr{P}/\mathtt{C}) = \overline{\mathtt{f}_i : \mathtt{t}_i}^{\,i} \qquad \overline{\mathtt{G} \vdash_{\mathtt{H}} \mathtt{e}_i : \mathtt{t}_i}^{\,i}}{\mathtt{G} \vdash_{\mathtt{H}} \mathtt{new}\ \mathscr{P}/\mathtt{C}(\overline{\mathtt{e}_i}^{\,i}) : \mathscr{P}/\mathtt{C}}$$

$$\textsc{T-IfElse}\ \frac{\begin{array}{c}\mathtt{G} \vdash_{\mathtt{H}} \mathtt{e} : \mathtt{t}_0\,\&\,\mathtt{Object} \qquad \mathsf{cnames}(\mathsf{trait}(\mathtt{H})) \vdash \mathscr{P}/\mathtt{C} \\ \mathtt{G} \cdot (\mathtt{e} : \mathtt{t}_0\,\&\,\mathscr{P}/\mathtt{C}) \vdash_{\mathtt{H}} \mathtt{e}_1 : \mathtt{t} \qquad \mathtt{G} \cdot (\mathtt{e} : \mathtt{t}_0\,\&\,\sim\mathscr{P}/\mathtt{C}) \vdash_{\mathtt{H}} \mathtt{e}_2 : \mathtt{t}\end{array}}{\mathtt{G} \vdash_{\mathtt{H}} \mathtt{if\ e\ is}\ \mathscr{P}/\mathtt{C}\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 : \mathtt{t}} \qquad \textsc{T-Sub}\ \frac{\mathtt{G} \vdash_{\mathtt{H}} \mathtt{e} : \mathtt{t}_0 \qquad \mathtt{H} \vdash \mathtt{t}_0 \leq \mathtt{t}}{\mathtt{G} \vdash_{\mathtt{H}} \mathtt{e} : \mathtt{t}}$$

$$\mathsf{mtype}_{\mathtt{T}_?}(\mathtt{f}, \mathscr{P}/\mathtt{C}) = \mathtt{t}\,[\mathtt{this} \mapsto \mathscr{P}] \quad \text{if } (\mathtt{f} : \mathtt{t}) \in \mathsf{ctor}_{\mathtt{T}_?}(\mathscr{P}/\mathtt{C})$$

$$\mathsf{mtype}_{\mathtt{T}_?}(\mathtt{f}, \mathtt{prog}/\mathtt{C}) = \mathtt{t}\,[\mathtt{this} \mapsto \mathtt{C}][\mathtt{now} \mapsto \mathtt{C}] \quad \text{if } \mathtt{f} \notin \mathsf{ctor}_{\mathtt{T}_?}(\mathtt{prog}/\mathtt{C}) \text{ and } \mathsf{mtype}_{\mathtt{T}_?}(\mathtt{f}, \mathtt{Anything}\,\&\,\overline{\mathsf{parents}(\mathtt{prog}/\mathtt{C})}) = \mathtt{t}$$

$$\mathsf{mtype}_{\mathtt{T}_?}(\mathtt{f}, \mathtt{T}) = \begin{cases} \mathtt{t} & \text{if } (\mathtt{f} : \mathtt{t}) \in \mathtt{T} \text{ or } (\mathtt{f} : \mathtt{t} = \mathtt{e}) \in \mathtt{T} \\ \mathsf{mtype}_{\mathtt{T}_?}(\mathtt{f}, \mathtt{Anything}\,\&\,\overline{\mathtt{U}_\mathtt{T}}) & \text{if } \mathtt{f} \notin \mathtt{T} \text{ and } \mathsf{parents}(\mathtt{T}) = \overline{\mathtt{U}_\mathtt{T}} \end{cases}$$

$$\mathsf{mtype}_{\mathtt{T}_?}(\mathtt{f}, \mathtt{t}_1\,\&\,\mathtt{t}_2) = \mathsf{mtype}_{\mathtt{T}_?}(\mathtt{f}, \mathtt{t}_1)\ \textcircled{\&}\ \mathsf{mtype}_{\mathtt{T}_?}(\mathtt{f}, \mathtt{t}_2)$$

$$\mathsf{mtype}_{\mathtt{T}_?}(\mathtt{f}, \mathtt{t}) = \varnothing \quad \text{otherwise}$$

$$\mathsf{super}(\mathtt{f}, \mathtt{T}) = \mathtt{t}\,[\mathtt{now} \mapsto \mathtt{super.f}] \quad \text{if } \mathsf{searchimpl}(\mathtt{f}, \mathsf{parents}(\mathtt{T})) = (\mathtt{t}, \mathtt{U})$$

$$\mathsf{searchimpl}(\mathtt{f}, (\overline{\mathtt{T}}, \mathtt{T})) = \begin{cases} (\mathtt{t}, \mathtt{T}) & \text{if } (\mathtt{f} : \mathtt{t} = \mathtt{e}) \in \mathtt{T} \\ \mathsf{searchimpl}(\mathtt{f}, \overline{\mathtt{T}}) & \text{if } (\mathtt{f} : \mathtt{t} = \mathtt{e}) \notin \mathtt{T} \end{cases}$$

$$\mathsf{searchimpl}(\mathtt{f}, \epsilon) = \varnothing$$

Fig. 2. Typing rules of the surface language.

3.2 Expression Typing

Figure 2 shows the typing rules for surface language expressions. Judgment $\mathtt{G} \vdash_{\mathtt{H}} \mathtt{e} : \mathtt{t}$ states that expression e has type t under typing context G, and the trait method currently being typed is H. When H is ϵ, it means we are typing a top-level expression. Typing for lambda expressions, applications, object construction, and ascriptions is standard. For T-This, as `this` is only allowed in a trait method body, H must be a specific trait method name, i.e. T/f, and the trait T is returned as `this`'s type. For T-New, the constructor signature is looked up in the current trait by $\mathsf{ctor}_{\mathtt{T}_?}$, which finds the constructor signature of a class. If $\mathtt{T}_?$ is a specific trait and an inner class is being looked up, it returns the constructor signature of the inner class in that trait. trait(H) returns a $\mathtt{T}_?$, which is either the current trait if we are typing a method body, or ϵ if H is ϵ.

Method access. There are three typing rules that type method calls differently according to the receiver expression. T-Method handles method calls on general expressions. The method type lookup $\mathsf{mtype}_{\mathtt{T}_?}(\mathtt{f}, \mathtt{t}_0)$ returns the method's signature from the receiver type $\mathtt{t}_0$. When $\mathtt{t}_0$ is an inner class type `this`/C, T-Method allows access to its fields. However, in the general case, the

method signature has to be *closed*, formulated as the type well-formedness judgment $\epsilon \vdash \mathtt{t}$, which checks that the method signature `t` contains no subexpressions with `this` or `now` path prefixes. Allowing access to methods with extensible variant types in general would be unsound (an example illustrating this is given in the appendix). The root cause is that when typing trait methods, we make subtyping assumptions on extensible variant types, and for a general receiver expression we do not know which assumptions it makes. Only when the receiver has an inner class type in a specific trait context, or is exactly `this` or `super`, can we be sure that the current trait method's assumptions apply. The following two rules handle such situations.

For method calls on `this` inside the method body of `T/g`, T-ThisMethod allows access to methods defined in the current trait. Crucially, the returned type undergoes a substitution $\mathtt{t}[\mathtt{now} \mapsto \mathtt{this}]$, which substitutes `now` path prefixes in the method signature of `f` with `this`, reflecting that method calls with `this` will be late-bound to the implementation in the object, where `this` and `now` versions of extensible variant types meet. T-Super enables calls to inherited method implementations via `super.f`. The super method lookup $\mathrm{super}(\mathtt{f}, \mathtt{T})$ finds the *nearest concrete method*'s signature in the `requires` clause of the current trait `T` by calling searchimpl. $\mathrm{searchimpl}(\mathtt{f}, \overline{\mathtt{T}})$ iterates over a list of traits from the rightmost to find the first implementation of `f` and the trait that provides it. The type substitution $\mathtt{t}[\mathtt{now} \mapsto \mathtt{super.f}]$ substitutes `now` path prefixes with `super.f`, since methods involving extensible variant types accessed through `super` handle certain variants as expected by method `f`. This formalizes the understanding of `super` method calls discussed in Section 2.5. When typing traits, specific interpretations of the extensible variant types will be captured by the subtyping relation, as we will see in the next subsection.

Method type lookup. The auxiliary function $\mathrm{mtype}_{\mathtt{T}_?}(\mathtt{f}, \mathtt{t})$ looks up the method signature in a type. For class types, it first checks the constructor fields. For inner class fields, `this` is substituted with the concrete class name if $\mathscr{P}$ is a class name. For example, to get the type of field `x` of `ExtendedRegions/Scale` in Section 2.3, ctor looks up `this/Scale`'s constructor signature in its defining trait `ScaleMyRegions`, and the returned type `this/Region` gets `this` substituted with `ExtendedRegions`. If the method name does not appear as a constructor field, it then looks up the method signature in the intersection type of the super traits of that top-level class, where both `now` and `this` path prefixes are substituted with the concrete class name. This formalizes how the type of `SimpleRegions.contains` is resolved in Section 2.2.

For trait types, it looks up methods in the trait definition, then recursively searches parent traits if the method is neither declared nor defined, by looking up the intersection of the super traits as returned by parents. For an intersection type $\mathtt{t}_1 \,\&\, \mathtt{t}_2$, it computes the intersection of method types returned for $\mathtt{t}_1$ and $\mathtt{t}_2$ using the $\textcircled{\&}$ operator, which returns the intersection of two types, or one of them if the other is $\varnothing$, or $\varnothing$ if both are $\varnothing$. Notably, mtype returns $\varnothing$ for a union type, whose values must be upcast to a common supertype for method type lookup, following the design of Scala [25].

Conditional expressions. Rule T-IfElse types the conditional construct, which performs class-based pattern matching on objects. The scrutinee expression must have a subtype of `Object` to make sure the scrutinee is a class instance. When the scrutinee is a term variable, we refine the scrutinee's type in each branch, as the variable can be mentioned in the branches. In the then-branch, the scrutinee has type $\mathtt{t}_0 \,\&\, \mathscr{P}/\mathtt{C}$, the intersection with the tested class, while in the else-branch it has type $\mathtt{t}_0 \,\&\, \sim\mathscr{P}/\mathtt{C}$, the intersection with the negation of the tested class. This is represented by adding a binding of the *scrutinee* at its refined type to the typing context, using the shorthand in Figure 1. Note that to make sure the inner class reference is well-scoped when $\mathscr{P}$ is `this`, we require that `this/C` is well-formed with the inner class names in the current trait, as returned by $\mathrm{cnames}(\mathrm{trait}(\mathtt{H}))$.

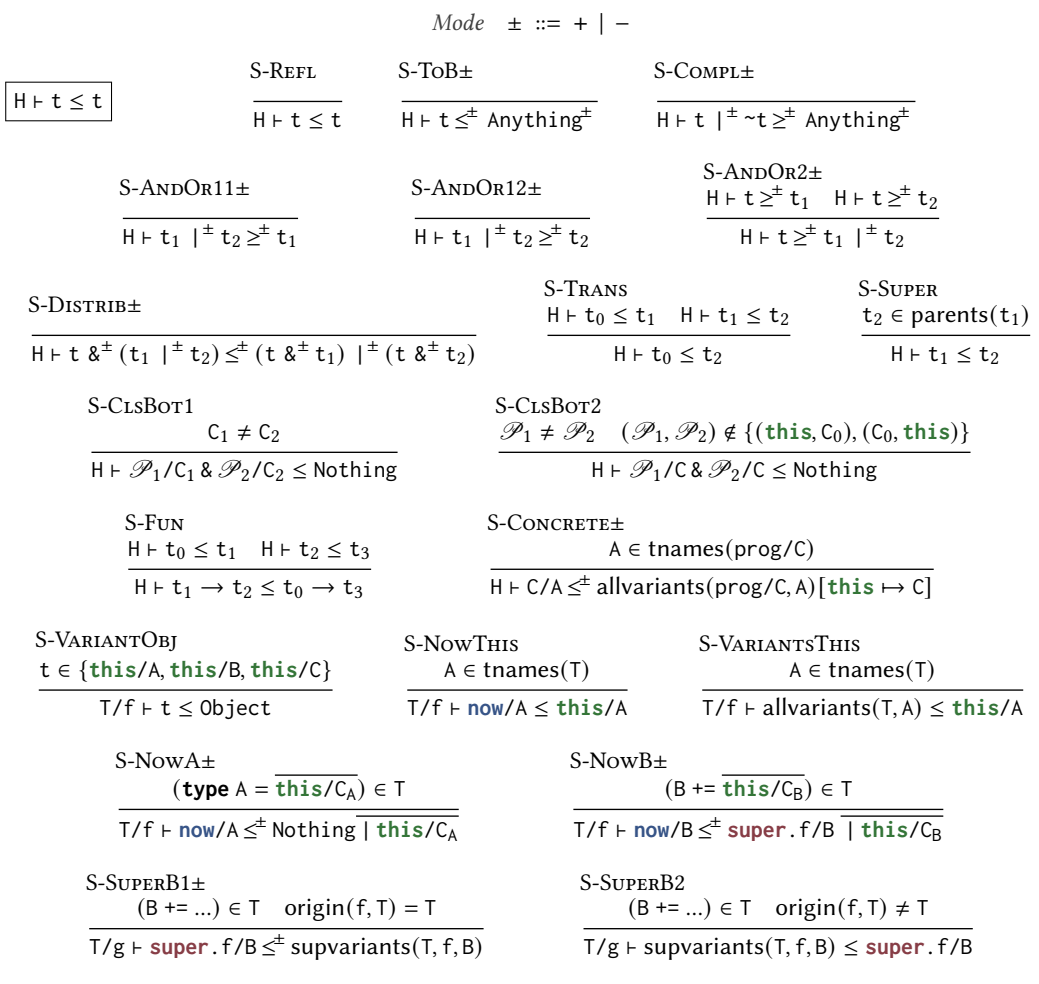


Fig. 3. Declarative subtyping.

3.3 Subtyping

Figure 3 shows the declarative subtyping rules for the surface language. The subtyping rules can be categorized into two parts: rules with H as the context are applicable both in a trait and at the top level, while those with a method name T/f in the context establish subtyping assumptions on extensible variant types specific to the trait method. Our subtyping builds upon Boolean-algebraic subtyping [34], where types form a Boolean algebra with union, intersection, and negation constructs. This foundation allows us to encode extensible variant types as unions of class types and to type pattern matches that eliminate them, which serves as a basis for modular type checking of traits with extensible variant types. We use positive and negative *modes* on the direction of subtyping, the Anything type, and the intersection and union type constructors to give a compact presentation of dual subtyping rules, following the convention of Parreaux and Chau [34] and Oliveira et al. [31]. When the mode is positive, it is simply ignored; when it is negative, the direction of subtyping is reversed, Anything becomes Nothing, and & and | are swapped.

Boolean-algebraic rules. Subtyping corresponds to an ordering on this algebra, a complemented distributive lattice satisfying the standard Boolean laws: commutativity, associativity, distributivity, and De Morgan's laws [34]. Rule S-ToB± makes every type a subtype of the top type in the positive mode and, dually, a supertype of the bottom type in the negative mode. Rule S-Compl± establishes that the union of a type and its complement is equivalent to the top type, while their intersection is equivalent to the bottom type, reflecting the complement laws of Boolean algebra. Rules S-AndOr11±, S-AndOr12±, and S-AndOr2± capture the lattice properties of unions and intersections: the union of two types represents their least upper bound, while the intersection represents their greatest lower bound. Rule S-Distrib± ensures distributivity of intersection over union, and vice versa.

Nominal subtyping, disjointness, and concrete variant types. Rule S-Super introduces nominal subtyping relationships between a class or trait and its super traits through the parents function, which returns the super traits of that class or trait as declared. Note that `Object`, as a trait type assumed to be inherited by any object, is always returned by parents. Rules S-ClsBot1 and S-ClsBot2 ensure that class types with distinct names or path prefixes are disjoint by having their intersection equivalent to the bottom type `Nothing`. This disjointness relationship is crucial for accurate type refinement in pattern matching, as it ensures that pattern matching for one class type excludes all others. Note that an inner class with path prefix **`this`** cannot be deemed disjoint from an inner class of the same name whose path prefix is a concrete class C_0: the **`this`** path prefix may itself refer to C_0, namely when the trait is composed into C_0. S-Concrete± equates a concrete variant type `C/A` with the union of all variants of `A` collected from the trait composition of class `prog/C`, as returned by allvariants(`prog/C`, `A`) after the **`this`** path prefix is concretized as class `C`.

Algorithmic subtyping. Our subtyping builds on Boolean-algebraic subtyping, whose complete type inference is studied in prior work on MLstruct [5, 34]. We foresee no fundamental challenge in its algorithmic formulation, as the existing type constructs scale straightforwardly to our needs. We do not include function merging and other algebraic rules from MLstruct, as they are not essential for the soundness of our approach, albeit crucial for implementing algorithmic subtyping based on *normal form constraining* [34]. We also foresee no issue in incorporating such rules into our type system. The rules for extensible variant types, i.e. those entailed by a trait method in Figure 3, can be implemented simply as subtyping bounds in the typing context, and are clearly separable from the pure Boolean-algebraic rules handled by the pre-existing algorithmic subtyping approach.

Example. We walk through typing derivations for the following program, which illustrates how Boolean-algebraic subtyping plays a crucial role in type checking pattern matches and exhaustiveness checking. It also shows how the **`now`** path prefix in a user signature is adapted to the **`this`** and **`super`**`.f` path prefixes, or to a concrete class name, when typing method calls on different forms of receivers:

```
trait T0 { type A = this/C0; class this/C0
  fun f(x: now/A): Int = if x is this/C0 then 0 }
trait T1 requires T0 { A += this/C1; class this/C1(g: this/A)
  fun f(x: now/A): Int = if x is this/C1 then this.f(x.g) else super.f(x) }
```

We write `e` for the body of `T1/f` and type it by invoking T-IfElse:

$$\frac{\texttt{x} : \texttt{now/A} \vdash_{\texttt{T1/f}} \texttt{x} : \texttt{now/A \& Object} \qquad \texttt{cnames(T1)} \vdash \texttt{this/C1} \qquad \mathcal{R}_1 \qquad \mathcal{R}_2}{\texttt{x} : \texttt{now/A} \vdash_{\texttt{T1/f}} \texttt{e} : \texttt{Int}}\ \text{T-IfElse}$$

The first premise establishes the scrutinee typing required by T-IfElse: T-Var gives `x` its annotated type **`now`**`/A`, which T-Sub weakens to **`now`**`/A & Object`. Then we check that the inner class **`this`**`/C1` to

match against is in the scope of `T1`. After typing both branches at `Int` in subderivations $\mathcal{R}_1$ and $\mathcal{R}_2$, we conclude that `e` has type `Int`.

The typing of the then-branch $\mathcal{R}_1$ under context $\mathtt{G}_1 = (\mathtt{x} : \mathtt{now/A}) \cdot (\mathtt{x} : \mathtt{now/A} \,\&\, \mathtt{this/C1})$ is as follows, where the scrutinee variable `x` is refined to the intersection of its original type and the matched class type:

$$\dfrac{\dfrac{\mathtt{mtype}_{\mathtt{T1}}(\mathtt{f}, \mathtt{T1}) = \mathtt{now/A} \to \mathtt{Int}}{\mathtt{G}_1 \vdash_{\mathtt{T1/f}} \mathtt{this.f} : (\mathtt{now/A} \to \mathtt{Int})\,[\mathtt{now} \mapsto \mathtt{this}]}\ \text{T-ThisMethod} \qquad \mathtt{G}_1 \vdash_{\mathtt{T1/f}} \mathtt{x.g} : \mathtt{this/A}}{\mathtt{G}_1 \vdash_{\mathtt{T1/f}} \mathtt{this.f(x.g)} : \mathtt{Int}}\ \text{T-App}$$

Notably, since the method is being called on `this`, T-ThisMethod substitutes the `now` path prefix of the signature declared in `T1` with `this`, reflecting that the call is late-bound to the implementation in the resulting object. Its argument is typed by T-Method. The refined binding of `x` is picked up by T-Var and narrowed to the inner class type `this/C1`. The field access is given type `this/A` by looking up the fields of `this/C1` in trait `T1` with mtype. Since the receiver has an inner class type, T-Method allows access to field `g` returning the extensible variant type `this/A`.

$\mathcal{R}_2$ types the super method call in the typing context $\mathtt{G}_2 = (\mathtt{x} : \mathtt{now/A}) \cdot (\mathtt{x} : \mathtt{now/A} \,\&\, \mathtt{\sim this/C1})$ for the else-branch. T-Super finds the implementation required from `T0`, i.e. $\mathtt{searchimpl}(\mathtt{f}, \mathtt{parents}(\mathtt{T1})) = (\mathtt{now/A} \to \mathtt{Int}, \mathtt{T0})$, and substitutes the `now` path prefixes with `super.f`:

$$\dfrac{\dfrac{\mathtt{super}(\mathtt{f}, \mathtt{T1}) = (\mathtt{now/A} \to \mathtt{Int})\,[\mathtt{now} \mapsto \mathtt{super.f}]}{\mathtt{G}_2 \vdash_{\mathtt{T1/f}} \mathtt{super.f} : \mathtt{super.f/A} \to \mathtt{Int}}\ \text{T-Super} \qquad \dfrac{\mathtt{G}_2 \vdash_{\mathtt{T1/f}} \mathtt{x} : \mathtt{now/A} \,\&\, \mathtt{\sim this/C1} \qquad \mathtt{T1/f} \vdash \mathtt{now/A} \,\&\, \mathtt{\sim this/C1} \le \mathtt{super.f/A}}{\mathtt{G}_2 \vdash_{\mathtt{T1/f}} \mathtt{x} : \mathtt{super.f/A}}\ \text{T-Sub}}{\mathtt{G}_2 \vdash_{\mathtt{T1/f}} \mathtt{super.f(x)} : \mathtt{Int}}$$

The subtyping premise to type the argument `x` at `super.f/A` is crucial: it holds because `C1`, the only variant `T1` adds to `A`, has just been excluded by the pattern match. The remaining variants are expected to be handled by the overridden implementation. We will provide detailed explanations of the subtyping rules for extensible variant types in Section 3.4. By chaining the subtyping relationships with S-Trans, we have:

$$\begin{aligned}
\mathtt{T1/f} \vdash \mathtt{now/A} \,\&\, \mathtt{\sim this/C1} &\le (\mathtt{super.f/A} \mid \mathtt{this/C1}) \,\&\, \mathtt{\sim this/C1} && \text{by S-NowB+, S-AndOr2−}\\
&\le \mathtt{super.f/A} \,\&\, \mathtt{\sim this/C1} \mid \mathtt{this/C1} \,\&\, \mathtt{\sim this/C1} && \text{by S-Distrib+}\\
&\le \mathtt{super.f/A} \,\&\, \mathtt{\sim this/C1} \mid \mathtt{Nothing} && \text{by S-Compl−}\\
&\le \mathtt{super.f/A} && \text{by S-AndOr2+, S-AndOr11−}
\end{aligned}$$

Finally, we compose the two traits into a top-level class: `class prog/C extends T0, T1`. When typing a method call `(new prog/C).f` by T-Method, the two signatures of `f` inherited from `T0` and `T1` are intersected, and both of their `now` path prefixes are substituted with the concrete class name:

$$\mathtt{mtype}_{\epsilon}(\mathtt{f}, \mathtt{prog/C}) = \big((\mathtt{now/A} \to \mathtt{Int}) \,\&\, (\mathtt{now/A} \to \mathtt{Int})\big)\,[\mathtt{this} \mapsto \mathtt{C}]\,[\mathtt{now} \mapsto \mathtt{C}]$$

which is the closed type $(\mathtt{C/A} \to \mathtt{Int}) \,\&\, (\mathtt{C/A} \to \mathtt{Int})$. Now we can pass in instances of `C/C0` and `C/C1`, as they are both subtypes of `C/A` by S-Concrete−.

3.4 Subtyping Extensible Variant Types

We introduce the crucial subtyping rules in Figure 3 that capture the subtyping assumptions of extensible variant types with examples, presented in the surface language syntax with more convenient pattern matching constructs. Consider the following base trait definition:

```
trait Base { type Exp = this/One; class this/One }
```

Rule S-NowA± states that for `A` newly defined in this trait, `now/A` is equivalent to the union of the variants introduced by `A`'s definition. In `Base`, `now/Exp` is equivalent to `this/One`. Rules S-NowThis

and S-VariantsThis state that for any A in the trait scope (i.e. type names in tnames(T)), now/A and all variants introduced to A as returned by allvariants(T, A) are subtypes of this/A. Therefore, both now/Exp and this/One are subtypes of this/Exp in Base and of Object by S-VariantObj, so they can be scrutinized by pattern matching. Now we consider the following trait extension:

```
trait ExtTwo requires Base {
  Exp += this/Two; class this/Two;
  fun eval(x: now/Exp): Int = if x is this/One then 1 | this/Two then 2 }
```

Now that Exp is extended with a new variant Two, the interpretation of now/Exp is given by rule S-NowB±, which states that for any B extended with new variants, now/B is equivalent to the union of new variants and super.f/B, where f is the method name. In the context of method ExtTwo/eval, we have now/Exp equivalent to this/Two | super.eval/Exp.

Moreover, eval is a non-overriding base method in ExtTwo, as there is no eval implementation in the required super trait Base. This is given by origin(eval, ExtTwo) = ExtTwo, where origin(f, T) returns the trait where the base implementation of method f in trait T is defined. For method f, with its base implementation in T, rule S-SuperB1± interprets super.f/B as equivalent to the union of B variants inherited from the super traits, as computed by supvariants(T, f, B).

supvariants formalizes the understanding of extensible variant types with super path prefix discussed in Section 2.5. For a base method f in T, the variants of B inherited from all parent traits of T are returned, which is exactly this/One for ExtTwo/eval. variants($\overline{\mathsf{T}}$, B) returns the variants introduced to B in the sequence of traits $\overline{\mathsf{T}}$. Combining S-NowB±, S-SuperB1±, and subtyping transitivity, we have now/Exp equivalent to this/One | this/Two in ExtTwo/eval. Therefore, the pattern match is exhaustive by handling exactly One and Two. Next, we add more variants to Exp and override eval in the following two traits:

```
trait ExtZero requires Base { Exp += this/Zero; class this/Zero }
trait ExtZeroTwo requires Base, ExtZero, ExtTwo {
  Exp += this/Three; class this/Three;
  fun eval(x: now/Exp): Int = if x is this/Three then 3 else super.eval(x) }
```

Note that the extension to Exp in ExtZero is independent of ExtTwo. Trait ExtZeroTwo requires ExtZero and ExtTwo to be mixed together, and introduces a new variant Three to Exp. As methods override those with the same name, ExtZeroTwo/eval must be an overriding implementation of eval, which is originally defined in ExtTwo, as returned by origin(eval, ExtZeroTwo). Again by S-NowB±, we have now/Exp equivalent to super.eval/Exp | this/Three in ExtZeroTwo/eval. The pattern match first handles Three. In the else-branch where x is refined to have type super.eval/Exp, it delegates the handling of the remaining variants by passing x to super.eval with type super.eval/Exp ⟶ Int. Therefore, ExtZeroTwo/eval type checks.

Unlike ExtTwo/eval, ExtZeroTwo/eval is an overriding implementation. The interpretation of super.eval/Exp is governed by rule S-SuperB2, where a *lower bound* on super.f/B is established when f is *not* originally defined in the current trait. Rather than the equivalence relationship given by rule S-SuperB1±, rule S-SuperB2 states that supvariants(T, f, B) is a subtype of super.f/B. Additional variants of B may be introduced by traits composed before T in the final class, so only a lower bound can be given.

The second case of supvariants returns the union of variants expected to be already handled by the overridden implementation of T/f. Those are in the traits $\overline{\mathsf{T}_i}$ composed before T, but not before the origin trait T_0 of f, together with those handled exactly by T_0 itself, as provided by variants((parents(T_0), T_0), B). In our example, supvariants(ExtZeroTwo, eval, Exp) expands to variants((Base, ExtTwo), Exp) and returns this/One | this/Two. Crucially, although ExtZero introduces this/Zero, it is correctly excluded from the result, as super.eval cannot handle it without

$$\frac{\begin{array}{c}\textbf{trait}\ \mathsf{T}\ \textbf{requires}\ \overline{\mathsf{U}}\ \{\overline{\mathscr{T}};\ \overline{\textbf{class}\ \textbf{this}/\mathsf{C}(\overline{\mathsf{f_C} : \mathsf{t_C}});}\ \overline{\mathscr{M}}\} \\ \overline{\mathscr{T}} = \overline{\textbf{type}\ \mathsf{A} = \overline{\textbf{this}/\mathsf{C_A}}; \mathsf{B}\ \texttt{+=}\ \overline{\textbf{this}/\mathsf{C_B}}} \quad \overline{\mathscr{M}} = \overline{\mathsf{f} : \mathsf{t_f}}; \overline{\mathsf{g} : \mathsf{t_g} = \mathsf{e_g}} \\ \overline{\vdash \mathsf{U}} \quad \overline{\mathsf{U} \Rightarrow \mathsf{T}} \quad \mathsf{A} \notin \{\ \mathsf{tnames}(\mathsf{U})\ \} \quad \mathsf{B} \in \{\ \mathsf{tnames}(\mathsf{U})\ \} \quad \overline{\mathsf{unique}(\mathsf{B}, \mathsf{T})} \quad \overline{\mathsf{unique}(\textbf{this}/\mathsf{C}, \mathsf{T})} \\ \overline{\textbf{this}/\mathsf{A} \cdot \textbf{now}/\mathsf{A} \cdot \textbf{this}/\mathsf{B} \cdot \textbf{now}/\mathsf{B} \cdot \mathsf{cnames}(\mathsf{T}) \vdash \mathsf{t_f}, \mathsf{t_g}} \\ \overline{\textbf{this}/\mathsf{A} \cdot \textbf{this}/\mathsf{B} \cdot \mathsf{cnames}(\mathsf{T}) \vdash \mathsf{t_C}} \quad \{\ \overline{\textbf{this}/\mathsf{C_A} \cdot \textbf{this}/\mathsf{C_B}}\ \} \subseteq \mathsf{cnames}(\mathsf{T}) \\ \overline{\epsilon \vdash_{\mathsf{T}/\mathsf{g}} \mathsf{e_g} : \mathsf{t_g}} \quad \forall \mathsf{f}_i \in \mathsf{mnames}(\mathsf{T}).\ \mathsf{mvalid}(\mathsf{f}_i, \mathsf{T}) \wedge \mathsf{unique}(\mathsf{f}_i, \mathsf{T})\end{array}}{\vdash \mathsf{T}}$$

$$\frac{\begin{array}{c}\textbf{class}\ \mathsf{prog}/\mathsf{C}(\overline{\mathsf{f} : \mathsf{t}})\ \textbf{extends}\ \overline{\mathsf{T}} \\ \overline{\vdash \mathsf{T}} \quad \overline{\mathsf{T} \Rightarrow \mathsf{prog}/\mathsf{C}} \quad \overline{\epsilon \vdash \mathsf{t}} \quad \forall \mathsf{A} \in \mathsf{tnames}(\mathsf{C}).\ \mathsf{unique}(\mathsf{A}, \mathsf{prog}/\mathsf{C}) \\ \forall \textbf{this}/\mathsf{C}' \in \mathsf{cnames}(\mathsf{prog}/\mathsf{C}).\ \mathsf{unique}(\textbf{this}/\mathsf{C}', \mathsf{prog}/\mathsf{C}) \\ \forall \mathsf{f} \in \mathsf{mnames}(\mathsf{prog}/\mathsf{C}).\ \mathsf{mvalid}(\mathsf{f}, \mathsf{prog}/\mathsf{C}) \wedge \mathsf{unique}(\mathsf{f}, \mathsf{prog}/\mathsf{C}) \\ \forall \mathsf{T} \in \overline{\mathsf{T}}.\ \forall \mathsf{f} \in \mathsf{mnames}(\mathsf{T}).\ \mathsf{supervalid}(\mathsf{f}, \mathsf{T}, \mathsf{prog}/\mathsf{C}) \vee \mathsf{super}(\mathsf{f}, \mathsf{T}) = \varnothing\end{array}}{\vdash \mathsf{prog}/\mathsf{C}}$$

Fig. 4. Top-level definition typing.

modifying `ExtZero` itself. Allowing `ExtZeroTwo` to be used directly to form a class would be unsound, as `eval` would have a type that allows `Zero`. Later, we will see how our type system requires retroactive handling of `Zero`, so that `ExtZeroTwo` can be used to form a valid class.

3.5 Trait and Class Typing

Figure 4 presents the typing rules for top-level trait and class definitions. Specifically, the method validity checks ensure that the extensible variant types in the trait composition of a class are consistent with the assumptions made by each trait when typing the trait methods to maintain soundness.

Trait typing. Judgment $\vdash \mathsf{T}$ states that trait T is well-typed. The trait definition and its contents are used as premises. We first recursively check that all super traits are well-typed. Judgments $\overline{\mathsf{U} \Rightarrow \mathsf{T}}$ validate that each super trait U of T has its own required super traits appearing before U in $\overline{\mathsf{U}}$, so that its super trait requirement is satisfied. Additionally, each trait can only appear once in $\overline{\mathsf{U}}$, and all trait requirement relationships are implicitly assumed to be acyclic. The following four premises check that base extensible variant type and inner class definitions are not duplicated in any trait hierarchy, and that all extensions have their base definitions in the super traits. tnames returns all extensible variant types defined in a top-level definition. $\overline{\mathsf{unique}(\mathsf{B}, \mathsf{T})}$ checks that extensible variant type extensions have unique base definitions in the trait.

Then, we check that all method signatures only involve extensible variant types defined or extended in the current trait, as well as the inner classes in scope as returned by $\mathsf{cnames}(\mathsf{T})$. This is formulated as type well-formedness conditions. Note that **super** never appears in the path prefixes of method signatures. Additionally, **now** does not appear in the path prefixes of inner class field types, and only inner classes in scope can be introduced to extensible variant types.

Finally, the rule type checks each method body against its signature, under an empty term context and with the corresponding trait name passed in. Then we perform method validity checks for all methods in the trait returned by $\mathsf{mnames}(\mathsf{T})$, and make sure each method has no more than one base implementation, which is formulated as the unique condition. This avoids conflicts of base method implementations, as well as accidental overriding of unrelated methods with the same name [24].

$$\frac{\forall \mathtt{U} \in \mathsf{parents}(\mathtt{T}).\ \mathsf{mtype}_{\mathtt{U}}(\mathtt{f},\mathtt{U}) = \mathtt{t} \implies \mathtt{T}/\mathtt{f} \vdash \mathsf{mtype}_{\mathtt{T}}(\mathtt{f},\mathtt{T}) \le \mathtt{t} \quad \mathsf{mimpl}(\mathtt{f},\mathtt{T}) = \varnothing \vee (\mathsf{mimpl}(\mathtt{f},\mathtt{T}) = (\mathtt{t}_0,\mathtt{T}_0) \wedge (\mathtt{T}_0 \neq \mathtt{T} \implies \mathtt{T}/\mathtt{f} \vdash \mathtt{t}_0\,[\mathtt{now} \mapsto \mathtt{super}.\mathtt{f}] \le \mathsf{mtype}_{\mathtt{T}}(\mathtt{f},\mathtt{T})))}{\mathsf{mvalid}(\mathtt{f},\mathtt{T})}$$

$$\frac{\mathsf{mimpl}(\mathtt{f},\mathsf{prog}/\mathtt{C}) = (\mathtt{t}_0,\mathsf{prog}/\mathtt{C}) \quad \forall \mathtt{U} \in \mathsf{parents}(\mathsf{prog}/\mathtt{C}).\ \mathsf{mtype}_{\mathtt{U}}(\mathtt{f},\mathtt{U}) = \mathtt{t} \implies \epsilon \vdash \mathsf{mtype}_{\epsilon}(\mathtt{f},\mathsf{prog}/\mathtt{C}) \le \mathtt{t}\,[\mathtt{now} \mapsto \mathtt{C}][\mathtt{this} \mapsto \mathtt{C}]}{\mathsf{mvalid}(\mathtt{f},\mathsf{prog}/\mathtt{C})}$$

$$\frac{\mathsf{mimpl}(\mathtt{f},\mathsf{prog}/\mathtt{C}) = (\mathtt{t}_0,\mathtt{T}) \quad \mathsf{tnames}(\mathtt{T}) = \overline{\mathtt{A}} \quad \theta = [\overline{\mathtt{now}/\mathtt{A} \mapsto \mathsf{nowvariants}(\mathsf{prog}/\mathtt{C},\mathtt{T},\mathtt{f},\mathtt{A})}] \quad \epsilon \vdash \mathtt{t}_0\theta\,[\mathtt{this} \mapsto \mathtt{C}] \le \mathsf{mtype}_{\epsilon}(\mathtt{f},\mathsf{prog}/\mathtt{C}) \quad \forall \mathtt{U} \in \mathsf{parents}(\mathsf{prog}/\mathtt{C}).\ \mathsf{mtype}_{\mathtt{U}}(\mathtt{f},\mathtt{U}) = \mathtt{t} \implies \epsilon \vdash \mathsf{mtype}_{\epsilon}(\mathtt{f},\mathsf{prog}/\mathtt{C}) \le \mathtt{t}\,[\mathtt{now} \mapsto \mathtt{C}][\mathtt{this} \mapsto \mathtt{C}]}{\mathsf{mvalid}(\mathtt{f},\mathsf{prog}/\mathtt{C})}$$

$$\frac{\mathsf{parents}(\mathsf{prog}/\mathtt{C}) = (\overline{\mathtt{T}_i},\mathtt{T},\ldots) \quad \mathsf{super}(\mathtt{f},\mathtt{T}) = \mathtt{t} \quad \mathsf{tnames}(\mathtt{T}) = \overline{\mathtt{A}} \quad \mathsf{mnames}(\mathtt{T}) = \overline{\mathtt{g}} \quad \theta = [\overline{\mathtt{super}.\mathtt{g}/\mathtt{A} \mapsto \mathsf{supvariants}(\mathsf{prog}/\mathtt{C},\mathtt{T},\mathtt{g},\mathtt{A})}] \quad \mathsf{searchimpl}(\mathtt{f},\overline{\mathtt{T}_i}) = (\mathtt{t}_0,\mathtt{U}) \quad \mathsf{tnames}(\mathtt{U}) = \overline{\mathtt{B}} \quad \theta_0 = [\overline{\mathtt{now}/\mathtt{B} \mapsto \mathsf{nowvariants}(\mathsf{prog}/\mathtt{C},\mathtt{U},\mathtt{f},\mathtt{B})}] \quad \epsilon \vdash \mathtt{t}_0\theta_0\,[\mathtt{this} \mapsto \mathtt{C}] \le \mathtt{t}\theta\,[\mathtt{this} \mapsto \mathtt{C}]}{\mathsf{supervalid}(\mathtt{f},\mathtt{T},\mathsf{prog}/\mathtt{C})}$$

$$\mathsf{nowvariants}(\mathsf{prog}/\mathtt{C},\mathtt{T},\mathtt{f},\mathtt{B}) = \begin{cases} \mathsf{supvariants}(\mathsf{prog}/\mathtt{C},\mathtt{T},\mathtt{f},\mathtt{B})\ \overline{\mid \mathtt{this}/\mathtt{C}_{\mathtt{B}}} & \text{if } (\mathtt{B} \mathrel{+}= \overline{\mathtt{this}/\mathtt{C}_{\mathtt{B}}}) \in \mathtt{T} \\ \mathtt{Nothing}\ \overline{\mid \mathtt{this}/\mathtt{C}_{\mathtt{B}}} & \text{if } (\mathbf{type}\ \mathtt{B} = \overline{\mathtt{this}/\mathtt{C}_{\mathtt{B}}}) \in \mathtt{T} \end{cases}$$

$$\mathsf{supvariants}(\mathsf{prog}/\mathtt{C},\mathtt{T},\mathtt{f},\mathtt{B}) = \begin{cases} \mathsf{variants}((\mathsf{parents}(\mathtt{T}_0),\mathtt{T}_0,\overline{\mathtt{T}_i}),\mathtt{B}) & \text{if } \mathsf{origin}(\mathtt{f},\mathtt{T}) = \mathtt{T}_0 \\ & \text{and } \mathsf{parents}(\mathsf{prog}/\mathtt{C}) = \ldots,\mathtt{T}_0,\overline{\mathtt{T}_i},\mathtt{T},\ldots \\ \mathsf{variants}(\mathsf{parents}(\mathtt{T}),\mathtt{B}) & \text{if } \mathsf{origin}(\mathtt{f},\mathtt{T}) = \mathtt{T} \end{cases}$$

Fig. 5. Method validity checks.

Class typing. Judgment $\vdash \mathsf{prog}/\mathtt{C}$ states that top-level class $\mathsf{prog}/\mathtt{C}$ is well-typed. The rule first validates that all traits composed are well-typed and that the trait composition satisfies the super trait requirements of each trait. For each variant type name in the class, i.e. in $\mathsf{tnames}(\mathtt{C})$, the rule checks that it has a unique base definition. Each inner class name $\mathtt{C}'$ in $\mathsf{cnames}(\mathtt{C})$, which returns all inner class names in the composed traits, should also be uniquely defined. Lastly, we perform method validity checks for all methods of the class, and require every method to have a unique base implementation. For each inherited trait, the assumptions on the overridden implementation of each method should be satisfied, which is formulated as the supervalid condition; otherwise, the method should be a base implementation with empty $\mathsf{super}(\mathtt{f},\mathtt{T})$.

Method validity checks. Judgment $\mathsf{mvalid}(\mathtt{f},\mathscr{N})$ defined in Figure 5 states that method $\mathtt{f}$ is valid in a class or trait $\mathscr{N}$. These checks ensure that method signatures respect the nominal subtyping relationships between traits and between a class and its parent traits. Also, the assumptions on the extensible variant types in traits should be consistent with the concrete variants as determined by the class composition.

For trait $\mathtt{T}$, we check that the method signature of $\mathtt{f}$ in $\mathtt{T}$ is a subtype of its signature in every parent trait where it is declared. Those subtyping checks are performed under the context $\mathtt{T}/\mathtt{f}$, which provides the appropriate subtyping assumptions on extensible variant types for method $\mathtt{f}$ in $\mathtt{T}$, as established by the subtyping rules in Figure 3.

The second premise checks that if method $\mathtt{f}$ is only implemented in a super trait $\mathtt{T}_0$, then the implementation's type $\mathtt{t}_0$ should conform to $\mathtt{f}$'s signature in $\mathtt{T}$, after adapting $\mathtt{now}/$ in $\mathtt{t}_0$ to

`super.f/`. mimpl(`f`, `T`) simply expands to searchimpl(`f`, (parents(`T`), `T`)). When a trait extends an extensible variant type with new variants without providing overriding implementations of the related methods, this condition handles the case where the inherited implementation may not be compatible after the variant type is extended. It is greyed out because it is not essential for type soundness: the later class-level checks ensure that inherited implementations conform to the finalized class signatures. It nonetheless serves a useful software engineering purpose, localizing the error to the trait rather than to every class using it. This condition rules out the problematic `ScaleMyRegions` trait where `show` was not overridden to handle the new variant `Scale`, as discussed in Section 2.3.

For class `prog/C`, there are two rules that distinguish two cases depending on where the implementation of `f` resides as decided by mimpl(`f`, `prog/C`). mimpl first looks up the class fields, then searches the parent traits with searchimpl. When `f` is directly implemented by `prog/C` itself as a class field, we check that `f`'s signature in the class is compatible with the signature inherited from every parent trait. Since variant types are finalized in the class, all **`now`** and **`this`** path prefixes in the inherited signatures are substituted with the concrete class name `C`.

When `f` is implemented by a super trait `T`, a subtyping check in the third mvalid rule ensures the implementation's type t_0 conforms to the class's finalized method signature. As t_0 may contain extensible variant types with the **`now`** path prefix we need to resolve them to the actual variants determined by the trait composition. This resolution is achieved by a substitution θ, which maps each **`now`**`/A` in `T` to the concrete variants returned by nowvariants(`prog/C`, `T`, `f`, `A`). After applying θ and concretizing **`this`** to `C`, the implementation's type is checked to be a subtype of the class method signature.

Auxiliary function nowvariants(`prog/C`, `T`, `f`, `B`) computes the concrete set of variants that **`now`**`/B` represents in the context of trait `T` mixed into class `prog/C`. When `B` is extended in `T` with new variants via `+=`, the result combines the variants from the super traits supvariants with the newly introduced variants. When `B` is newly defined in `T` with `=`, the result is simply the union of the variants in its definition. Function supvariants(`prog/C`, `T`, `f`, `B`) is analogous to the supvariants defined for subtyping in Figure 3, returning the union of `B` variants that, under `T`'s assumptions, the overridden implementation of `f` should handle. The key difference is that it uses the full trait composition parents(`prog/C`) rather than parents(`T`), since which variants sit between the origin of `f` and `T` is only known once the class is formed.

Returning to the example in Section 3.4, we can now see precisely why `ExtZeroTwo` can never be composed to form a valid class as is, without following the retroactive handling pattern in Section 2.4. Since `ExtZero` sits *before* `ExtTwo` and is not required by `ExtTwo`, `Zero` is not included in the actual variants of **`now`**`/Exp` as resolved by nowvariants. Suppose we have a class `C` that extends `ExtZeroTwo`. The subtyping check fails because **`now`**`/Exp` $\longrightarrow$ `Int`, after being substituted to (`C/One` | `C/Two`) $\longrightarrow$ `Int`, is not a subtype of the class's finalized signature, which is (`C/Zero` | `C/One` | `C/Two`) $\longrightarrow$ `Int`.

Super method validity. Judgment supervalid(`f`, `T`, `prog/C`) validates that the assumptions on super method calls made by trait `T` are satisfied in the context of the class `prog/C`[4]. When a trait method calls **`super`**`.f`, it relies on the assumption that the overridden implementation handles certain variants of the extensible variant types, as recorded by the subtyping constraints on the types with the **`super`**`.f` path prefix in the type of **`super`**`.f`. The supervalid check ensures that the actual overridden implementation, as determined by the class's trait composition, conforms to this assumption. The rule first retrieves the type of **`super`**`.f` in `T` by calling super(`f`, `T`), and then for every method name `g`, it resolves the **`super`**`.g/A` references to the variants it actually expects to be handled by the overridden implementation in the trait composition of `prog/C`, which is

[4]Scala 3 also performs a similar check for validity of the **`super`** accessor. Related discussions can be found in issue #5433.

$$
\begin{array}{rrcl}
\textit{Class path} & p & ::= & \kappa \mid C \mid \mathit{prog} \\
\textit{Type} & \tau & ::= & I[\overline{\tau}, p] \mid \top \mid \bot \mid \alpha, \beta, \gamma \mid \tau \to \tau \mid \tau \sqcap \tau \mid \tau \sqcup \tau \mid \neg\tau \mid \#p/C \\
\textit{Expression} & e & ::= & x, y \mid \textbf{this} \mid \textbf{super}.f \mid e : \tau \mid e.f \\
 & & & \mid \lambda x : \tau.\, e \mid e\ e \mid \textbf{new}\ p/C(\overline{e}) \mid \textbf{if}\ e\ \textbf{is}\ p/C \lhd \tau\ \textbf{then}\ e\ \textbf{else}\ e \\
\textit{Structural refinement} & \mathcal{R} & ::= & \{\ \overline{f : \tau}\ \} \\
\textit{Type synonym} & \mathcal{T} & ::= & \textbf{type}\ \alpha = \tau \\
\textit{Top-level definition} & \mathcal{D} & ::= & \\
\textit{Interface} & & & \textbf{interface}\ I[\overline{\alpha}, \kappa] \lhd \overline{J[\overline{\tau}, p]}\ \{\ \overline{\mathcal{T}}\ \overline{f : \tau}\ \} \\
\textit{Mixin} & & \mid & \textbf{mixin}\ M[\overline{\alpha}, \kappa]^{\mathcal{R}}_{\tau}\ \{\ \overline{\mathcal{T}}\ \overline{f : \tau = e}\ \} \\
\textit{Class} & & \mid & \textbf{class}\ p/C(\overline{f : \tau}) \lhd I[\overline{\tau}, p], \overline{M[\overline{\tau}, p]} \\
\textit{Program} & \mathcal{P} & ::= & \overline{\mathcal{D}}\,;\, \overline{\mathcal{T}}\,;\, e \\
\textit{Typing context} & \Gamma & ::= & \epsilon \mid \Gamma \cdot (x : \tau) \mid \Gamma \cdot (\textbf{this} : \tau) \mid \Gamma \cdot (\textbf{super} : \mathcal{R}) \mid \Gamma \cdot (\tau \le \tau) \mid \Gamma \cdot \rhd(\tau \le \tau) \\
\textit{Elaboration context} & \Xi & ::= & \epsilon \mid \mathsf{T} \mid \mathsf{T/f} \qquad \xi ::= \epsilon \mid \xi \cdot (\alpha\ \mathsf{of}\ \mathsf{T} \mapsto \tau)
\end{array}
$$

Fig. 6. Syntax of the core language.

computed by supvariants(prog/C, T, g, A). Then, it finds the actual overridden implementation the super call is dispatched to by applying searchimpl to the traits composed before T, which returns the implementation's type $\mathtt{t}_0$ and the trait U that provides it. The types with the **now** path prefix in $\mathtt{t}_0$ are again resolved by nowvariants. Finally, the rule checks that the resolved implementation's type is a subtype of the resolved type of **super**.f, with **this** concretized to C on both sides.

4 Elaboration

This section introduces the syntax of the core language adapted from the SuperOOP core calculus [12], and the elaboration process from the surface language of traits to the core language, which provides semantics and establishes soundness for the surface language. The full definitions of the core language can be found in the appendix.

4.1 The Core Language

Figure 6 presents the syntax of the core language. Paths p include a class name, the top-level prog prefix, or a *path variable* κ that abstractly represents the enclosing class during trait elaboration and is concretized when the trait is composed into a class, corresponding to the **this** path prefix of inner classes in the surface language.

Types include interfaces parameterized by type arguments and a path, type variables, class tags $\#p/C$ that express the nominal identity of classes, and the Boolean-algebraic types. Expressions have constructs similar to those in the surface language. Objects are constructed from classes and typed as the intersection of the class tag and the interface the class implements. For conditional expressions, the scrutinee is matched against a class tag, with a type annotation representing the interface the class conforms to. A type synonym equates a type variable to a type, where the type variable itself can appear in the type *under a function or interface type constructor*, as required by the guardedness checks of equi-recursive types in Boolean-algebraic subtyping [34].

Interfaces I, parameterized by type variables and a path variable, extend other interfaces J. Mixins contain method implementations. Crucially, mixins have *precise types* for the **super**.f calls and **this** references in the methods, provided as $\mathcal{R}$ and τ annotations respectively in the mixin definition. Unlike traits, mixins do *not* introduce types or subtyping relationships. Classes have a list of field

declarations, implement an interface, and extend multiple mixins. The safety of open-recursive method calls is ensured by checking if the class and its mixin composition satisfy the precise types of **this** and **super** in each mixin.

Typing context Γ tracks subtyping hypotheses that may be under a *"later" modality* $\triangleright$ to delay the applicability of subtyping hypotheses, enabling subtyping of equi-recursive types [5, 17, 34]. Elaboration context Ξ, used for trait elaboration and for type and term translation, identifies the current trait or trait method being elaborated. For class elaboration, elaboration context ξ maps type variables of each trait to their concrete instantiations.

4.2 Trait Elaboration

Below we present the elaboration of the following surface language trait definition. We denote method names returned by $\mathsf{mnames}(\mathtt{T})$ as $\overline{\mathtt{f}}$, which include all methods declared in the entire trait hierarchy. $[\![\cdot]\!]_{\Xi,\kappa?}$ translates the surface language types and terms with elaboration context Ξ and an optional path variable κ; its full definition is provided in the appendix.

$$\textbf{trait}\ \mathtt{T}\ \textbf{requires}\ \overline{\mathtt{U}}\ \{\overline{\textbf{type}\ \mathtt{A} = \overline{\textbf{this}/\mathtt{C_A}}};\ \overline{\mathtt{B}\ \texttt{+=}\ \overline{\textbf{this}/\mathtt{C_B}}};\ \overline{\textbf{class}\ \textbf{this}/\mathtt{C}(\overline{\mathtt{f_C} : \mathtt{t}})};\ \overline{\mathtt{g'} : \mathtt{t_{g'}}};\ \overline{\mathtt{g} : \mathtt{t_g} = \mathtt{e_g}};\ \}$$

The interface of the trait, "as usual". The following interface stands for the type that the trait introduces *as usual*:

$$\textbf{interface}\ I_{\mathtt{T}} \lhd \overline{I_{\mathtt{U}}}\ \{\ \overline{g' : [\![\mathtt{t_{g'}}]\!]_\epsilon};\ \overline{g : [\![\mathtt{t_g}]\!]_\epsilon}\ \text{where}\ \epsilon \vdash \mathtt{t_{g'}}\ \text{and}\ \epsilon \vdash \mathtt{t_g}\}$$

Interface $I_{\mathtt{T}}$ extends the usual interfaces of the super traits $I_{\mathtt{U}}$ and contains only the method signatures whose types do *not* mention any extensible variant types, as restricted by the type well-formedness conditions $\epsilon \vdash \mathtt{t}$. This is a direct reflection of the fact that when a trait is used in a general setting, it is not aware of the constraints of the extensible variant types in the class inheriting it, as also captured by rule T-Method for the surface language typing. As a consequence, any reference to the trait type in the surface language in general is translated to this "usual" interface of the trait.

***The interface of the trait, as the type of* this.** Recall that by rule T-ThisMethod of the surface language typing, one may call methods involving extensible variant types when the receiver is **this**, and the called method signature has its **now** path prefixes substituted with **this**. This corresponds to the core language interface $I_{\mathtt{T\$this}}$ to be used *as the precise type of* **this** in the trait mixin:

$$\begin{aligned}&\textbf{interface}\ I_{\mathtt{T\$this}}[\overline{\alpha_{\mathtt{A}}}, \overline{\alpha_{\mathtt{B}}}, \overline{\beta_{\mathtt{B\$f}}}, \kappa] \lhd I_{\mathtt{T}}\ \{\ \overline{\mathcal{T}_{\mathtt{A1}}};\ \overline{\mathcal{T}_{\mathtt{A2}}};\ \overline{\mathcal{T}_{\mathtt{B0}}};\ \overline{\mathcal{T}_{\mathtt{B1}}};\ \overline{\mathcal{T}_{\mathtt{B2}}};\\ &\quad\overline{C' : \overline{[\![\mathtt{t_0}]\!]_{\mathtt{T},\kappa} \to [\![\textbf{this}/\mathtt{C'}]\!]_{\mathtt{T},\kappa}}};\ \text{where}\ \mathsf{cnames}(\mathtt{T}) = \overline{\textbf{this}/\mathtt{C'}}\ \text{and}\ \overline{\mathsf{ctor}_{\mathtt{T}}(\textbf{this}/\mathtt{C'}) = \overline{\mathtt{f_0} : \mathtt{t_0}}}\\ &\quad\overline{f : [\![\mathtt{t}[\textbf{now} \mapsto \textbf{this}]]\!]_{\mathtt{T}/\mathtt{f},\kappa}}\ \text{where}\ \mathsf{mnames}(\mathtt{T}) = \overline{\mathtt{f}}\ \text{and}\ \overline{\mathsf{mtype}_{\mathtt{T}}(\mathtt{f}, \mathtt{T}) = \mathtt{t}}\}\end{aligned}$$

$I_{\mathtt{T\$this}}$ extends the usual interface $I_{\mathtt{T}}$, so **this** can be upcast to the usual trait interface type. Crucially, interface $I_{\mathtt{T\$this}}$ is parameterized by type variables and a path variable: $\overline{\alpha_{\mathtt{A}}}$, for the *future extensions* to the extensible variant types $\overline{\mathtt{A}}$ *newly defined* in the current trait; $\overline{\alpha_{\mathtt{B}}}$, for the future extensions to the extensible variant types $\overline{\mathtt{B}}$ *extended* in the current trait; $\overline{\beta_{\mathtt{B\$f}}}$, for the *unknown base variants* of the inherited extensible variant types, one for each extended type $\mathtt{B}$ and method $\mathtt{f}$. They are method-specific because each method may have different origins and thus different views of which base variants are expected to be handled. Finally, κ is the path variable to be instantiated to the class it is composed into. The interface first defines five series of type synonyms for the different interpretations of the extensible variant types in the trait. With those in scope, we translate into the core language the signatures of all methods $\overline{\mathtt{f}}$ in $\mathsf{mnames}(\mathtt{T})$, and the inner class declarations $\overline{\textbf{this}/\mathtt{C'}}$ in $\mathsf{cnames}(\mathtt{T})$ as constructor methods. Their references to the extensible variant types have the **now** path prefixes substituted with **this**, reflecting that $I_{\mathtt{T\$this}}$ is the type of **this** in the trait.

Now we define the type synonyms that directly encode the subtyping assumptions established by the rules in Figure 3:

$$\mathcal{T}_{\mathsf{A}1} := (\textbf{type}\ \alpha_{\mathsf{A\$f\$now}} = \bot \sqcup \overline{[\![\ \texttt{this}/\mathsf{C_A}\ ]\!]_{\mathsf{T},\kappa}}) \quad \text{for each } \mathtt{f} \in \mathsf{mnames}(\mathsf{T})$$

$$\mathcal{T}_{\mathsf{A}2} := (\textbf{type}\ \alpha_{\mathsf{A\$this}} = (\alpha_{\mathsf{A}} \sqcap \mathsf{Object}) \sqcup \overline{[\![\ \texttt{this}/\mathsf{C_A}\ ]\!]_{\mathsf{T},\kappa}})$$

$$\mathcal{T}_{\mathsf{B}0} := \begin{cases} (\textbf{type}\ \alpha_{\mathsf{B\$f\$super}} = (\beta_{\mathsf{B\$f}} \sqcap \mathsf{Object}) \sqcup [\![\ \mathsf{supvariants}(\mathsf{T},\mathtt{f},\mathsf{B})\ ]\!]_{\mathsf{T},\kappa} & \text{if } \mathsf{origin}(\mathtt{f},\mathsf{T}) \neq \mathsf{T} \\ (\textbf{type}\ \alpha_{\mathsf{B\$f\$super}} = [\![\ \mathsf{supvariants}(\mathsf{T},\mathtt{f},\mathsf{B})\ ]\!]_{\mathsf{T},\kappa} & \text{if } \mathsf{origin}(\mathtt{f},\mathsf{T}) = \mathsf{T} \end{cases}$$
$$\text{for each } \mathtt{f} \in \mathsf{mnames}(\mathsf{T})$$

$$\mathcal{T}_{\mathsf{B}1} := (\textbf{type}\ \alpha_{\mathsf{B\$f\$now}} = \alpha_{\mathsf{B\$f\$super}} \sqcup \overline{[\![\ \texttt{this}/\mathsf{C_B}\ ]\!]_{\mathsf{T},\kappa}}) \quad \text{for each } \mathtt{f} \in \mathsf{mnames}(\mathsf{T})$$

$$\mathcal{T}_{\mathsf{B}2} := (\textbf{type}\ \alpha_{\mathsf{B\$this}} = (\alpha_{\mathsf{B}} \sqcap \mathsf{Object}) \sqcup [\![\ \mathsf{allvariants}(\mathsf{T},\mathsf{B})\ ]\!]_{\mathsf{T},\kappa} \overline{\sqcup\ (\beta_{\mathsf{B\$f}} \sqcap \mathsf{Object})})$$
$$\text{where } \mathtt{f} \in \mathsf{mnames}(\mathsf{T}) \text{ and } \mathsf{origin}(\mathtt{f},\mathsf{T}) \neq \mathsf{T}$$

$\mathcal{T}_{\mathsf{A}1}$ defines $\alpha_{\mathsf{A\$f\$now}}$, one for each method $\mathtt{f}$, as the union of the variants of $\mathtt{A}$ defined in the current trait, encoding $\texttt{now}/\mathtt{A}$ in the surface language. $\mathcal{T}_{\mathsf{A}2}$ encodes $\texttt{this}/\mathtt{A}$ as the union of the known variants and the future extensions α_{A}. $\alpha_{\mathsf{B\$f\$super}}$ represents $\texttt{super}.\mathtt{f}/\mathtt{B}$ for each method $\mathtt{f}$: when $\mathtt{f}$ originates in T, it is equal to the known super variants as computed by supvariants; otherwise, the unknown base variants $\beta_{\mathsf{B\$f}}$ are included. $\alpha_{\mathsf{B\$f\$now}}$ represents $\texttt{now}/\mathtt{B}$; it is the union of $\alpha_{\mathsf{B\$f\$super}}$ and the newly introduced variants. $\alpha_{\mathsf{B\$this}}$ stands for $\texttt{this}/\mathtt{B}$; it is the union of all known variants, the unknown base variants of all overriding methods $\overline{\beta_{\mathsf{B\$f}}}$, and the future extensions α_{B}.

The inner classes. When elaborating the trait, we generate an interface $I_{\mathsf{T\$C}}$ that includes the field types for each inner class in the trait, with only type parameters for the future extensions and the type synonyms for the $\texttt{this}$ path prefix, as inner class fields do not mention $\texttt{now}/$. Type synonyms $\overline{\mathcal{T}'_{\mathsf{B}2}}$ are similar to $\overline{\mathcal{T}_{\mathsf{B}2}}$ but without summing over the unknown base variants. During class elaboration, the inner class will be concretized as a top-level class with its path parameter properly instantiated.

$$\textbf{interface}\ I_{\mathsf{T\$C}}[\overline{\alpha_{\mathsf{A}}}, \overline{\alpha_{\mathsf{B}}}, \kappa]\ \{\ \overline{\mathcal{T}_{\mathsf{A}2}};\ \overline{\mathcal{T}'_{\mathsf{B}2}};\ \overline{f_{\mathsf{C}} : [\![\ \mathtt{t}\ ]\!]_{\mathsf{T},\kappa}};\}\ \text{for each } \mathtt{C} \in \overline{\texttt{this}/\mathtt{C}}$$

The mixin. Finally, we translate the trait into a mixin M_{T} that contains the method bodies translated into the core language:

$$\textbf{mixin}\ M_{\mathsf{T}}[\overline{\alpha_{\mathsf{A}}}, \overline{\alpha_{\mathsf{B}}}, \overline{\beta_{\mathsf{B\$f}}}, \kappa]^{\mathcal{R}}_{\tau}\{\ \overline{\mathcal{T}_{\mathsf{A}1}};\ \overline{\mathcal{T}_{\mathsf{A}2}};\ \overline{\mathcal{T}_{\mathsf{B}0}};\ \overline{\mathcal{T}_{\mathsf{B}1}};\ \overline{\mathcal{T}_{\mathsf{B}2}};\quad \overline{g : [\![\ \mathtt{t_g}\ ]\!]_{\mathsf{T/g},\kappa} = [\![\ \mathtt{e_g}\ ]\!]_{\mathsf{T/g},\kappa}};\}$$

$$\tau := I_{\mathsf{T\$this}}[\overline{\alpha_{\mathsf{A}}}, \overline{\alpha_{\mathsf{B}}}, \overline{\beta_{\mathsf{B\$f}}}, \kappa] \qquad \mathcal{R} := \{\ \overline{f : [\![\ \mathtt{t}\ ]\!]_{\mathsf{T/f},\kappa}}\ \} \text{ where } \overline{\mathtt{f}} = \mathsf{mnames}(\mathsf{T}) \text{ and } \overline{\mathsf{super}(\mathtt{f},\mathsf{T}) = \mathtt{t}}$$

Mixin M_{T} is also parameterized by the same type and path parameters and provides the type synonyms for the extensible variant types. Crucially, we ascribe precise types for **this** and **super** in the mixin, namely interface $I_{\mathsf{T\$this}}$ for **this**, and a structural refinement $\mathcal{R}$ which includes all the overridden method signatures callable via **super**.

4.3 Class Elaboration

Figure 7 presents the elaboration of surface class definitions. Class elaboration finalizes all extensible variant types and concretizes inner classes defined in the trait composition.

Finalizing variant types. For each extensible variant type $\mathtt{A}$ in the class, the elaboration creates a type synonym $\alpha_{\mathsf{C\$A}}$ at the program level that is equal to the union of all variants collected from the trait hierarchy via $\mathsf{allvariants}(\mathtt{prog}/\mathtt{C}, \mathtt{A})$, as the finalized version of $\mathtt{A}$ in top-level class $\mathtt{prog}/\mathtt{C}$, after concretizing the $\texttt{this}$ path prefix to $\mathtt{C}$.

Source language class `class prog/C(`$\overline{\texttt{f}_0 : \texttt{t}}$`) extends` $\overline{\texttt{T}}$

type $\alpha_{\texttt{C\$A}} = [\![\texttt{allvariants(prog/C, A)[this} \mapsto \texttt{C]}]\!]_\epsilon$
for each $\texttt{A} \in \text{tnames}(\texttt{C})$

class $C/C'(\overline{f : [\![\texttt{t[this} \mapsto \texttt{C]}]\!]_\epsilon}) \lhd I_{\texttt{T'\$C'}}[\overline{\alpha[\xi_\texttt{C} \text{ of } \texttt{T'}]}, C]$

mixin $M_{\texttt{C\$C'}}\,^{\{\}}_{\texttt{T}}\,\{\, C'(\overline{x : [\![\texttt{t[this} \mapsto \texttt{C]}]\!]_\epsilon}) : \alpha_{\texttt{C\$C'}} = \textbf{new}\ C/C'(\overline{x})\ \}$

type $\alpha_{\texttt{C\$C'}} = \#C/C' \sqcap I_{\texttt{T'\$C'}}[\overline{\alpha[\xi_\texttt{C} \text{ of } \texttt{T'}]}, C]$
for each $\texttt{C'} \in \text{cnames}(\texttt{C})$
where $\text{origin}(\texttt{C'}, \texttt{C}) = \texttt{T'}$ and **interface** $I_{\texttt{T'\$C'}}[\overline{\alpha}, \kappa]$ is defined and $\text{ctor}_{\texttt{T'}}(\texttt{C'}) = \overline{\texttt{f : t}}$

interface $I_{\texttt{prog/C}} \lhd \overline{I_{\texttt{T\$this}}[\overline{\alpha[\xi_\texttt{C} \text{ of } \texttt{T}]}, C]}\ \{\ \overline{f_0 : [\![\texttt{t}]\!]_\epsilon}\ \}$
where **interface** $I_{\texttt{T\$this}}[\overline{\alpha}, \kappa]$ is defined

class $prog/C(\overline{f_0 : [\![\texttt{t}]\!]_\epsilon}) \lhd I_{\texttt{prog/C}}, \overline{M_{\texttt{C\$C'}}}, \overline{M_\texttt{T}[\overline{\alpha[\xi_\texttt{C} \text{ of } \texttt{T}]}, C]}$
where $\text{cnames}(\texttt{C}) = \overline{\texttt{C'}}$ and **mixin** $M_\texttt{T}[\overline{\alpha}, \kappa]$ is defined

$\xi_\texttt{C} := \overline{(\alpha_\texttt{A} \text{ of } \texttt{T} \mapsto \alpha_{\texttt{C\$A}}) \cdot (\beta_{\texttt{A\$f}} \text{ of } \texttt{T} \mapsto [\![\texttt{supvariants(prog/C, T, f, A)[this} \mapsto \texttt{C]}]\!]_\epsilon)}$
for each $\texttt{T} \in \text{parents}(\texttt{prog/C})$ and $\texttt{f} \in \text{mnames}(\texttt{T})$ and $\texttt{A} \in \text{tnames}(\texttt{C})$

$[\xi_\texttt{C} \text{ of } \texttt{T}] := [\overline{\alpha \mapsto \tau}]$
where $\overline{(\alpha \text{ of } \texttt{T} \mapsto \tau) \in \xi_\texttt{C}}$

Fig. 7. Elaboration of class definitions.

Inner class generation. For each inner class $\texttt{C'}$ defined in the trait hierarchy of `prog/C` as computed by $\text{cnames}(\texttt{C})$, the elaboration generates a concrete class C/C' at the top level with fields $\overline{f : [\![\texttt{t[this} \mapsto \texttt{C]}]\!]_\epsilon}$ that implements the interface $I_{\texttt{T'\$C'}}$, where $\texttt{T'}$ is the originating trait that defines the inner class with class fields $\overline{\texttt{f : t}}$, determined by $\text{origin}(\texttt{C'}, \texttt{C})$ and $\text{ctor}_{\texttt{T'}}(\texttt{C'})$, and the type parameters of the interface are instantiated by applying the substitutions for $\texttt{T'}$ in the class elaboration context, written as $\overline{\alpha[\xi_\texttt{C} \text{ of } \texttt{T'}]}$. The path parameter κ is instantiated to the top-level class name C. The path prefix **this** in field types is substituted with `C`, reflecting that the **this** path prefix of variant types and inner class types now refers to the concrete class `prog/C`. Note that all types are translated into the core language under an empty elaboration context. The elaboration then creates a constructor mixin $M_{\texttt{C\$C'}}$ that implements the constructor method. The constructor method returns $\alpha_{\texttt{C\$C'}}$, which is a type synonym that intersects the nominal tag with the interface, with its parameters properly instantiated. This synonym stands for the type of instances of this concrete inner class.

The class itself and its interface. Core language class $prog/C$ implements the interface $I_{\texttt{prog/C}}$, which extends all super trait interfaces for the type of **this** in their corresponding mixins with type and path parameters instantiated, and contains the types of class fields in the interface body. For the method implementations, the class extends all generated inner class constructor mixins, and then all mixins of the traits in the *trait composition order*. Note that the requirements of **this** in each mixin are all satisfied, as the class itself conforms to the trait interfaces for **this**.

Type parameter instantiation. Finally, class elaboration context $\xi_\texttt{C}$ maps type parameters of various traits to their concrete instantiations in the class: for each trait `T` in the class's linearization, $\alpha_\texttt{A}$ is mapped to $\alpha_{\texttt{C\$A}}$, the finalized union of all variants of `A` in the class; $\beta_{\texttt{B\$f}}$ is mapped to the concrete super variants as returned by $\text{supvariants}(\texttt{prog/C}, \texttt{T}, \texttt{f}, \texttt{B})$ after type elaboration, with **this**

resolved to C. The path variable κ is instantiated to the class name C. These instantiations supply the concrete variants that the parameterized interfaces and mixins abstract over for the extensibility of variant types.

4.4 Metatheory

We show the theorems that establish the soundness of the surface language. We denote the top-level definitions produced by the elaboration for trait T in Section 4.2 as $\overline{\mathcal{D}_{\mathsf{T}}}$, and the top-level definitions and type synonyms produced by the elaboration for class C in Figure 7 as $\overline{\mathcal{D}_{\mathsf{C}}}$ and $\overline{\mathcal{T}_{\mathsf{C}}}$. For a core language program $\mathcal{P}$, we write $\mathcal{D}_{\mathcal{P}}$ for the top-level definitions and $e_{\mathcal{P}}$ for the top-level expression in $\mathcal{P}$. The elaboration soundness theorem is as follows:

Theorem 4.1 (Elaboration soundness). *Given:*

- *a surface language program* $\mathsf{P} = \overline{\mathsf{T}}, \overline{\mathsf{C}}; \mathsf{e}$
- *a core language program* $\mathcal{P} = \overline{\mathcal{D}_{\mathsf{T}}}, \overline{\mathcal{D}_{\mathsf{C}}}, \overline{\mathcal{T}_{\mathsf{C}}}; [\![\mathsf{e}]\!]_{\epsilon}$ *where* $\mathsf{T} \in \overline{\mathsf{T}}$ *and* $\mathsf{C} \in \overline{\mathsf{C}}$

If $\overline{\vdash \mathsf{T}}$ *and* $\overline{\vdash \mathsf{C}}$ *and* $\epsilon \vdash_{\epsilon} \mathsf{e} : \mathsf{t}$*, then* $\overline{\mathcal{D}_{\mathcal{P}}\ \textbf{\textit{ok}}}$ *and* $\overline{\mathcal{T}_{\mathsf{C}}}; \epsilon \vdash e_{\mathcal{P}} : [\![\mathsf{t}]\!]_{\epsilon}$.

We prove soundness of the core language by adapting the results of Fan and Parreaux [12] to our extensions. In particular, following the approach of Chau and Parreaux [5], we show that the core language subtyping is sound by defining several characteristic *Boolean homomorphisms* from the core language types to various target Boolean algebras. Rules for typing core language top-level definitions $\mathcal{D}\ \textbf{\textit{ok}}$ and expressions $\overline{\mathcal{T}}; \Gamma \vdash e : \tau$, as well as the big-step semantics $E \vdash e \Downarrow r$ (where E is the evaluation context and r is the result), can be found in the appendix.

Theorem 4.2 (Core language soundness). *Given program* $\mathcal{P}$*, if* $\overline{\mathcal{D}_{\mathcal{P}}\ \textbf{\textit{ok}}}$ *and* $\overline{\mathcal{T}}; \epsilon \vdash e_{\mathcal{P}} : \tau$*, then* $\epsilon \vdash e_{\mathcal{P}} \Downarrow \textbf{val}\ v$*, or* $e_{\mathcal{P}}$ *diverges.*

5 Related Work

This section discusses related work on language designs featuring extensible programming.

Scala traits. Scala traits [24, 29] offer a powerful mechanism [30] widely adopted for scalable and modular software design, in particular for modular embedded DSL development. The pattern that inspired our design is that traits include extensible variant types as abstract classes, and the derived traits extend the variant types by adding case classes. Despite its popularity and successful applications [37, 41], this particular pattern is not type-safe, undermining reliable modular programming. Nonetheless, Scala traits and this pattern support crucial features for extensible DSL development, such as method overriding and convenient deep pattern matching with support for separate compilation.

Family polymorphism. Family polymorphism [10] allows for grouping *late-bound* definitions of more than just methods. Nested classes as members of a family can also be late-bound. Most family polymorphism approaches do not support the definition of variants that can be pattern-matched with exhaustiveness guarantees. The methods attached to data types are usually implemented as members of classes [11, 19, 28, 48]. By contrast, we define data types as inner classes and methods acting on them as sibling methods, and we support pattern matching on these inner classes. In our system, traits do group inner classes as families, and inner classes mixed into different top-level classes are considered distinct types. Path prefixes with concrete class names are *absolute* paths that distinguish inner classes in different contexts. The `this` and `now` path prefixes are *relative* paths that abstract over different versions of extensible variant types in a certain trait context. Therefore, our system supports a limited form of family polymorphism, although our inner classes do not

carry methods and are not late-bound. While we only support one level of nesting, some family polymorphism approaches support arbitrary levels of nesting and late-bound families [8, 11, 22], and types may depend on term-level constructs (such as Scala's path-dependent types [2]), which goes beyond what we support. As future work, we are interested in generalizing our support to full-fledged family polymorphism by allowing nested and late-bound trait definitions, enabling more extensibility and flexibility for modular DSL development.

Extensible programming paradigms. Persimmon [22] is a family polymorphism approach featuring extensible variant types. Like our work, Persimmon addresses the challenge of type-safe extensibility, particularly for pattern matching constructs that are traditionally difficult to extend. This is achieved by the "linkage" mechanism, a data structure containing type or term definitions of families, their reusable code components. Pattern matching cases are at the top level of family linkages. When typing a composition of families, linkages and the match cases inside are concatenated. Therefore, inline pattern matching and modular language optimizations are not supported in Persimmon. According to the authors, the theory of Persimmon requires linkages to be recomputed several times on demand when some type information of a family is required during type checking, which hinders separate type checking. Furthermore, Persimmon does not support covariant overriding of function or match case signatures in derived families, which we deem important for modular DSL development.

Zhang et al. [47] propose *Compositional Programming* (CP). CP leverages *the merge operator*, the introduction form of intersection types. The reusable components in CP are first-class traits, which can be nested and subject to composition via the merge operator. Recently, Sun et al. [43] have studied separate compilation of CP programs, and Xu et al. [46] propose a type difference operator that allows for method overriding. CP is also suitable for modular eDSL development [42]. However, due to the absence of inspectable AST data types, it is not straightforward to implement deep pattern matching in CP.

FPOP [20] and Rocqet [9] are two systems featuring extensible metatheory and certified compiler development by leveraging family polymorphism. Users program in a surface language of families containing extensible inductive types, recursive functions (i.e. `FRecursion`), and inductive proofs on those inductive types. The `FRecursion` construct is top-level only, and does not support deep patterns where constructors nest in each other. When a family extends an inductive type, an exhaustiveness check is performed to see if functions defined over the extended type are extended with the new cases. Moreover, overriding is not supported for every definition in general. By contrast, our approach supports deep pattern matching as a first-class term construct and supports overriding all methods; both are crucial features for modular DSL development. Also, our exhaustiveness checking is performed purely through the type system as subtyping checks, providing a uniform type-based mechanism across all cases.

SuperOOP. SuperOOP [12] is an object-oriented programming model composed of classes, interfaces, and mixins. By giving precise types to `this` and `super` for each mixin, SuperOOP imposes requirements on the object type and the inherited methods. Unlike our traits or Scala traits, mixins do not introduce new types or subtyping relationships. SuperOOP heavily relies on type inference to become practical, as it is difficult to write out all precise types manually. Although SuperOOP is general enough to serve as the target language of our trait system, encoding extensible variants in SuperOOP is cumbersome and complicates type error messages, showing its low-level nature. Inspired by Scala traits, our approach features a trait language, where method signatures reflect the runtime behavior of open recursion, overriding obligations are predictable, and exhaustiveness of pattern matching is guaranteed by types.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback that greatly improved the paper, Matthew Toohey for his helpful suggestions, and the participants of the Ontario Programming Languages Seminar 2025 for inspiring discussions. This work was partially funded by the Natural Sciences and Engineering Research Council of Canada.

References

[1] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L Steele Jr, Sam Tobin-Hochstadt, Joao Dias, Carl Eastlund, et al. 2005. The Fortress language specification. *Sun Microsystems* 139, 140 (2005), 116.

[2] Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. *The Essence of Dependent Object Types.* Springer International Publishing, Cham, 249–272. doi:10.1007/978-3-319-30936-1_14

[3] Xuan Bi and Bruno C. d. S. Oliveira. 2018. Typed First-Class Traits. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 109)*, Todd Millstein (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:28. doi:10.4230/LIPIcs.ECOOP.2018.9

[4] Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyễn. 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. doi:10.1145/2951913.2951928

[5] Chun Yin Chau and Lionel Parreaux. 2026. The Simple Essence of Boolean-Algebraic Subtyping: Semantic Soundness for Algebraic Union, Intersection, Negation, and Equi-recursive Types. *Proc. ACM Program. Lang.* 10, POPL, Article 47 (Jan. 2026), 30 pages. doi:10.1145/3776689

[6] Luyu Cheng and Lionel Parreaux. 2024. The Ultimate Conditional Syntax. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 306 (Oct. 2024), 30 pages. doi:10.1145/3689746

[7] Luyu Cheng and Lionel Parreaux. 2026. A Simple Recipe for Writing Decent Recursive Descent Parsers. In *40th European Conference on Object-Oriented Programming (ECOOP 2026) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 372)*, Robbert Krebbers and Alexandra Silva (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 30:1–30:28. doi:10.4230/LIPIcs.ECOOP.2026.30

[8] Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. 2007. Tribe: a simple virtual class calculus. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development* (Vancouver, British Columbia, Canada) *(AOSD '07).* Association for Computing Machinery, New York, NY, USA, 121–134. doi:10.1145/1218563.1218578

[9] Oghenevwogaga Ebresafe, Ian Zhao, Ende Jin, Arthur Bright, Charles Jian, and Yizhou Zhang. 2025. Certified compilers à la carte. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 372–395. doi:10.1145/3729261

[10] Erik Ernst. 2001. Family Polymorphism. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01).* Springer-Verlag, Berlin, Heidelberg, 303–326. doi:10.1007/3-540-45337-7_17

[11] 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. doi:10.1145/1111037.1111062

[12] Andong Fan and Lionel Parreaux. 2023. super-Charging Object-Oriented Programming Through Precise Typing of Open Recursion. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 11:1–11:28. doi:10.4230/LIPIcs.ECOOP.2023.11

[13] 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. doi:10.1007/11924661_17

[14] Cunyuan Gao and Lionel Parreaux. 2025. A Lightweight Type-and-Effect System for Invalidation Safety: Tracking Permanent and Temporary Invalidation with Constraint-Based Subtype Inference. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 366 (Oct. 2025), 31 pages. doi:10.1145/3763144

[15] Benedict R. Gaster and Mark P. Jones. 1996. A Polymorphic Type System for Extensible Records and Variants.

[16] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. 2008. Polymorphic embedding of dsls. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering* (Nashville, TN, USA) *(GPCE '08).* Association for Computing Machinery, New York, NY, USA, 137–148. doi:10.1145/1449913.1449935

[17] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. 2005. Regular Expression Types for XML. *ACM Trans. Program. Lang. Syst.* 27, 1 (Jan. 2005), 46–90. doi:10.1145/1053468.1053470

[18] 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. doi:10.1145/503502.503505

[19] Atsushi Igarashi, Chieri Saito, and Mirko Viroli. 2005. Lightweight Family Polymorphism. In *Programming Languages and Systems*, Kwangkeun Yi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 161–177. doi:10.1007/11575467_12

[20] Ende Jin, Nada Amin, and Yizhou Zhang. 2023. Extensible metatheory mechanization via family polymorphism. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1608–1632. doi:10.1145/3591286

[21] Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. 2014. Yin-yang: concealing the deep embedding of DSLs. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences* (Västerås, Sweden) *(GPCE 2014)*. Association for Computing Machinery, New York, NY, USA, 73–82. doi:10.1145/2658761.2658771

[22] Anastasiya Kravchuk-Kirilyuk, Gary Feng, Jonas Iskander, Yizhou Zhang, and Nada Amin. 2024. Persimmon: Nested Family Polymorphism with Extensible Variant Types. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 119 (apr 2024), 27 pages. doi:10.1145/3649836

[23] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, Kc Sivaramakrishnan, and Jérôme Vouillon. 2024. *The OCaml system release 5.2: Documentation and user's manual.* Intern report. Inria. 1044 pages. https://inria.hal.science/hal-00930213

[24] 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. doi:10.1145/3486610.3486894

[25] Guillaume André Fradji Martres. 2023. *Type-Preserving Compilation of Class-Based Languages.* Ph. D. Dissertation. EPFL, Lausanne. doi:10.5075/epfl-thesis-8218

[26] Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. 2012. Scala-virtualized. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation* (Philadelphia, Pennsylvania, USA) *(PEPM '12)*. Association for Computing Machinery, New York, NY, USA, 117–120. doi:10.1145/2103746.2103769

[27] J. Garrett Morris and James McKinna. 2019. Abstracting extensible data types: or, rows by any other name. *Proc. ACM Program. Lang.* 3, POPL, Article 12 (Jan. 2019), 28 pages. doi:10.1145/3290325

[28] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. 2004. Scalable Extensibility via Nested Inheritance. 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, 99–115. doi:10.1145/1028976.1028986

[29] 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).

[30] Martin Odersky and Matthias Zenger. 2005. Scalable component abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 41–57. doi:10.1145/1094811.1094815

[31] Bruno C. d. S. Oliveira, Cui Shaobo, and Baber Rehman. 2020. The Duality of Subtyping. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 29:1–29:29. doi:10.4230/LIPIcs.ECOOP.2020.29

[32] Lionel Parreaux. 2020. *Type-Safe Metaprogramming and Compilation Techniques For Designing Efficient Systems in High-Level Languages.* Ph. D. Dissertation. EPFL, Lausanne. doi:10.5075/epfl-thesis-10285

[33] Lionel Parreaux, Aleksander Boruch-Gruszecki, Andong Fan, and Chun Yin Chau. 2024. When Subtyping Constraints Liberate: A Novel Type Inference Approach for First-Class Polymorphism. *Proc. ACM Program. Lang.* 8, POPL, Article 48 (jan 2024), 33 pages. doi:10.1145/3632890

[34] 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. doi:10.1145/3563304

[35] Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. 2017. Squid: type-safe, hygienic, and reusable quasiquotes. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala* (Vancouver, BC, Canada) *(SCALA 2017)*. Association for Computing Machinery, New York, NY, USA, 56–66. doi:10.1145/3136000.3136005

[36] D. Rémy. 1989. Type checking records and variants in a natural extension of ML. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 77–88. doi:10.1145/75277.75284

[37] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *SIGPLAN Not.* 46, 2 (Oct. 2010), 127–136. doi:10.1145/1942788.1868314

[38] Tiark Rompf, Arvind K Sujeeth, HyoukJoong Lee, Kevin J Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. Building-blocks for performance oriented DSLs. *arXiv preprint arXiv:1109.0778* (2011). doi:10.4204/EPTCS.66.5

[39] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. 2003. Traits: Composable units of behaviour. In *European Conference on Object-Oriented Programming*. Springer, 248–274. doi:10.1007/978-3-540-45070-

2_12
[40] Charles Smith and Sophia Drossopoulou. 2005. Chai: Traits for Java-Like Languages. In *ECOOP 2005 - Object-Oriented Programming*, Andrew P. Black (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 453–478. doi:10.1007/11531142_20
[41] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embed. Comput. Syst.* 13, 4s, Article 134 (April 2014), 25 pages. doi:10.1145/2584665
[42] 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. doi:10.1145/3563294
[43] Yaozhu Sun, Xuejing Huang, and Bruno C. D. S. Oliveira. 2025. Type-Safe Compilation of Dynamic Inheritance via Merging. *ACM Trans. Program. Lang. Syst.* (Oct. 2025). doi:10.1145/3765518 Just Accepted.
[44] Philip Wadler. 1998. The expression problem. Posted on the Java Genericity mailing list. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt
[45] Mitchell Wand. 1987. Complete type inference for simple objects. In *Proc., IEEE Symposium on Logic in Computer Science*. 37–44.
[46] Han Xu, Xuejing Huang, and Bruno C. d. S. Oliveira. 2023. Making a Type Difference: Subtraction on Intersection Types as Generalized Record Operations. *Proc. ACM Program. Lang.* 7, POPL, Article 31 (Jan. 2023), 28 pages. doi:10.1145/3571224
[47] 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. doi:10.1145/3460228
[48] Yizhou Zhang and Andrew C. Myers. 2017. Familia: unifying interfaces, type classes, and family polymorphism. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 70 (Oct. 2017), 31 pages. doi:10.1145/3133894

Received 2026-03-17; accepted 2026-06-10