A foundation for trait-based metaprogramming (extended version)

John Reppy
University of Chicago
jhr@cs.uchicago.edu

Aaron Turon
University of Chicago
adrassi@cs.uchicago.edu

February 1, 2006

Abstract

Schärli *et al.* introduced traits as reusable units of behavior independent of the inheritance hierarchy. Despite their relative simplicity, traits offer a surprisingly rich calculus. Trait calculi typically include operations for resolving conflicts when composing two traits. In the existing work on traits, these operations (method exclusion and aliasing) are *shallow*, *i.e.*, they have no effect on the body of the other methods in the trait. In this paper, we present a new trait system, based on the Fisher-Reppy trait calculus, that adds *deep* operations (method hiding and renaming) to support conflict resolution. The proposed operations are deep in the sense that they preserve any existing connections between the affected method and the other methods of the trait. Our system uses Riecke-Stone dictionaries to support these features. In addition, we define a more fine-grained mechanism for tracking trait types than in previous systems. The resulting calculus is more flexible and expressive, and can serve as the foundation for *trait-based metaprogramming*, an idiom we introduce. This is an extended version of our workshop paper with the complete formal model and additional details for the soundness proof.

1 Introduction

A *trait* is a simple collection of methods that represent the partial implementation of a class. Methods defined in a trait are *provided* methods; any method referenced in a trait but not provided by it is a *required* method. Traits are similar to abstract classes, but with two important differences: traits cannot introduce state, and they do not lie within the inheritance hierarchy. The primary mechanism of class reuse is inheritance, which is an asymmetric operation. With traits, reuse takes place via *composition*, a symmetric concatenation of two traits. In addition to composition, trait calculi include fine-grained operations for manipulating traits as method suites. Ultimately, traits are *inlined* into classes during class formation. Unfulfilled trait requirements must be provided by the class at the time of inlining.

Traits were introduced by Schärli *et al.* in the setting of SMALLTALK [SDNB03]. A companion paper explored the use of traits to refactor the SMALLTALK collection classes, with encouraging results: a 10% reduction in method count for a 12% reduction in overall code size [BSD03]. Since SMALLTALK is untyped, the original work on traits did not include a type system. Fisher and Reppy gave a calculus for statically-typed traits [FR04, FR03]. Other work subsequently developed typed trait calculi for JAVA [LS04, SD05]. Quitslund performed a simple analysis of the Swing Java library, which suggests that code reuse can also be improved by adding traits to Java [Qui04]. A fair amount of activity has followed, with trait implementations underway or completed for C#, JAVA, PERL, and SCALA.[1]

We present a calculus, based on the Fisher-Reppy polymorphic trait calculus [FR03], with support for trait privacy, hiding and deep renaming of trait methods, and a more granular trait typing. Our calculus is more expressive (it provides new forms of conflict-resolution) and more flexible (it allows after-the-fact renaming) than the previous work. Traits provide a useful mechanism for sharing code between otherwise unrelated classes. By adding deep renaming, our trait calculus supports sharing code between *methods*. For example, the JAVA notion of synchronized methods can implemented as a trait in our system and can be applied to multiple methods in the same class to produce synchronized versions. We term this new use of traits *trait-based metaprogramming*.

We review the standard trait operations in Section 2 and describe our additions in Section 3; the presentation is organized around a series of examples, culminating with the introduction of our metaprogramming idiom. The formal description of our proposal begins with Section 4, which outlines our notation and the syntax of the calculus. Section 5 describes the static semantics. The system types traits at the time of trait formation, rather than trait inlining. It improves on previous work by tracking trait requirements at a per-method, rather than per-trait basis, which allows spurious requirements to be dropped as a trait is manipulated. Properly handling evaluation in the presence of privacy can be somewhat subtle. We utilize Riecke-Stone dictionaries [RS02] to handle privacy; this technique also provides support for renaming. Our approach is detailed in Section 6. Finally, we outline the proof of type soundness in Section 7. After the formal description, we take stock of the calculus in a broader context. Section 8 details other work in traits and relates our presentation of traits to other constructs, *e.g.* mixins and aspects. The calculus raises some interesting questions for language design. We examine these questions and conclude in Section 9.

This tech report is an extended version of our workshop paper [RT06]; this version includes details on the run-time typing rules, which can be found in Section 7. Also included is an appendix with the complete formal model.

2 Traits

Traits originate from the observation that classes serve two often conflicting purposes [SDNB03]. From one perspective, classes are meant to generate objects, which means they must be complete and monolithic. At the same time, classes act as units of reuse via inheritance, and from this perspective they should be small, fine-grained, and possibly incomplete. Inheritance must straddle these two roles, which often forces the designer of a class hierarchy to choose between interface cleanliness and implementation cleanliness.

Consider the case of two classes in different subtrees of the inheritance hierarchy which both implement some common protocol. To avoid code duplication, the protocol implementation should be shared between the two classes. In a single inheritance framework, the common code can be lifted to a shared superclass, but doing so pollutes the interface of the superclass, affecting

[1] See `http://www.iam.unibe.ch/~scg/Research/Traits/` for more information.

all of its subclasses. With multiple inheritance, the protocol implementation can reside in a new superclass that is inherited along with the existing superclasses, but multiple inheritance complicates the implementation of subclasses (*e.g.*, with respect to instance variables).

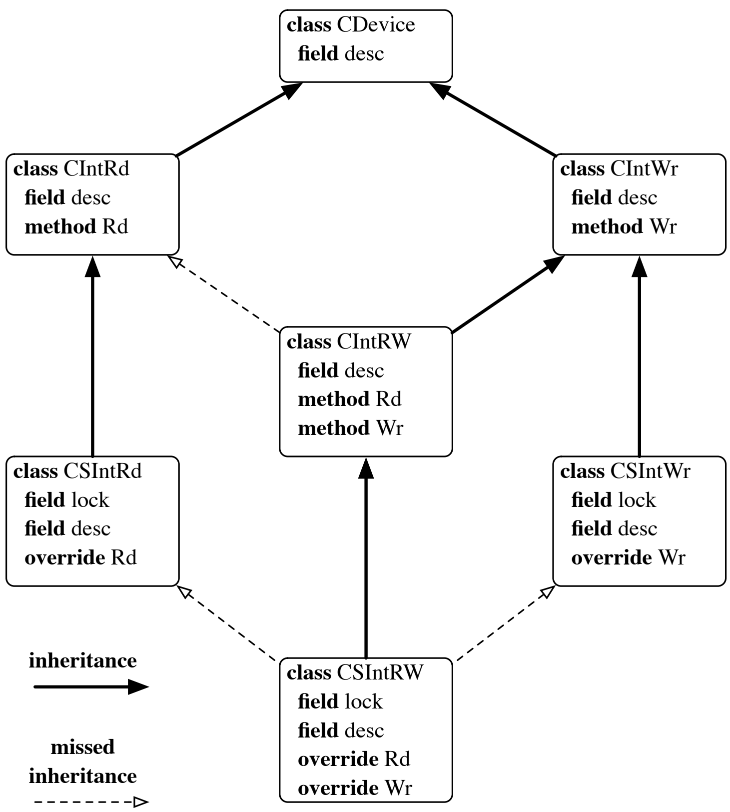

Figure 1: Synchronized readers and writers in a single inheritance framework

Figure 1 illustrates these issues with a concrete class hierarchy in a single inheritance framework. The root of the hierarchy is the `CDevice`[2] class, which implements I/O on a file descriptor. It has two subclasses for, respectively, reading and writing integers on the device. Defining a class that supports both reading and writing (`CIntRW`) requires reimplementing one of the methods (denoted by the dashed arrow in the figure). While we could lift the `Rd` and `Wr` methods to the `CDevice` class, doing so would pollute the interface of the original `CIntRd` and `CIntWr` subclasses as well as new subclasses such as boolean readers and writers. The class hierarchy is further extended with support for synchronized reading and writing by adding a lock. Single inheritance again forces a reimplementation of methods.

In contrast to inheritance, which specifies the relationship between a family of classes, traits allow the implementation of a single class to be factored into multiple, structured parts. Classes are retained as hierarchically organized object generators, but traits are introduced as flatly organized, partial class implementations. Traits thus assist in separating the roles distinguished above.

[2]For concrete names, we prefix classes with `C` and traits with `T`.

Figure 2 shows how traits can allow code reuse without having to define methods too high in the hierarchy; in this example, four traits are used to generate six classes. Note that the traits in the example have field requirements, not just method requirements.[3] The **`override`** annotation on provided methods signifies that the method invokes the super-method with the same name.

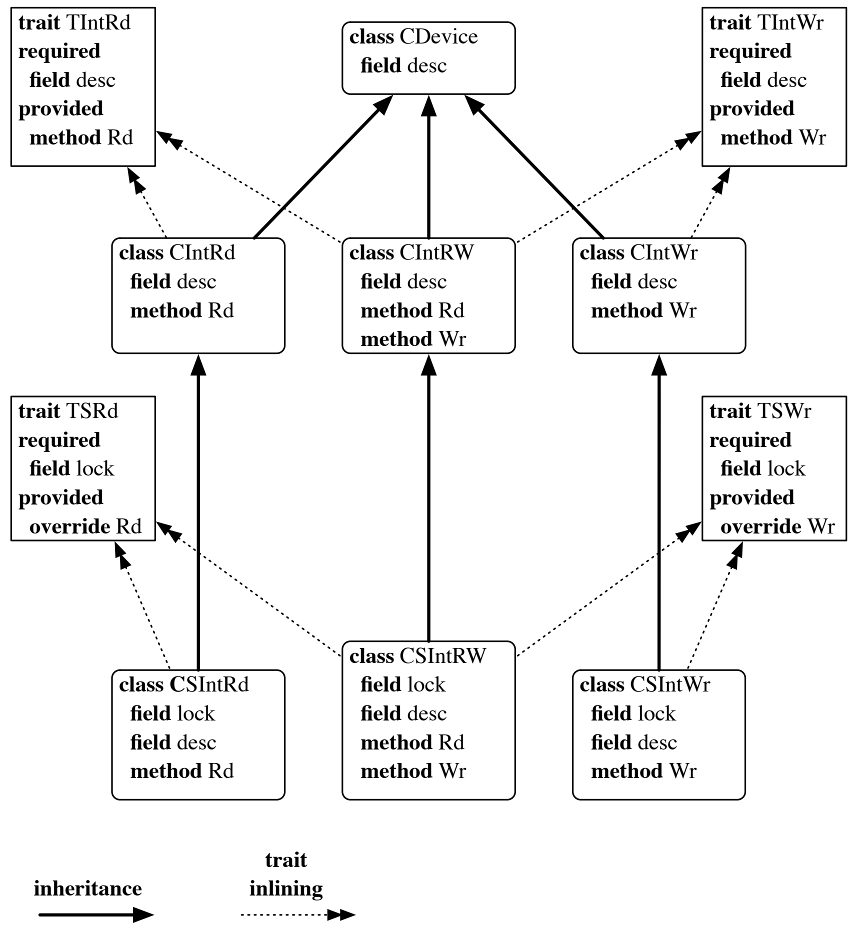

Figure 2: Using traits to implement synchronized readers and writers

Operations on traits

Traits are partial class implementations, but they are restricted to providing only a simple set of methods. In particular, traits cannot introduce state. The methods of a trait are only loosely coupled: they can be freely removed and replaced by other implementations. These properties make traits more nimble and lighter-weight than either multiple inheritance [Str94] or mixins [BC90, FKF98].

[3] Field requirements were introduced in [FR04].

Two traits can be combined via *trait composition*, written $T_1 + T_2$. The resulting trait is a flat merger of the operands. Composing two traits may fulfill method requirements for one or both of the traits. In a statically-typed setting, composition requires its operands to be disjoint; method conflicts must be resolved explicitly, using other trait operations. Under this definition, trait composition is commutative, obviating the need for a linear ordering found with single inheritance mixins.

Trait composition is a symmetric alternative to reuse via inheritance. By defining additional operations on traits, one can capture more complex idioms of reuse. The typical suite of trait operations include *alias* and *exclude*, which are useful for conflict resolution. As an illustration, assume that we have two traits, `TPoint` and `TColored`, which both provide a `toString` method, and that we wish to compose them into a single trait, `TCPoint`. In the simplest case, we can resolve such a conflict by choosing one method or the other. We exclude the unwanted method implementation, and compose the resulting trait:

```
TCPoint = TPoint + (TColored exclude toString)
```

`TCPoint` will provide a single `toString` method, using the implementation from `TPoint`. In addition, any invocations of `toString` from within other methods provided by `TColored` will now invoke `TPoint`'s implementation, making the exclude-compose combination similar to method override.

Sometimes it is useful to retain both implementations of a conflicting method, perhaps providing a new implementation combining the two. The alias operation creates a new name for an existing method:

```
TCPoint = {
    provides toString() : string {
       self.strP() + ": " + self.strC();
    }
  } + ((TPoint alias toString as strP)
         exclude toString)
    + ((TColored alias toString as strC)
         exclude toString)
```

It is important to realize that, while `strP` and `strC` are available in this version of `TCPoint`, the original `toString` methods have effectively been overridden. Invocations of `toString` from the other methods provided by `TPoint` and `TColored` will use the new, combined `toString` implementation provided by `TCPoint`. The combination of aliasing and excluding thus yields a *shallow* renaming: existing references to an aliased method continue to refer to the original method name.

3 Traits with hiding and deep renaming

We extend the trait system of the previous section with two new trait operations, *hide* and *rename*, which act as the deep variants of exclude and alias. These two operations provide new forms of conflict resolution when composing traits. Each is also useful in isolation: hiding without composition yields trait privacy, while renaming yields an idiom we term *trait-based metaprogramming*, the most exciting aspect of our work.

The hide operation permanently binds a provided method to a trait, while hiding the method's name. A new method with the same name can be introduced as a new provided or required method of the trait, but existing references to the method from other provided methods are statically bound to its implementation at the time of hiding. Returning to the `TCPoint` example, we can write

```
TCPoint = TPoint + (TColored hide toString)
```

Here, `TCPoint` will provide the `toString` implementation from `TPoint`. Unlike the exclusion example, any references from within `TColored` to the `toString` method will continue to refer to `TColored`'s implementation. In effect, `TColored`'s `toString` implementation is provided by `TCPoint`, but in a nameless and inaccessible form. Where combining exclusion and composition leads to overriding, the combination of hiding and composition yields *shadowing*.

Method hiding can also be used to hide trait implementation details — *i.e.*, as a form of trait privacy. A trait implements some collection of behavior, and it may need to make use of new methods that are specific to its implementation but are not appropriate to provide publicly. Although such "helper methods" could be made private after they are inlined into a class, we believe that traits should not only factor out, but also encapsulate, units of behavior. The importance of trait privacy will depend to some extent on the strength of the surrounding language features. In a language with a powerful module system, for example, it is likely that trait privacy could be achieved through signature ascription instead [FR99]. In any case, trait privacy is a free by-product of introducing method hiding for conflict resolution.

Loosening the connection between method names and method implementations suggests the possibility of a deep renaming operation, as opposed to shallow renaming with alias-exclude. As with the other operations, pairing renaming with composition provides a new form of conflict resolution:

```
TCPoint = (TPoint rename toString to strP)
        + (TColored rename toString to strC)
```

This version of `TCPoint` does not provide a `toString` method at all. Existing references to `toString` from within `TPoint` and `TColored` now refer to `strP` and `strC`, respectively; this is the sense in with the renaming is "deep." `TCPoint` can now be extended with another implementation of `toString`, even one with a different type.

Of course, renaming can be used alone in order to align a "misnamed" provided method with an existing class hierarchy or signature constraint. Required methods may be renamed for the same reason. Furthermore, super-method names may be renamed, so that for example all the invocations of **`super`**`.foo` within a trait may be renamed to invoke **`super`**`.bar`. The rationale for this last form of renaming will become more clear in the following discussion.

Trait-based metaprogramming

Schärli *et al.* summarized their system with the following equation [SDNB03]:

$$\mathit{Class} = \mathit{Superclass} + \mathit{State} + \mathit{Traits} + \mathit{Glue}$$

A trait represents a flat fragment of a class's behavior, and class behavior is closely tied to naming. But a trait can also be seen as a collection of named provided methods *parameterized* over a collection of named required methods (and fields). Thus, traits capture a relationship between two families of named methods. Often this relationship carries as much information as the names of the methods themselves.

Reconsider the example of traits usage presented in Figure 2. The trait `TIntRd` requires a field `desc` and provides a method `Rd`. In a broad sense, it does not matter what we call these entities (`Rd`, `read`, `Read`, *etc.*), so long as (1) the names convey "descriptor" and "read," respectively, and (2) the relationship between the entities remains intact. After-the-fact renaming allows the names to be changed as needed.

But what can we say about the traits `TSRd` and `TSWr` from the same example? Each requires the field `lock` and provides a single method that wraps a super-method invocation with synchronization code. For these traits, the relationship between the provided methods and the trait's requirements carries essentially *all* of the relevant information; the provided method names `Rd` and `Wr` are incidental. We can capture the relationship in a single trait, `TSync`, which is parameterized by type:

```
trait TSync<ty1, ty2> = {
    provides Op(x : ty1) : ty2 {
      self.lock.Acquire();
      super.Op(x) before
        self.lock.Release();
    }
    requires field lock : LockObj
  }
```

The generic method `Op` and its associated super-send can be renamed and used to override a method with a synchronized wrapper. Returning to the readers and writers example, we could rename `Op` to `Rd` and **`super.`**`Op` to **`super.`**`Rd`, and likewise for `Wr`. The key insight is that `TSync` can be instantiated several times, once for each method we want to synchronize. Figure 3 shows the readers/writers example using `TSync`; we now have six classes derived from three traits. It is worth noting that, with the ability to rename field requirements, different instantiations of `TSync` could use different locks. Our model does not support field renaming, but this is for simplicity only; the feature would be a fairly straightforward addition.

Traits were intended to capture specific, named behavior at the class level. Examples like `TSync` shift the focus to behavior at the method family level. With the latter perspective it becomes sensible to inline a trait several times into a class; we label this idiom *trait-based metaprogramming*. The power of the technique is that we can freely mix flat requirements for the class (*e.g.*, `lock`) with provided methods that may be instantiated several times. Traits provide the structure for specifying method relationships, and renaming provides the flexibility to apply traits in multiple contexts within a single class.

The metaprogramming technique we have outlined is somewhat *ad hoc*: examples like `TSync` would be best served by an explicit notion of method name abstraction. A good starting point would be a "lambda calculus of traits," with method renaming playing the role of substitution. It is easy to imagine further extensions. A sophisticated trait-metalanguage could codify certain patterns of trait use, perhaps providing a mechanism like JAVA's **synchronized** keyword for appropriately instantiating traits like `TSync`. Borrowing a line from aspect-oriented programming, a language might also allow *join points* to be specified for trait application. Such possibilities are exciting and deserve to be explored, but before we can define abstraction mechanisms for higher-level calculi, we need a well-defined notion of substitution.

Starting with the next section, we give a detailed semantics of a calculus with hiding and renaming for traits, which can serve as the foundation for trait-based metaprogramming.

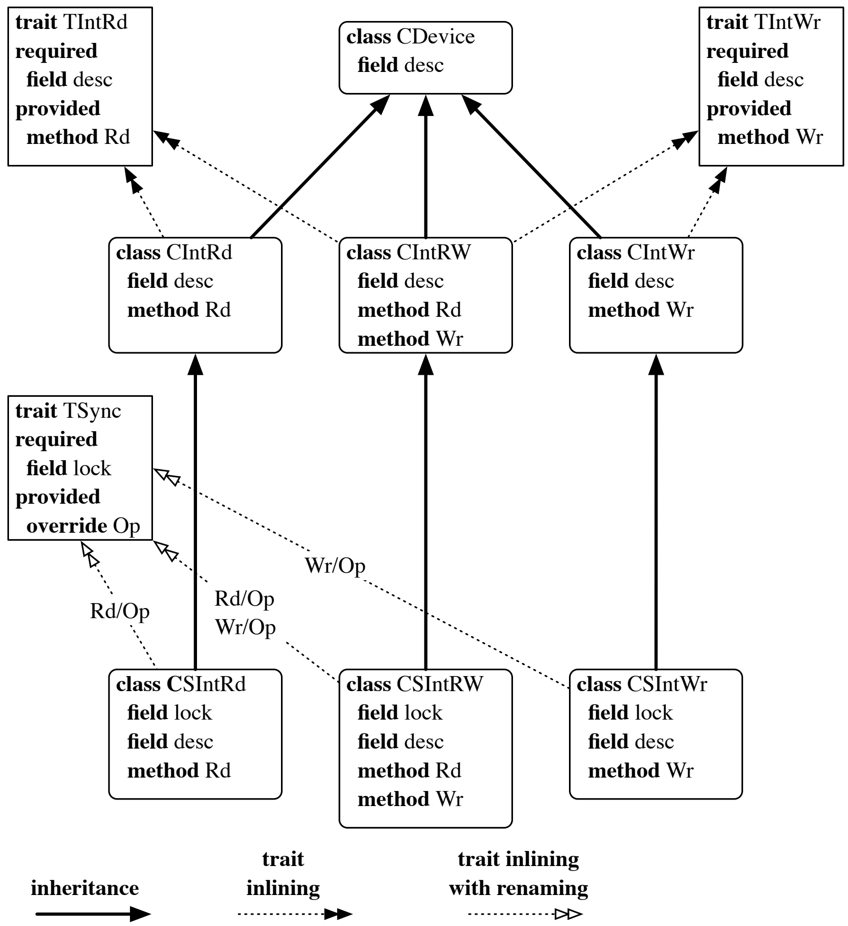

Figure 3: Using traits with renaming to implement synchronized readers and writers

4 A trait calculus with hiding and renaming

To model our proposed features, we have developed a statically-typed trait calculus loosely based on Fisher and Reppy's [FR03]. The calculus is meant to give a rigorous semantics of the new trait operations, and does not represent a surface language design. We model the trait language in considerable detail, but restrict the rest of the language to essential features. Specifically, our class definition form does not support definition of methods directly. Instead, we leave this responsibility to traits. This choice is purely to minimize the complexity of the formal system; method definition in classes could be easily added.

The syntax for our calculus is given in Figure 4. To keep the syntax lightweight, we separate names into disjoint "universes," as shown in Figure 5. This convention allows entities to be distinguished by the "type" of their name, without any additional bookkeeping.

In our model, a program is a series of zero or more declarations followed by an expression. The calculus is organized into three components: a trait language, a class language, and an ex-

P	$::=$	$D;\ P \quad \vert \quad e$	program
D	$::=$	$t = (\vec{\alpha})T$	trait declaration
	$\vert$	$c = C$	class declaration
	$\vert$	$x = e$	expression declaration
T	$::=$	$t(\vec{\tau})$	polymorphic trait name
	$\vert$	$\langle\!\vert\ \mu;\ \rho\ \vert\!\rangle$	trait formation
	$\vert$	$T_1 + T_2$	trait composition
	$\vert$	T **exclude** m	method exclusion
	$\vert$	T **hide** m	method hiding
	$\vert$	T **alias** m **as** m'	method aliasing
	$\vert$	T **rename** r **to** r'	method renaming
μ	$::=$	$m\ (x : \tau_1) : \tau_2\ \{e\}$	method
ρ	$::=$	$\langle\!\langle r : \tau_r\ ^{r \in \mathcal{R}} \rangle\!\rangle$	inlining assumptions
θ	$::=$	$\langle\!\vert\ m : \tau_m @ \rho_m\ ^{m \in \mathcal{M}}\ \vert\!\rangle$	trait type
	$\vert$	$\Lambda(\vec{\alpha}).\theta$	polymorphic trait type
C	$::=$	c	class name
	$\vert$	**nil**	empty class
	$\vert$	I **in** T **extends** C	subclass formation
I	$::=$	$\lambda(x : \tau).($**super** $e_1) \oplus e_2$	constructor
χ	$::=$	$\tau \to \{\!\vert\ l : \tau_l\ ^{l \in \mathcal{L}}\ \vert\!\}$	class type
e	$::=$	x	variable
	$\vert$	$\lambda(x : \tau).e$	function abstraction
	$\vert$	$e_1\ e_2$	function application
	$\vert$	**new** $C\ e$	object instantiation
	$\vert$	**self**	host object
	$\vert$	**super**$.m$	super-method dispatch
	$\vert$	$e.m$	method dispatch
	$\vert$	$e.f$	field selection
	$\vert$	$e_1.f := e_2$	field update
	$\vert$	$\{f = e_f\ ^{f \in \mathcal{F}}\}$	field record
	$\vert$	$e_1 \oplus e_2$	field concatenation
	$\vert$	$()$	unit value
τ	$::=$	α	type variable
	$\vert$	$\langle l : \tau_l\ ^{l \in \mathcal{L}} \rangle$	object type
	$\vert$	$\tau_1 \to \tau_2$	function type
	$\vert$	$\{f : \tau_f\ ^{f \in \mathcal{F}}\}$	field record type
	$\vert$	**unit**	unit type

Figure 4: Trait calculus syntax

	Items	$\in$	Sets	$\subset$	Universe
Field names	f	$\in$	$\mathcal{F}$	$\subset$	$\mathcal{F}_U$
Method names	m	$\in$	$\mathcal{M}$	$\subset$	$\mathcal{M}_U$
Labels	l	$\in$	$\mathcal{L}$	$\subset$	$\mathcal{L}_U = \mathcal{F}_U \cup \mathcal{M}_U$
Super methods	**super**.m	$\in$	$\mathcal{S}$	$\subset$	$\mathcal{S}_U$
Requirements	r	$\in$	$\mathcal{R}$	$\subset$	$\mathcal{R}_U = \mathcal{L}_U \cup \mathcal{S}_U$
Slots	i	$\in$	$\mathcal{I}$	$\subset$	$\mathcal{I}_U$
Variables	x	$\in$			VARS
Type vars	α	$\in$			TYVARS
Trait names	t	$\in$			TRAITNAMES
Class names	c	$\in$			CLASSNAMES

Figure 5: Naming conventions

pression language. The trait language includes expressions forms for all of the features we have discussed: composition of traits, and exclusion, hiding, aliasing, and renaming of methods. The declaration form for traits allows parameterization by type variables. Traits are initially formed with a single method rather than a method suite, which simplifies the semantics; a trait with multiple methods can be constructed via repeated composition.

The class language provides a simple model of single inheritance with no visibility control. The calculus has an empty base class, **nil**; all other classes must be defined via inheritance. Subclass formation takes a super class, a trait expression, and a constructor. In our model, the only methods introduced in a subclass are those provided by the trait; there is no separate notion of method definition for classes. In particular, this means that any required methods of the trait must be provided by the super class. We will sometimes refer to subclass formation as *trait inlining* to emphasize the role that traits play. When the trait being inlined is the result of simple trait formation and concatenation, trait inlining reduces to standard single inheritance.

Subclass formation also allows the introduction of state via a constructor function. State is restricted to *field records* which can be concatenated using the $\oplus$ operator. Class constructors are syntactically constrained to apply the super class constructor to an expression, and concatenate the result with a new field record. Fields referenced in an inlined trait may originate from either the super class or the newly formed subclass.

The expression language is a simple object calculus with first class functions. It has imperative features (object instantiation and field update) which allow us to put the class language to work in a realistic way.

We will need several notational tools: we write $A \pitchfork B$ for $A \cap B = \emptyset$; when f is a function, $f[x \mapsto y]$ denotes the function that takes x to y and otherwise behaves the same as f; the notation $f \downarrow A$ yields the restriction of the function f to the domain A; the notation $f \setminus x$ is shorthand for $f \downarrow (\mathrm{dom}(f) \setminus \{x\})$ or, if x is a set, $f \downarrow (\mathrm{dom}(f) \setminus x)$. We make heavy use of notation like $\{x_y\ ^{y \in Y}\}$ to describe a collection of elements x_y indexed by a set Y. Such notation allows us to give types to a collection of labels, *e.g.* $\langle l : \tau_l\ ^{l \in \mathcal{L}} \rangle$. We define the binary operator $\uplus$, as follows:

$$\frac{\tau_l = \tau'_l \text{ for all } l \in \mathcal{L}_1 \cap \mathcal{L}_2}{\{l : \tau_l\ ^{l \in \mathcal{L}_1}\} \uplus \{l : \tau'_l\ ^{l \in \mathcal{L}_2}\} = \{l : \tau_l\ ^{l \in \mathcal{L}_1},\ l : \tau'_l\ ^{l \in \mathcal{L}_2}\}}$$

The $\uplus$ operator joins two possibly overlapping label/type collections. It is only defined when the operands agree on the type of any shared labels. While we group different label/type collections in syntactically distinct categories (*e.g.*, trait types versus object types), we freely use $\uplus$ to join two or more such collections in the same syntactic category.

5 Static semantics

Typing judgments in our calculus are written in terms of an ordered context Γ. Types can inhabit one of three syntactic categories: trait types θ, class types χ, or expression types τ. The most important judgment forms are the following:

$\Gamma \vdash T : \theta$	trait T has type θ
$\Gamma \vdash C : \chi$	class C has type χ
$\Gamma \vdash e : \tau$	expression e has type τ
$\Gamma \vdash \mu : \tau$	method μ has type τ
$\Gamma \vdash \tau_1 <: \tau_2$	τ_1 is a subtype of τ_2

We also make heavy use of well-formedness checks, which take the form $\Gamma \vdash \Box \text{ ok}$ where $\Box$ is a type form, or $\Gamma \vdash \text{ok}$ to assert that the context Γ is well-formed.

Although traits can be viewed as sophisticated syntactic sugar that is "flattened" to a core class-based language [NDS06], there are advantages to recognizing traits directly. For the static semantics, giving types to traits allows the detection of a number of errors during trait manipulation that would otherwise not be detected until trait inlining; it also makes separate compilation of traits possible.

In order to typecheck a trait, we must know the types of all of its provided methods and of the self-methods, super-methods, and fields that it mentions. This type information amounts to a collection of assumptions about (or constraints on) the classes in which the trait will be inlined. The assumptions are guaranteed to hold of well-typed provided methods, which are syntactically required to specify their own type. The remaining assumptions constitute requirements of the trait, which fall into two categories:

1. Self-method requirements, which can be fulfilled via trait concatenation or trait method aliasing.

2. Super-method and field requirements, which can only be fulfilled when inlining a trait.

A straightforward type system might structure trait types as a collection of typed labels, some of which are marked as required; this is the approach of [FR03]. Unfortunately, this view of trait types forces a conservative typing of method exclusion. Consider the trait `TFoo`:

```
trait TFoo = {
    provides A() : unit { print("Hello!"); }
    provides B() : unit { self.A(); self.C(); }
    requires C : unit -> unit
  }
```

The trait `TFoo` **`exclude`** `A` should require both `A` and `C`, while `TFoo` **`exclude`** `B` should not have any requirements. But a flat trait type assigning types to labels gives no way to distinguish between excluding `A` and excluding `B`; whenever a method is excluded, it must be conservatively counted as a required method because it may be invoked from another provided method.

To overcome this limitation, we track requirements on a per-provided-method, rather than per-trait, basis. The requirements for a provided method are collected into a set of *inlining assumptions* ρ that represent that method's view of any class that inlines the trait. To distinguish between super-method and self-method requirements, we have a universe of super-method names, $\mathcal{S}_U$; for any $s \in \mathcal{S}_U$ there is a unique $m \in \mathcal{M}_U$ such that $s = \mathbf{super}.m$. A method may require both m and $\mathbf{super}.m$, but for the two requirements to be coherent, their types must be compatible. Since our calculus only supports width subtyping, an overriding method must have the same type as its corresponding super-method. This condition is reflected in the well-formedness judgment for inlining assumptions:

$$\frac{\Gamma \vdash \tau_r \text{ ok for all } r \in \mathcal{R} \qquad \tau_m = \tau_{\mathbf{super}.m} \text{ for all } m \text{ with } m \in \mathcal{R}, \mathbf{super}.m \in \mathcal{R}}{\Gamma \vdash \langle\!\langle r : \tau_r{}^{\,r \in \mathcal{R}} \rangle\!\rangle \text{ ok}}$$

Inlining assumptions can be seen as a compact and uniform representation of a pair of object types, $\tau_{\mathbf{super}}$ and $\tau_{\mathbf{self}}$, which are supertypes of the eventual super- and self-object types. To transform a set of inlining assumptions into its corresponding object types, we have the functions super and self:

$$\mathrm{super}\ \langle\!\langle r : \tau_r{}^{\,r \in \mathcal{R}} \rangle\!\rangle = \langle m : \tau_{\mathbf{super}.m}{}^{\,\mathbf{super}.m \in \mathcal{R}} \rangle$$
$$\mathrm{self}\ \langle\!\langle r : \tau_r{}^{\,r \in \mathcal{R}} \rangle\!\rangle = \langle l : \tau_l{}^{\,l \in \mathcal{R} \cap \mathcal{L}_U} \rangle$$

A trait type θ is a labeled collection of method types with inlining assumptions. A trait formation expression provides a single method and the initial inlining assumptions for that method. To type trait formation, we ensure that the given inlining assumptions are well-formed and that the given method can be typed under them:

$$\frac{\mu = m\ (x : \tau_1) : \tau_2\ \{e\} \quad \Gamma \vdash \rho \uplus \langle\!\langle m : \tau \rangle\!\rangle \text{ ok} \qquad \Gamma,\ \mathit{super} : \mathrm{super}(\rho),\ \mathit{self} : \mathrm{self}(\rho) \vdash \mu\ :\ \tau}{\Gamma \vdash \langle\!|\ \mu;\ \rho\ |\!\rangle\ :\ \langle\!|\ m : \tau @ \rho\ |\!\rangle}$$

Notice that a *recursive* method must specify its own type in its inlining assumptions. The $\uplus$ operator ensures that this type agrees with the method's actual type.

Although trait formation only produces single-method traits, in general traits will acquire several methods via composition. Each provided method has its own inlining assumptions, but we also want to consider the inlining assumptions for the trait as a whole. We introduce a function Inl that yields the trait inlining assumptions for given a trait type. Trait inlining assumptions are formed using the $\uplus$ operator, which ensures that repeated assumptions about a method or field will have the same type:

$$\frac{\rho = \biguplus_{m \in \mathcal{M}} (\rho_m \uplus \langle\!\langle m : \tau_m \rangle\!\rangle) \qquad \Gamma \vdash \rho \text{ ok}}{\mathrm{Inl}(\langle\!|\ m : \tau_m @ \rho_m{}^{\,m \in \mathcal{M}}\ |\!\rangle) = \rho}$$

We also have a well-formedness check for trait types, which simply checks that Inl can succeed in producing inlining assumptions for the trait:

$$\frac{\mathrm{Inl}(\theta) \text{ is defined}}{\Gamma \vdash \theta \text{ ok}}$$

The well-formedness check for trait types does a lot of work for the type system by centralizing coherency checking. Typing judgments need only include additional, specialized constraints. For trait composition, the additional constraint is that the given traits are disjoint (*i.e.*, specify a disjointly-named set of provided methods):

$$\frac{\Gamma \vdash T_1 \ : \ \theta_1 \quad \Gamma \vdash T_2 \ : \ \theta_2 \qquad \text{dom}(\theta_1) \pitchfork \text{dom}(\theta_2) \quad \Gamma \vdash \theta_1 \uplus \theta_2 \text{ ok}}{\Gamma \vdash T_1 + T_2 \ : \ \theta_1 \uplus \theta_2}$$

Method exclusion requires that the method to be excluded is actually provided by the trait:

$$\frac{\Gamma \vdash T \ : \ \theta \quad m \in \theta \quad \Gamma \vdash \theta \setminus m \text{ ok}}{\Gamma \vdash T \ \mathbf{exclude}\ m \ : \ \theta \setminus m}$$

Method aliasing is somewhat more complex. For aliasing m as m', the typing judgment first ensures that m is a provided method and that m' is not. It then forms a new trait type by joining inlining assumptions for m' to the old trait type. The inlining assumptions for m' are the same as those for m; in particular, if m invoked itself, m' will invoke m:[4]

$$\frac{\Gamma \vdash T \ : \ \theta \quad m \in \theta \quad m' \notin \theta \qquad \theta' = \theta \uplus \langle\!| \ m' : \theta(m) \ |\!\rangle \quad \Gamma \vdash \theta' \text{ ok}}{\Gamma \vdash T \ \mathbf{alias}\ m \ \mathbf{as}\ m' \ : \ \theta'}$$

Note that m' may be the name of a required method, so aliasing can be used to fulfill trait requirements. The same is true for renaming.

Method renaming allows provided methods, required self-methods, and required super-methods to be renamed. To rename r to r', the typing judgment checks that r and r' are not field names, that r is mentioned in the inlining assumptions (*i.e.*, it is either provided or required), and that r' is not a provided method. The new trait type is formed in two steps: first, r is renamed to r' in all of the sets of inlining assumptions; then, r is renamed to r' in the resulting trait type, which will only have an effect if r is a provided method:

$$\frac{\Gamma \vdash T \ : \ \langle\!| \ m : \tau_m @ \rho_m \ ^{m \in \mathcal{M}} \ |\!\rangle \qquad r, r' \notin \mathcal{F}_U \quad r' \notin \mathcal{M} \quad r \in \rho_m \text{ for some } \rho_m \in \mathcal{M} \qquad \theta = \langle\!| \ m : \tau_m @ (\rho_m[r'/r]) \ ^{m \in \mathcal{M}} \ |\!\rangle [r'/r] \quad \Gamma \vdash \theta \text{ ok}}{\Gamma \vdash T \ \mathbf{rename}\ r \ \mathbf{to}\ r' \ : \ \theta}$$

Typing method hiding is somewhat challenging for our system. We want to completely remove any mention of a method m' that is to be hidden, so that its name may be reused (possibly at a different type). But if we simply remove m' from the trait type, its requirements (which may not yet be fulfilled) will disappear from the trait as well, which is unsound. Our strategy is to remove m' from the trait's type, but transitively record its requirements in the inlining assumptions of any other provided method that invokes m'. In effect, we are doing a one-step path compression on the method call graph. The hide function achieves this result:

$$\text{hide}(\rho, m', \rho_{m'}) = \begin{cases} (\rho \uplus \rho_{m'}) \setminus m' & m' \in \rho \\ \rho & \text{otherwise} \end{cases}$$

[4]The issue of aliasing a recursive method is detailed in the dynamic semantics.

The type judgement for hiding simply applies hide to each provided method, dropping m':

$$\frac{\Gamma \vdash T \;:\; \langle\!\!|\; m : \tau_m @ \rho_m \;{}^{m \in \mathcal{M}} \;|\!\!\rangle \quad m' \in \mathcal{M} \qquad \theta = \langle\!\!|\; m : \tau_m @ \mathrm{hide}(\rho_m, m', \rho_{m'}) \;{}^{m \in \mathcal{M} \setminus m'} \;|\!\!\rangle \quad \Gamma \vdash \theta \text{ ok}}{\Gamma \vdash T \;\mathbf{hide}\; m' \;:\; \theta}$$

Notice that, if m' is not used by any other provided method in the trait, its requirements are dropped altogether. In this case, m' is dead code for the trait, and the method itself is eliminated in the dynamic semantics.

Finally, we have subclass formation. We first type the constructor, trait, and superclass. We then ensure that the trait's inlining assumptions about the class hold. A given type τ_l might be specified as part of the expected **super**-type, the expected **self**-type, and the actual superclass type; the typing judgement will only succeed if these specifications agree on the form of τ_l. Because of this requirement, we need only check the relationships between the various label sets to ensure that the class is well-typed:

$$\frac{\begin{array}{c} \Gamma,\ x : \tau \vdash e_{\text{cons}} \;:\; \tau_{\text{cons}} \quad \Gamma,\ x : \tau \vdash e_F \;:\; \{ f : \tau_f \;{}^{f \in \mathcal{F}} \} \\ \Gamma \vdash T \;:\; \theta \quad \Gamma \vdash C \;:\; \tau_{\text{cons}} \to \{\!|\; l : \tau_l \;{}^{l \in \mathcal{L}_C} \;|\!\} \quad \mathcal{F} \pitchfork \mathcal{L}_C \\ \mathcal{L} = \mathcal{L}_C \cup \mathcal{F} \cup \mathrm{dom}(\theta) \\ \langle l : \tau_l \;{}^{l \in \mathcal{L}_{\mathbf{super}}} \rangle = \mathrm{super}(\mathrm{Inl}(\theta)) \quad \mathcal{L}_{\mathbf{super}} \subset \mathcal{L}_C \\ \langle l : \tau_l \;{}^{l \in \mathcal{L}_{\mathbf{self}}} \rangle = \mathrm{self}(\mathrm{Inl}(\theta)) \quad \mathcal{L}_{\mathbf{self}} \subset \mathcal{L} \end{array}}{\begin{array}{c} \Gamma \vdash \lambda(x : \tau).(\mathbf{super}\; e_{\text{cons}}) \oplus e_F \;\mathbf{in} \\ T \;\mathbf{extends}\; C \;:\; \tau \to \{\!|\; l : \tau_l \;{}^{l \in \mathcal{L}} \;|\!\} \end{array}}$$

6 Dynamic semantics

We define evaluation with a big-step operational semantics. Evaluation judgments are written in the context of an environment E and a store S. Environments map names to trait, class, and expression values. Stores map addresses to object values, allowing objects to have mutable fields. Declaration evaluation yields a new environment and possibly a new store, while expression evaluation (which can occur in a declaration evaluation) yields an expression value and possibly a new store:

Program evaluation	$E,\ S \vdash P \longrightarrow ev,\ E',\ S'$
Declaration evaluation	$E,\ S \vdash D \longrightarrow E',\ S'$
Expression evaluation	$E,\ S \vdash e \longrightarrow ev,\ S'$

At the core of our calculus is a revised notion of trait values. The standard view of trait values as simple collections of named provided methods is insufficient to support trait privacy, and makes a realistic model of deep renaming difficult to achieve. We adopt Riecke and Stone's approach [RS02] and distinguish between internal method names (which we term *slots*) and external method names. In our model, trait values are collections of internally named provided methods, some of which may be externally named as well. To support deep renaming of required methods, we distinguish between internal and external names for them as well; the internal name of a required method is eventually assigned to the method that fulfills that requirement.

More formally, a dictionary ϕ is a finite partial function that maps method names to slots. A method suite value Mv maps slots to method values. A trait value tv is a method suite value together with dictionaries for its provided and required methods:

$$\begin{array}{rcll} \phi & ::= & \{r \mapsto i^{\;r \in \mathcal{R}}\} & \text{dictionary} \\ Mv & ::= & \{i \mapsto \mu v_i^{\;i \in \mathcal{I}}\} & \text{method suite value} \\ \mu v & ::= & [E;\ \phi_\mu;\ \lambda(x:\tau).e;\ \rho] & \text{method value} \\ tv & ::= & \langle\!\langle\, Mv;\ \phi_P;\ \phi_R \,\rangle\!\rangle & \text{trait value} \end{array}$$

To prove type soundness we will need to give types to trait values, so we track inlining assumptions in method values.

Method hiding and renaming only affect a trait value's dictionaries, *i.e.*, its external naming; its method suite remains unchanged.[5] Notice that a method value μv contains a dictionary ϕ_μ in addition to the standard closure over the lexical environment E. This dictionary, established during evaluation of trait formation, can be thought of as a closure over the trait's current external name environment. Unlike the trait itself, which has two dictionaries, each method closure contains only the single dictionary ϕ_μ; from the perspective of a particular method, there is no difference between provided and required methods, because by the time the method is invoked all required methods must have been provided. It is ϕ_μ that allows a method to remain coherent in the presence of method hiding and renaming.

Ultimately, the methods in a trait value will be inlined into a class value. Method dispatch is dictionary-based, but the dictionary used is *dependent on the location of the call* (this is the crux of Riecke and Stone's approach). More concretely: suppose we have a trait, `TFooBar`, which provides methods `foo` and `bar`. Further, suppose that `foo` invokes `bar`. When the trait is first formed, we assign slots to `foo` and `bar` and record these assignments in the ϕ_μ dictionaries for both methods. We can assume $\phi_\mu = \{\texttt{foo} \mapsto 1,\ \texttt{bar} \mapsto 2\}$ for both methods.[6] If we then rename `bar` to `baz`, the dictionary for the trait itself is changed to reflect this ($\phi_p = \{\texttt{foo} \mapsto 1,\ \texttt{baz} \mapsto 2\}$), but the dictionaries for `foo` and `bar` remain the same. Suppose we inline `TFooBar` into a class and instantiate an object `obj`. We are able to invoke `obj.baz` using a dictionary giving an external view of the object (similar to ϕ_P). But if we invoke `obj.foo`, what happens when we reach the call to **`self`**`.bar`, which no longer exists from the external viewpoint? The ϕ_μ dictionary associated with `foo` is used to discover the *slot* for `bar`, which will be the same as the slot for `baz`.

The previous example glosses over a few evaluation details, which are explained below. The main idea to keep in mind is the motivation for all the dictionary juggling: we need to know what actual method to invoke, given a method name and context, and we need to do this in the face of aliasing, renaming, hiding, and excluding. Figure 6 shows the previous example and others in more formal detail; it should be read in parallel with the evaluation rules. The remainder of this section gives a complete description of evaluation for traits and classes and describes the nonstandard portions of expression evaluation. Additional rules for expression evaluation appear in the appendix.

6.1 Trait evaluation

Trait evaluation judgments have the form $E \vdash T \longrightarrow tv$, since trait evaluation does not use or modify the store. We will often write $E \vdash T \longrightarrow \langle\!\langle\, {}^{\mathcal{I}}Mv;\ {}^{\mathcal{M}}\phi_P;\ {}^{\mathcal{R}}\phi_R \,\rangle\!\rangle$ to assert that T

[5] One of the rules for renaming changes the closures in the suite, but it leaves the slot assignments of the suite intact.

[6] In this example and others, we let $\mathcal{I}_U = \mathbb{Z}^+$, but formally $\mathcal{I}_U$ is held abstract.

We define

$$\begin{array}{lcl} \mu_{\texttt{foo}} & =_{def} & \texttt{foo}\,(x : \textbf{unit}) : \textbf{unit}\ \{\ \textbf{self}.\texttt{bar}()\ \} \\ \mu_{\texttt{bar}} & =_{def} & \texttt{bar}\,(x : \textbf{unit}) : \textbf{unit}\ \{\ \cdots\ \} \end{array}$$

and derive

$$\begin{array}{r} \emptyset,\ \emptyset \vdash \texttt{TFoo}\ =\ \llparenthesis\ \mu_{\texttt{foo}};\ \langle\!\langle \texttt{bar} : \textbf{unit} \to \textbf{unit} \rangle\!\rangle\ \rrparenthesis \longrightarrow E_1,\ \emptyset \\ E_1,\ \emptyset \vdash \texttt{TBar}\ =\ \llparenthesis\ \mu_{\texttt{bar}};\ \langle\!\langle\ \rangle\!\rangle\ \rrparenthesis \longrightarrow E_2,\ \emptyset \\ E_2,\ \emptyset \vdash \texttt{TFooBar}\ =\ \texttt{TFoo} + \texttt{TBar} \longrightarrow E_3,\ \emptyset \end{array}$$

where $E_3\ =$

$$\left\{ \begin{array}{l} \texttt{TFoo} \mapsto \llparenthesis\ \{1 \mapsto \mu v_{\texttt{foo}}\};\ \{\texttt{foo} \mapsto 1\};\ \{\texttt{bar} \mapsto 2\}\ \rrparenthesis, \\ \texttt{TBar} \mapsto \llparenthesis\ \{1 \mapsto \mu v_{\texttt{bar}}\};\ \{\texttt{bar} \mapsto 1\};\ \emptyset\ \rrparenthesis, \\ \texttt{TFooBar} \mapsto \left\langle\!\left| \begin{array}{l} \{1 \mapsto \mu v_{\texttt{foo}},\ 2 \mapsto \mu v_{\texttt{bar}}\}; \\ \{\texttt{foo} \mapsto 1,\ \texttt{bar} \mapsto 2\};\ \emptyset \end{array} \right|\!\right\rangle \end{array} \right\}$$

so that

$$\begin{array}{l} E_3 \vdash \texttt{TFooBar}\ \textbf{alias}\ \texttt{foo}\ \textbf{as}\ \texttt{fiz} \\ \longrightarrow \quad \left\langle\!\left| \begin{array}{l} \{1 \mapsto \mu v_{\texttt{foo}},\ 2 \mapsto \mu v_{\texttt{bar}},\ 3 \mapsto \mu v_{\texttt{foo}}\}; \\ \{\texttt{foo} \mapsto 1,\ \texttt{bar} \mapsto 2,\ \texttt{fiz} \mapsto 3\};\ \emptyset \end{array} \right|\!\right\rangle \end{array}$$

$$\begin{array}{l} E_3 \vdash \texttt{TFooBar}\ \textbf{rename}\ \texttt{bar}\ \textbf{to}\ \texttt{baz} \\ \longrightarrow \quad \left\langle\!\left| \begin{array}{l} \{1 \mapsto \mu v'_{\texttt{foo}},\ 2 \mapsto \mu v'_{\texttt{bar}}\}; \\ \{\texttt{foo} \mapsto 1,\ \texttt{baz} \mapsto 2\};\ \emptyset \end{array} \right|\!\right\rangle \end{array}$$

$$\begin{array}{ll} E_3 \vdash \texttt{TFooBar}\ \textbf{exclude}\ \texttt{bar} & \\ \longrightarrow & \llparenthesis\ \{1 \mapsto \mu v_{\texttt{foo}}\};\ \{\texttt{foo} \mapsto 1\};\ \{\texttt{bar} \mapsto 2\}\ \rrparenthesis \\ & =\ E_3(\texttt{TFoo}) \end{array}$$

$$\begin{array}{ll} E_3 \vdash \texttt{TFooBar}\ \textbf{hide}\ \texttt{bar} & \\ \longrightarrow & \llparenthesis\ \{1 \mapsto \mu v''_{\texttt{foo}},\ 2 \mapsto \mu v''_{\texttt{bar}}\};\ \{\texttt{foo} \mapsto 1\};\ \emptyset\ \rrparenthesis \end{array}$$

$$\begin{array}{l} E_3 \vdash (\texttt{TFooBar}\ \textbf{hide}\ \texttt{bar}) + \texttt{TBar} \\ \longrightarrow \quad \left\langle\!\left| \begin{array}{l} \{1 \mapsto \mu v''_{\texttt{foo}},\ 2 \mapsto \mu v''_{\texttt{bar}},\ 3 \mapsto \mu v_{\texttt{bar}}\}; \\ \{\texttt{foo} \mapsto 1,\ \texttt{bar} \mapsto 3\};\ \emptyset \end{array} \right|\!\right\rangle \end{array}$$

$$\begin{array}{ll} E_3 \vdash \texttt{TFooBar}\ \textbf{hide}\ \texttt{foo} & \\ \longrightarrow & \llparenthesis\ \{2 \mapsto \mu v_{\texttt{bar}}\};\ \{\texttt{bar} \mapsto 2\};\ \emptyset\ \rrparenthesis \\ & \approx\ E_3(\texttt{TBar}) \end{array}$$

Figure 6: Evaluation examples

evaluates to $\langle\!|\ Mv;\ \phi_P;\ \phi_R\ |\!\rangle$ with $\mathrm{dom}(Mv) = \mathcal{I}$, $\mathrm{dom}(\phi_P) = \mathcal{M}$, and $\mathrm{dom}(\phi_R) = \mathcal{R}$; we always have that $\mathcal{M} \pitchfork \mathcal{R}$. For slot manipulation, we will use a function NS ("new slots") which takes a set of method names and a set of slots, and a function FS ("fresh slots") which takes two sets of slots. NS yields a dictionary mapping each of the given names to a unique, new slot not in the given set of slots. FS yields a translation function φ which maps each of the slots in its first parameter to a unique slot not contained in its second parameter. In other words,

$$\frac{\phi = \mathrm{NS}(\mathcal{M},\ \mathcal{I})}{\mathrm{dom}(\phi) = \mathcal{M} \quad \mathrm{rng}(\phi) \pitchfork \mathcal{I} \quad \phi \text{ is one-to-one}} \qquad \frac{\varphi = \mathrm{FS}(\mathcal{I}_1,\ \mathcal{I}_2)}{\mathrm{dom}(\varphi) = \mathcal{I}_1 \quad \mathrm{rng}(\varphi) \pitchfork \mathcal{I}_2 \quad \varphi \text{ is one-to-one}}$$

For example, we might have

$$\begin{array}{rcl} \mathrm{NS}(\{m_1, m_2\},\ \{1,2,3\}) & = & \{m_1 \mapsto 4,\ m_2 \mapsto 5\} \\ \mathrm{FS}(\{1,4\},\ \{1,2,3\}) & = & \{1 \mapsto 4,\ 4 \mapsto 5\} \end{array}$$

Evaluating a trait formation expression to a trait value establishes the initial slot assignment for the provided method and each required (super- or self-) method using NS. Slots are not established for field requirements, which cannot be renamed in our model. The resulting dictionary is used as the external name closure in the constructed method values:

$$\frac{\phi_P = \mathrm{NS}(\{m\},\ \emptyset) \quad \phi_R = \mathrm{NS}(\mathrm{dom}(\rho) \setminus \mathcal{F}_U,\ \{\phi_P(m)\}) \quad Mv = \{\phi_P(m) \mapsto [E;\ \phi_P \cup \phi_R;\ \lambda(x:\tau_1).e;\ \rho]\ \}}{E \vdash \langle\!|\ m\ (x:\tau_1) : \tau_2\ \{e\};\ \rho\ |\!\rangle \longrightarrow \langle\!|\ Mv;\ \phi_P;\ \phi_R\ |\!\rangle}$$

Notice that the dictionaries ϕ_P and ϕ_R are joined as a single dictionary in the method value. An alternative presentation of trait values could maintain a single dictionary at the trait level, and determine which methods were actually provided by examining $\mathrm{dom}(Mv)$. For clarity and simplicity, we separate the dictionaries and maintain the invariants $\mathrm{rng}(\phi_P) \subseteq \mathrm{dom}(Mv)$ and $\mathrm{dom}(Mv) \pitchfork \mathrm{rng}(\phi_R)$. The set of slots used by a trait is $\mathrm{dom}(Mv) \cup \mathrm{rng}(\phi_R)$.

Aliasing a method does not simply map a new external name to the existing internal name for the method; if we alias m as m', we want the two methods to share implementations but to otherwise be independent, so that in particular we may later exclude m without impacting m'. Another concern is recursion: if m invokes itself, and we alias m as m', what should happen to the recursive call for m'? To see why this is important, consider that after aliasing m as m', we could exclude m from the trait and replace it with some other method. This is a somewhat thorny issue, because if we choose to have m' recurse on itself rather than invoking m, we have only dealt with *direct* recursion; if m recurses on itself indirectly via some other method, m' would still end up invoking m via that same method. To keep the semantics simple and consistent, aliasing m as m' will leave invocations of m as still calling m.

Aliasing has two cases, which we handle with different rules. A method can be aliased to fulfill a method requirement, in which case the slot assigned for the required method is used for the alias, and the requirement is removed from ϕ_R:

$$\frac{\begin{array}{c} E \vdash T \longrightarrow \langle\!|\ {}^{\mathcal{I}}Mv;\ {}^{\mathcal{M}}\phi_P;\ {}^{\mathcal{R}}\phi_R\ |\!\rangle \\ m \in \mathcal{M} \quad m' \notin \mathcal{M} \quad m' \in \mathcal{R} \\ \phi'_P = \phi_P[m' \mapsto \phi_R(m')] \\ Mv' = Mv[\phi_R(m') \mapsto Mv(\phi_P(m))] \end{array}}{E \vdash T\ \textbf{alias}\ m\ \textbf{as}\ m' \longrightarrow \langle\!|\ Mv';\ \phi'_P;\ \phi_R \setminus m'\ |\!\rangle}$$

If the aliasing operation does not fulfill a method requirement, we use NS to establish a new internal name for the alias:

$$\frac{\begin{array}{c} E \vdash T \longrightarrow \langle\!\langle\ ^{\mathcal{I}}Mv;\ ^{\mathcal{M}}\phi_P;\ ^{\mathcal{R}}\phi_R\ \rangle\!\rangle \\ m \in \mathcal{M} \quad m' \notin \mathcal{M} \quad m' \notin \mathcal{R} \\ \phi'_P = \phi_P \cup \mathrm{NS}(\{m'\},\ \mathcal{I} \cup \mathrm{rng}(\phi_R)) \\ Mv' = Mv[\phi'_P(m') \mapsto Mv(\phi_P(m))] \end{array}}{E \vdash T\ \mathbf{alias}\ m\ \mathbf{as}\ m' \longrightarrow \langle\!\langle\ Mv';\ \phi'_P;\ \phi_R\ \rangle\!\rangle}$$

As with aliasing, provided methods may be renamed to fulfill required methods. In addition, required methods may be renamed. We give three rules for renaming. To rename a provided method without fulfilling a method requirement, we remove its old external name from the ϕ_P dictionary and insert a new mapping from the new name to the existing slot. We also rename the method in the inlining assumptions for each provided method:

$$\frac{\begin{array}{c} E \vdash T \longrightarrow \langle\!\langle\ ^{\mathcal{I}}Mv;\ ^{\mathcal{M}}\phi_P;\ ^{\mathcal{R}}\phi_R\ \rangle\!\rangle \\ r \in \mathcal{M} \quad r' \notin \mathcal{M} \quad r' \notin \mathcal{R} \\ Mv = \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \rho_i]^{\ i \in \mathcal{I}}\} \\ \phi'_P = (\phi_P \setminus r)[r' \mapsto \phi_P(r)] \\ Mv' = \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \rho_i[r'/r]]^{\ i \in \mathcal{I}}\} \end{array}}{E \vdash T\ \mathbf{rename}\ r\ \mathbf{to}\ r' \longrightarrow \langle\!\langle\ Mv';\ \phi'_P;\ \phi_R\ \rangle\!\rangle}$$

The rule for renaming required methods is similar, but works on ϕ_R:

$$\frac{\begin{array}{c} E \vdash T \longrightarrow \langle\!\langle\ ^{\mathcal{I}}Mv;\ ^{\mathcal{M}}\phi_P;\ ^{\mathcal{R}}\phi_R\ \rangle\!\rangle \\ r \in \mathcal{R} \quad r' \notin \mathcal{M} \quad r' \notin \mathcal{R} \\ Mv = \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \rho_i]^{\ i \in \mathcal{I}}\} \\ \phi'_R = (\phi_R \setminus r)[r' \mapsto \phi_R(r)] \\ Mv' = \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \rho_i[r'/r]]^{\ i \in \mathcal{I}}\} \end{array}}{E \vdash T\ \mathbf{rename}\ r\ \mathbf{to}\ r' \longrightarrow \langle\!\langle\ Mv';\ \phi_P;\ \phi'_R\ \rangle\!\rangle}$$

Renaming a provided method to fulfill a method requirement is more complex. Unlike the situation for aliasing, we are not establishing a new, independent method, and thus the slot assignment for the original method must be retained. At the same time, the required method being fulfilled has its own slot assignment which must also be retained. Both of these assignments represent commitments made to the ϕ_μ dictionaries in the method closures for the trait. To fulfill the commitments, we slightly modify the promise by altering the target slots for ϕ_μ; we arbitrarily choose to drop the required method's slot in favor of the provided method's slot. The modification is performed by composing a translation function φ with the original ϕ_μ dictionaries. The translation is the identity function on slots except at the required method's slot, where it maps to the provided method's slot:

$$\frac{\begin{array}{c} E \vdash T \longrightarrow \langle\!\langle\ ^{\mathcal{I}}Mv;\ ^{\mathcal{M}}\phi_P;\ ^{\mathcal{R}}\phi_R\ \rangle\!\rangle \\ r \in \mathcal{M} \quad r' \notin \mathcal{M} \quad r' \in \mathcal{R} \\ Mv = \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \rho_i]^{\ i \in \mathcal{I}}\} \\ \phi'_P = (\phi_P \setminus r)[r' \mapsto \phi_P(r)] \\ \varphi = \{i \mapsto i,\ \phi_R(r') \mapsto \phi_P(r)\} \\ Mv' = \{i \mapsto [E_i;\ \varphi \circ \phi_i;\ e_i;\ \rho_i[r'/r]]^{\ i \in \mathcal{I}}\} \end{array}}{E \vdash T\ \mathbf{rename}\ r\ \mathbf{to}\ r' \longrightarrow \langle\!\langle\ Mv';\ \phi'_P;\ \phi_R \setminus m'\ \rangle\!\rangle}$$

To support hiding and excluding methods, and to properly type trait values, we need to drop unreachable hidden methods and unused required methods. The judgment $Mv \vdash i$ uses i' says that within the method suite Mv, the method in slot i' is directly mentioned by the method in slot i:

$$\frac{Mv(i) = [E;\ \phi;\ e;\ \rho] \qquad i' \in \mathrm{rng}(\phi)}{Mv \vdash i \text{ uses } i'}$$

The auxiliary judgement form $tv \hookrightarrow tv'$ rewrites a trait value after "dead-code elimination." The judgement collects all of the slots possibly used by the publicly provided methods, using the reflexive, transitive closure of the uses judgment. This allows the requirements of hidden methods to be included in the determination. The method suite and required method dictionary are then restricted to include only the necessary slots; the notation $\phi_R^{-1}(\mathcal{I})$ signifies the inverse image of $\mathcal{I}$ under ϕ_R:

$$\frac{\begin{array}{c} Mv = \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \rho_i]^{\ i \in \mathrm{dom}(Mv)}\} \\ \mathcal{I} = \{i \mid j \in \mathrm{rng}(\phi_P) \text{ and } Mv \vdash j \text{ uses}^* \ i\} \qquad \mathcal{R} = \phi_R^{-1}(\mathcal{I}) \end{array}}{\langle\!|\ Mv;\ \phi_P;\ \phi_R\ |\!\rangle \hookrightarrow \langle\!|\ Mv \cap \mathcal{I};\ \phi_P;\ \phi_R \cap \mathcal{R}\ |\!\rangle}$$

Hiding a method removes it from the provided method dictionary. It also updates the inlining assumptions of all provided methods, using the hide function from the static semantics:

$$\frac{\begin{array}{c} E \vdash T \longrightarrow \langle\!|\ {}^{\mathcal{I}}Mv;\ {}^{\mathcal{M}}\phi_P;\ {}^{\mathcal{R}}\phi_R\ |\!\rangle \\ m \in \mathcal{M} \quad j = \phi_P(m) \quad Mv = \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \rho_i]^{\ i \in \mathcal{I}}\} \\ Mv' = \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \mathrm{hide}(\rho_i, m, \rho_j)]^{\ i \in \mathcal{I}}\} \\ \langle\!|\ Mv';\ \phi_P \setminus m;\ \phi_R\ |\!\rangle \hookrightarrow tv \end{array}}{E \vdash T \ \mathbf{hide}\ m \longrightarrow tv}$$

A method m hidden by this operation may still be available in the method suite, via its internal name: previously established methods can gain access to m by looking up the slot for m in the ϕ_μ dictionary bundled with their closure. If m is not used elsewhere in the trait, however, it is dropped.

To exclude a method m from a trait, it must be removed from *both* ϕ_P and Mv. Because other methods in the trait may reference m, we add m to ϕ_R, maintaining the internal name for m; dead-code elimination will remove this spurious requirement, and possibly others, if m is not actually needed:

$$\frac{\begin{array}{c} E \vdash T \longrightarrow \langle\!|\ {}^{\mathcal{I}}Mv;\ {}^{\mathcal{M}}\phi_P;\ {}^{\mathcal{R}}\phi_R\ |\!\rangle \quad m \in \mathcal{M} \\ \langle\!|\ Mv \setminus \phi_P(m);\ \phi_P \setminus m;\ \phi_R[m \mapsto \phi_P(m)]\ |\!\rangle \hookrightarrow tv \end{array}}{E \vdash T \ \mathbf{exclude}\ m \longrightarrow tv}$$

The most complex trait operation is composition. The two composed traits must have disjoint external names for their provided methods, but there may be considerable overlap in their internal naming, so we must adjust slot assignments accordingly. We use a technique similar to the one described for renaming, and treat the slot assignments of one of the traits as authoritative, creating a translation φ to adjust the other trait's dictionaries:

$$\frac{\begin{array}{c} E \vdash T_1 \longrightarrow \langle\!| \, {}^{\mathcal{I}_1} Mv_1;\ {}^{\mathcal{M}_1}\phi_{P_1};\ {}^{\mathcal{R}_1}\phi_{R_1} \,|\!\rangle \\ E \vdash T_2 \longrightarrow \langle\!| \, {}^{\mathcal{I}_2} Mv_2;\ {}^{\mathcal{M}_2}\phi_{P_2};\ {}^{\mathcal{R}_2}\phi_{R_2} \,|\!\rangle \\ \mathcal{M}_1 \pitchfork \mathcal{M}_2 \quad Mv_2 = \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \rho_i]^{\ i \in \mathcal{I}_2}\} \\ \varphi_P = \{\phi_{P_2}(m) \mapsto \phi_{R_1}(m)^{\ m \in \mathcal{M}_2 \cap \mathcal{R}_1}\} \\ \varphi_R = \{\phi_{R_2}(m) \mapsto \phi_{P_1}(m)^{\ m \in \mathcal{R}_2 \cap \mathcal{M}_1}\} \\ \mathcal{I}_1' = \mathcal{I}_1 \cup \mathrm{rng}(\phi_{R_1}) \quad \mathcal{I}_2' = \mathcal{I}_2 \cup \mathrm{rng}(\phi_{R_2}) \\ \varphi_F = \mathrm{FS}(\mathcal{I}_2' \setminus \mathrm{dom}(\varphi_R \cup \varphi_P),\ \mathcal{I}_1') \\ \varphi = \varphi_P \cup \varphi_R \cup \varphi_F \quad \phi_P = \phi_{P_1} \cup (\varphi \circ \phi_{P_2}) \\ \phi_R = (\phi_{R_1} \cup (\varphi \circ \phi_{R_2})) \setminus (\mathcal{M}_1 \cup \mathcal{M}_2) \\ Mv = Mv_1 \cup \{\varphi(i) \mapsto [E_i;\ \varphi \circ \phi_i;\ e_i;\ \rho_i]^{\ i \in \mathcal{I}_2}\} \end{array}}{E \vdash T_1 + T_2 \longrightarrow \langle\!| \, Mv;\ \phi_P;\ \phi_R \,|\!\rangle}$$

The construction of the translation φ is performed in three steps: we construct φ_P for the methods provided in T_2 that fulfill methods required by T_1; we construct φ_R for the methods required in T_2 that are provided by T_1; finally, we construct φ_F to map all remaining slot assignments from T_2 to fresh slots that do not occur in T_1. A new method suite Mv joins the method suite from T_1 and an adjusted method suite for T_2 that reflects the translated slot assignments.

6.2 Class evaluation

Class evaluation results in an evaluated constructor, a method suite, and a dictionary into that method suite:

cv	$::=$	$\{\!| \, \lambda v;\ Mv;\ \phi_C \,|\!\}$	class value
λv	$::=$	$[E;\ \lambda(x:\tau).e]$	function value

The judgment form for class evaluation is written $E \vdash C \longrightarrow cv$; as with traits, the store is not used when evaluating a class expression. We evaluate **nil** to the empty class value, writing $\{\}$ for the empty field record:

$$\frac{}{E \vdash \mathbf{nil} \longrightarrow \{\!| \, [\emptyset;\ \lambda x.\{\}];\ \emptyset;\ \emptyset \,|\!\}}$$

Handling inheritance requires us to deal with super-invocations. Since class methods may be overridden, and hence no longer accessible from the class's method suite, we bind super-invocations to new, hidden provided methods. The judgment form $cv \vdash tv \Longrightarrow Mv$ extends the method suite in tv to a new method suite that provides the relevant super-methods from cv:

$$\frac{Mv' = Mv \cup \{\phi_R(s) \mapsto Mv_C(\phi_C(s))^{\ s \in \mathrm{dom}(\phi_R) \cap \mathcal{S}_U}\}}{\{\!| \, \lambda v_{\mathbf{super}};\ Mv_C;\ \phi_C \,|\!\} \vdash \langle\!| \, Mv;\ \phi_P;\ \phi_R \,|\!\rangle \Longrightarrow Mv'}$$

Note that this rewriting does not create any slots; since super-methods are treated as requirements, we simply use the slots from the required method dictionary.

Evaluation of inheritance is similar to evaluation of trait composition: we are reconciling two method suites with incompatible slot assignments. In this case, however, a method with the same external name may be provided by both the trait and the superclass. We retain the class's slot assignment for the method, but use the trait's implementation, thereby overriding the method:

$$\frac{\begin{array}{c} E \vdash T \longrightarrow tv \quad tv = \langle\!| \ ^{\mathcal{I}}Mv;\ ^{\mathcal{M}}\phi_P;\ ^{\mathcal{R}}\phi_R \ |\!\rangle \\ E \vdash C \longrightarrow cv \quad cv = \{\!| \ \lambda v_{\mathbf{super}};\ Mv_C;\ \phi_C \ |\!\} \\ cv \vdash tv \Longrightarrow \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \rho_i]^{\ i\in\mathcal{I}'}\} \\ \varphi_P = \{\phi_P(m) \mapsto \phi_C(m)^{\ m\in\mathcal{M}\cap\mathrm{dom}(\phi_C)}\} \\ \varphi_R = \{\phi_R(m) \mapsto \phi_C(m)^{\ m\in\mathcal{R}\cap\mathrm{dom}(\phi_C)}\} \\ \varphi_F = \mathrm{FS}\big(\mathcal{I}' \setminus \mathrm{dom}(\varphi_P),\ \mathrm{dom}(Mv_C)\big) \\ \varphi = \varphi_P \cup \varphi_R \cup \varphi_F \\ Mv'_T = \{\varphi(i) \mapsto [E_i;\ \varphi \circ \phi_i;\ e_i;\ \rho_i]^{\ i\in\mathcal{I}'}\} \\ Mv'_C = (Mv_C \setminus \mathrm{rng}(\varphi_P)) \cup Mv'_T \\ \mathit{super} \notin \mathrm{dom}(E) \quad E_{\mathit{cons}} = E[\mathit{super} \mapsto \lambda v_{\mathbf{super}}] \\ \lambda v_{\mathit{cons}} = [E_{\mathit{con}};\ \lambda(x:\tau).(\mathit{super}\ e_{\mathit{cons}}) \oplus e_F] \end{array}}{\begin{array}{c} E \vdash \lambda(x:\tau).(\mathbf{super}\ e_{\mathrm{cons}}) \oplus e_F\ \mathbf{in} \\ T\ \mathbf{extends}\ C \longrightarrow \{\!| \ \lambda v_{\mathit{cons}};\ Mv'_C;\ \phi_C \cup (\varphi \circ \phi_P) \ |\!\} \end{array}}$$

In reading the above rule, recall that $\mathrm{rng}(\phi_R) \pitchfork \mathcal{I}$. We also have that $\mathrm{dom}(\varphi_R) \pitchfork \mathcal{I}'$, because no super-method name (**`super`**$.m$) can appear in ϕ_C.

6.3 Object instantiation and method dispatch

Expression evaluation is written $E,\ S \vdash e \longrightarrow ev,\ S'$. Most of the rules for expression evaluation are standard, but we describe the rules related to object instantiation and method dispatch to highlight the role that ϕ_μ dictionaries play. The remaining expression evaluation rules can be found in Appendix C.

An object value is a field record value paired with a method suite:

ov	$::=$	$\langle fv;\ Mv\rangle$	object value
fv	$::=$	$\{f = ev_f{}^{\ f\in\mathcal{F}}\}$	field record value
ev	$::=$	$(a,\ \phi)$	object reference
	$\mid$	λv	function value
	$\mid$	fv	field record value
	$\mid$	$()$	unit value

Evaluating an object instantiation takes a class and a constructor parameter and updates the store to map a new address a to a new object value. The evaluation yields the address a paired with the class's dictionary ϕ_C:

$$\frac{\begin{array}{l} E \vdash C \longrightarrow \{\!| \ [E_F;\ \lambda x.e_F];\ Mv;\ \phi_C \ |\!\} \\ E,\ S \vdash e \longrightarrow ev,\ S_1 \\ E_F[x \mapsto ev],\ S_1 \vdash e_F \longrightarrow fv,\ S_2 \quad a \notin \mathrm{dom}(S_2) \end{array}}{E,\ S \vdash \mathbf{new}\ C\ e \longrightarrow (a,\ \phi_C),\ S_2[a \mapsto \langle fv;\ Mv\rangle]}$$

To evaluate a method dispatch $e.m$, we first evaluate e to an address a and dictionary ϕ. We look up the object value associated with the address, then use the dictionary to index into the object's method suite at m, yielding the closure for the method m. The evaluation results in a function value equivalent to m's closure: the environment of the function value takes $self$ to (a, ϕ_μ) so that self-invocations within m have m's view of the class:

$$\frac{E,\ S \vdash e \longrightarrow (a,\ \phi),\ S' \quad S'(a) = \langle fv;\ Mv\rangle \qquad Mv(\phi(m)) = [E_m;\ \phi_m;\ \lambda(x:\tau).e_m;\ \rho_m]}{E,\ S \vdash e.m \longrightarrow [E_m[\mathit{self} \mapsto (a,\ \phi_m)];\ \lambda(x:\tau).e_m],\ S'}$$

To support super-method invocation, we invoke the super-method name in the context of **self**:

$$\frac{E,\ S \vdash \mathbf{self}.(\mathbf{super}.m) \longrightarrow ev,\ S'}{E,\ S \vdash \mathbf{super}.m \longrightarrow ev,\ S'}$$

This approach works because super-method names are indexed in the dictionary for each method, and super-method implementations are provided, with the appropriate slot, during class formation. Notice that this rule does *not* apply to the constructor for a class, which does not invoke a super-method but invokes "super" itself.

Finally, the evaluation of **self** is a simple lookup in the current environment.

$$\frac{}{E,\ S \vdash \mathbf{self} \longrightarrow E(\mathit{self}),\ S}$$

7 Type soundness

We close the formal discussion of our system by introducing a final batch of judgments and proving type soundness.

One important issue for type soundness is typing for link-time and run-time values. We must give types to trait values, class values, expression values, object addresses, environments, and stores. Because stores may be cyclic, we cannot type addresses via a recursive examination. We follow the standard technique and introduce a store typing Σ, which is a map from object addresses to object types. The intuition behind this approach is that well-typed programs will always store values of the same type in a given address, so we need only type locations when they are first introduced. Run-time typing judgments are given in terms of store typings, and thus have the form $\Sigma \vdash \Box\ :\ \Box$.

To type a store S, each of the values within the store are typechecked against the typing Σ for the whole store:

$$\frac{\Sigma \vdash \mathrm{ok} \qquad \mathrm{dom}(\Sigma) = \mathrm{dom}(S) \qquad \forall a \in \mathrm{dom}(S)\ \Sigma \vdash S(a)\ :\ \Sigma(a)}{\vdash S\ :\ \Sigma}$$

The judgment $\Sigma \vdash E\ :\ \Gamma$ says that environment E has type Γ, as in the following judgment for trait values within an environment:

$$\frac{\Sigma \vdash E\ :\ \Gamma \qquad \Sigma \vdash_{(\vec{\alpha})} tv\ :\ \theta \qquad t \notin \mathrm{dom}(E)}{\Sigma \vdash E[t \mapsto (\vec{\alpha})tv]\ :\ \Gamma,\ t : \Lambda(\vec{\alpha}).\theta}$$

Note the subscript $(\vec{\alpha})$ in the trait run-time typing judgment. This mechanism makes abstracted type variables available in the context when typing trait method values.

Most of the complexity of run-time typing stems from the use of dictionaries in run-time values. We introduce a mapping σ from slots to types, which serves as a type for method suites:

$$\sigma \quad ::= \quad \{i \mapsto \tau^{\ i \in \mathcal{I}}\} \quad \text{method suite type}$$

Method suites within classes, traits, and object values are typed under store assumptions Σ, along with type variables $\vec{\alpha}$ (for the trait case) and a record type τ_{fields} giving assumptions about fields. Notice that σ is constrained so that it gives a type to each method value in the suite, and so that each method value successfully types under it:

$$\frac{\begin{array}{c}\operatorname{dom}(Mv) = \operatorname{dom}(\sigma) \\ Mv = \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \rho_i]^{\ i \in \operatorname{dom}(Mv)}\} \\ \forall i \in \operatorname{dom}(Mv) \quad \Sigma \vdash_{(\vec{\alpha});\sigma;\tau_{\text{fields}}} Mv(i) \ :\ \sigma(i)\end{array}}{\Sigma \vdash_{(\vec{\alpha});\tau_{\text{fields}}} Mv \ :\ \sigma}$$

An additional judgment imposes a particular dictionary's view on a method suite type, yielding an object type taking names to types:

$$\frac{}{\phi \vdash \sigma \ :\ \langle m : \sigma(\phi(m))^{\ m \in \operatorname{dom}(\phi)}\rangle}$$

Recall that a method value has the form μv. We use the static typing judgment for the method body, and use τ_{fields} and σ to construct the appropriate view of $\mathbf{self}$ and $\mathbf{super}$:

$$\frac{\begin{array}{c}\Sigma \vdash E \ :\ \Gamma \qquad \tau_{\text{fields}} = \{f : \tau_f^{\ f \in \mathcal{F}}\} \\ (\phi_\mu \cap \mathcal{M}_U) \vdash \sigma \ :\ \tau'_{\mathbf{self}} \quad (\phi_\mu \cap \mathcal{S}_U) \vdash \sigma \ :\ \tau_{\mathbf{super}} \\ \tau_{\mathbf{self}} = \tau'_{\mathbf{self}} \uplus \langle f : \tau_f^{\ f \in \mathcal{F}}\rangle \\ \Gamma,\ \mathit{super} : \tau_{\mathbf{super}},\ \mathit{self} : \tau_{\mathbf{self}},\ x : \tau \vdash e \ :\ \tau'\end{array}}{\Sigma \vdash_{(\vec{\alpha});\sigma;\tau_{\text{fields}}} [E;\ \phi_\mu;\ \lambda(x : \tau).e;\ \rho] \ :\ \tau \to \tau'}$$

With method value and method suite typings in hand, typing trait values is fairly straightforward. We construct the trait type based on the publicly provided methods, using the inlining assumptions ρ_i stored with method values and a method suite typing σ. Inlining assumptions ρ for the whole trait are formed, and checked against the method suite type and the field assumptions used to check methods:

$$\frac{\begin{array}{c}Mv = \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \rho_i]^{\ i \in \operatorname{dom}(Mv)}\} \\ \theta = \langle\!| \ m : \sigma(\phi_P(m))@\rho_{\phi_P(m)}^{\ m \in \operatorname{dom}(\phi_P)} \ |\!\rangle \\ \rho = \operatorname{Inl}(\theta) \qquad \phi = \phi_P \cup \phi_R \\ \tau_{\text{fields}} = \{f : \rho(f)^{\ f \in \operatorname{dom}(\rho) \cap \mathcal{F}_U}\} \\ \rho(r) = \sigma(\phi(r)) \quad \forall r \in \operatorname{dom}(\rho) \setminus \mathcal{F}_U \\ \Sigma \vdash_{(\vec{\alpha});\tau_{\text{fields}}} Mv \ :\ \sigma\end{array}}{\Sigma \vdash_{(\vec{\alpha})} \langle\!| \ Mv;\ \phi_P;\ \phi_R \ |\!\rangle \ :\ \theta}$$

Class value typing is performed by determining the fields τ_{fields} available in the class, checking the class's methods under these assumptions to produce a method suite type σ, then applying the class dictionary ϕ_C to achieve the right view of the class's methods:

$$\frac{\begin{array}{c}\Sigma \vdash \lambda v \ :\ \tau \to \tau_{\text{fields}} \quad \tau_{\text{fields}} = \{f : \tau_f^{\ f \in \mathcal{F}}\} \\ \Sigma \vdash_{();\tau_{\text{fields}}} Mv \ :\ \sigma \qquad \phi_C \vdash \sigma \ :\ \langle l : \tau_l^{\ l \in \mathcal{L}}\rangle\end{array}}{\Sigma \vdash \{\!| \ \lambda v;\ Mv;\ \phi_C \ |\!\} \ :\ \tau \to \{\!| \ l : \tau_l^{\ l \in \mathcal{L}},\ f : \tau_f^{\ f \in \mathcal{F}} \ |\!\}}$$

Recall that a store S takes object addresses to object values, which have the form $\langle fv;\ Mv\rangle$. The type of an object value is a method suite type σ and a field record type τ_{fields}:

$$\frac{\Sigma \vdash fv \ :\ \tau_{\text{fields}} \qquad \Sigma \vdash_{();\tau_{\text{fields}}} Mv \ :\ \sigma}{\Sigma \vdash \langle fv;\ Mv\rangle \ :\ \sigma, \tau_{\text{fields}}}$$

In our system, object addresses are always paired with a dictionary used to view the object. Object addresses are therefore typed by looking up the object value type for the address and applying the dictionary:

$$\frac{\begin{array}{c}\Sigma \vdash \text{ok} \qquad \Sigma(a) = \sigma,\ \{f : \tau_f{}^{f\in\mathcal{F}}\} \\ \phi \vdash \sigma \ :\ \langle l : \tau_l{}^{l\in\mathcal{L}}\rangle\end{array}}{\Sigma \vdash (a, \phi) \ :\ \langle l : \tau_l{}^{l\in\mathcal{L}},\ f : \tau_f{}^{f\in\mathcal{F}}\rangle}$$

The remaining few rules are standard, and can be found in the appendix.

Proving soundness

Type preservation is easy to state for a big-step semantics, but progress is much harder; the difficulty of stating a progress property is usually held as a drawback of the big-step style. The essential problem is that, for a big-step semantics, non-termination and WRONG are indistinguishable. In particular, if we do not have $\emptyset,\ \emptyset \vdash P \longrightarrow ev,\ E,\ S$, it could either be that P diverges or that P goes wrong. To overcome this problem, we follow [FR04] and introduce a *height function* $H_{E,\ S}$, which gives the height of the derivation tree for a program under E and S. We have the following definition:

Definition 1 (Divergence)**:**
We say a program P *diverges* if there is no n such that $H_{\emptyset,\ \emptyset}(P) = n$.

If $E,\ S \vdash P \longrightarrow ev$ then $H_{E,\ S}(P) = n$, where n is the height of the derivation tree for the judgment. If P diverges in the context of E and S, then $H_{E,\ S}(P)$ diverges as well. Most importantly, if P does not diverge but still does not evaluate to any ev under E and S (*i.e.*, P goes WRONG), then $H_{E,\ S}(P)$ *converges*, and measures the height of the evaluation derivation up to the point that P went wrong. For example,

$$H_{E,\ S}(t = (\vec{\alpha})T;\ P) \ = $$

$$1 + \begin{cases} H_{E',\ S}(P) & \text{if } E,\ S \vdash t = (\vec{\alpha})T \longrightarrow E',\ S \\ 1 & \text{otherwise} \end{cases}$$

Thus, the height function allows us to distinguish between non-termination and stuck states, without having explicit WRONG transitions. The full definition of the height function can be found in Figure 7 in the appendix. Using this height function, we can state and prove the following soundness result:

Theorem 2 (Type soundness)**:**
If $\emptyset \vdash_P P \ :\ \tau$ then either P diverges or there exist a store typing Σ, a store S, a context Γ, an environment E, a type τ', and an expression value ev such that $\Sigma \vdash E \ :\ \Gamma$ and $\vdash S \ :\ \Sigma$ and $\emptyset,\ \emptyset \vdash P \longrightarrow ev,\ E,\ S$ and $\Sigma \vdash ev \ :\ \tau'$ and $\emptyset \vdash \tau' \ <:\ \tau$.

This result says that a well-typed, *terminating* program P does not go wrong. In particular, P will evaluate to a result that improves on its static type. While this is a weaker statement than soundness for a small-step semantics, a characterization of terminating programs is sufficient for our purposes with this calculus.

8 Related work

Traits were originally proposed as a mechanism for SMALLTALK by Schärli *et al.* [SDNB03]. In addition to studying the language design and methodological issues, they also developed a formal model for their system [SDN$^+$02]. The most important difference between our system and this original work is that we are working in a strongly-typed setting, instead of an untyped language. Another major point of difference is that we have abandoned the *flattening* property [NDS06] in favor of supporting deep renaming and private methods in traits.[7] In this sense our system is not a conservative extension of traditional class-based designs, but the meta-programming techniques enabled by our system provide an argument for a dictionary-based semantics. Further argument for this approach can be found in Riecke and Stone's paper [RS02].

The introduction of traits for SMALLTALK has prompted a flurry of work on traits for statically-typed languages. Fisher and Reppy developed the first formal model of traits in a statically typed setting [FR04]. This model was subsequently extended to support polymorphic traits (key for examples such as the synchronized readers) and stateful objects [FR03]. This extended trait calculus was the starting point for the system in this paper. We have made a number of refinements in the static semantics. Our calculus tracks method requirements on a per-method basis, which provides more accuracy when excluding methods. We have also unified the handling of methods and super methods in the requirements by introducing a separate namespace for super methods (*e.g.* $\mathbf{super}.m$). While a simple trick, this technique streamlines the static semantics significantly. We have reformulated the link-time and dynamic semantics of method binding using Riecke-Stone dictionaries, which allows the support for the deep operations of renaming and hiding at the trait level. These last two are our most significant additions to the Fisher-Reppy calculus.

Smith and Drossopoulou recently described a family of three different extensions of JAVA with traits [SD05]. The first of these, $Chai_1$, defers all checking until traits have been included in a class. The second, $Chai_2$, adds trait types and is similar in expressiveness to the Fisher-Reppy trait calculus, with a couple of exceptions. Like JAVA, $Chai_2$ uses nominal subtyping, instead of structural typing, and does not have polymorphic traits. The differences between our work the Fisher-Reppy calculus apply to $Chai_2$ as well. $Chai_3$ extends $Chai_2$ by allowing traits to be replaced at runtime. This feature is orthogonal to the focus of our work, but could be added to our system.

Another proposed design of traits for JAVA is *Featherweight Trait* JAVA (FTJ) [LS04], which adds traits to Featherweight JAVA [IPW01]. This system is fairly similar to the Fisher-Reppy trait calculus (with some technical differences), and does not support either deep renaming or private methods at the trait level.

There are strong similarities between traits and *mixins* [BC90, FKF98, OAC$^+$04], which are another mechanism designed to give many of the benefits of multiple inheritance without

[7] Strictly speaking, one has to reformulate the flattening property for our system, since we do not have a mechanism for defining methods directly in a class.

the complications. The main difference between mixins and traits is that mixins force a linear order in their composition (it is this order that avoids the complexities of the diamond property). This linear order introduces fragility problems and may make code maintenance more difficult [SDNB03]. Mixin mechanisms must also deal with constructor functions, which can be another source of fragility, since it is hard to predict what the interface of the super-class constructor will be. A solution to this problem is to define the mixin's constructors at the point of mixin application [ALZ03]. This issue does not affect traits, since they are not defining or initializing object state. Personalities are another trait-like mechanism designed for JAVA, although they are much more limited in their expressiveness [Bla98] and they do not have a formal model. The language Scala has a special form of abstract class called a trait class [OAC^{+}04]. Trait classes are used as mixins and also to support family polymorphism, but they do not support the trait operations such as exclusion, or our deep renaming.

Bracha's Jigsaw framework is often cited as the first formal account of mixins [Bra92]. Like our calculus, and unlike the other trait systems discussed above, Jigsaw supports deep renaming (Bracha calls it global renaming) and method hiding. His system also has a static binding mechanism (called freezing). Bracha gave a dynamic semantics and a type system for Jigsaw, but did not prove type soundness. While it is possible that our metaprogramming idiom could be applied to mixins in the Jigsaw framework, we believe that traits are a better fit.

Examples like the `TSync` trait closely resemble the use of aspects [KLM^{+}97] to specify "cross-cutting" concerns. While traits have always had some overlap with aspects, trait-based metaprogramming brings the two even closer. Both traits and aspects are specified outside of the classes to which they are applied; the primary difference between the two is how they are applied to classes. Aspects specify points of application via *pointcuts*, which pick out *join points* in the target classes; hence, aspects are in control of their own application. On the other hand, traits are completely inert unless they are explicitly inlined during class formation, leaving control to the class implementor.

9 Conclusion

Traits provide a promising mechanism for constructing class hierarchies from reusable components. We have introduced two new trait operations, method *hiding* and *renaming*, which are the *deep* counterparts to method exclusion and aliasing. These operations provide new ways of resolving conflicts in trait composition. Furthermore, they support privacy at the trait level and trait-based metaprogramming. Our formal model gives a detailed semantics to these operations in a statically typed setting, while also improving the granularity of the type system over previous calculi. There are several remaining questions for a concrete language design built around our calculus, which we briefly raise:

- In a language with a rich module system, it is not yet clear how traits should interact with features such as signatures and functors. One can imagine using signature ascription to implement method hiding in traits (as done in MOBY at the class level [FR99]).
- While the type system we have presented in this paper is flexible, it is also verbose, since each method provided by a trait specifies its own requirements. This granular type information can be inferred from a trait definition, but introducing traits into a module system

will require a programmer to explicitly write down trait signatures, and it is not clear what form such signatures should take.

- The most important questions posed by our work regard the design of a trait metalanguage. At the least, such a language should include an explicit abstraction mechanism for trait method names, but the design space is large and essentially unexplored. We plan to first implement the trait calculus of this paper in the language MOBY, which will allow us to gain experience with trait programming and manual trait-based metaprogramming. We hope that programming experience will lead to the recognition of patterns and idioms that can then be codified into a trait metalanguage.

Acknowledgments We would like to thank Kathleen Fisher, who provided helpful feedback on an early draft and insight into the type theory for the paper. The anonymous reviewers gave thorough feedback on the formal system and good insights on the presentation; their work is appreciated.

References

[ALZ03] Ancona, D., G. Lagorio, and E. Zucca. Jam—designing a java extension with mixins. *ACM Transactions on Programming Languages and Systems*, **25**(5), September 2003, pp. 641–712.

[BC90] Bracha, G. and W. Cook. Mixin-based inheritance. In *ECOOP'90*, New York, NY, October 1990. ACM, pp. 303–311.

[Bla98] Blando, L. Designing and programming with personalities. Master's dissertation, Northeastern University, Boston, MA, December 1998. Available as Technical Report NU-CCS-98-12.

[Bra92] Bracha, G. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. Ph.D. dissertation, University of Utah, March 1992.

[BSD03] Black, A. P., N. Schärli, and S. Ducasse. Applying traits to the Smalltalk collection classes. In *OOPSLA'03*, New York, NY, October 2003. ACM, pp. 47–64.

[FKF98] Flatt, M., S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *POPL'98*, New York, NY, January 1998. ACM, pp. 171–183.

[FR99] Fisher, K. and J. Reppy. The design of a class mechanism for Moby. In *PLDI'99*, New York, NY, May 1999. ACM, pp. 37–49.

[FR03] Fisher, K. and J. Reppy. Statically typed traits. *Technical Report TR-2003-13*, Dept. of Computer Science, U. of Chicago, Chicago, IL, December 2003.

[FR04] Fisher, K. and J. Reppy. A typed calculus of traits. In *FOOL11*, January 2004.

[IPW01] Igarashi, A., B. C. Pierce, and P. Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems*, **23**(3), 2001, pp. 396–450.

[KLM+97] Kiczales, G., J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka (eds.), *ECOOP'97*, vol. 1241 of *LNCS*, pp. 220–242. Springer-Verlag, New York, NY, 1997.

[LS04] Liquori, L. and A. Spiwack. Featherweight-trait java: A trait-based extension for FJ. *Technical Report RR-5247*, Institut National de Recherche en Informatique et en Automatique, June 2004.

[NDS06] Nierstrasz, O., S. Ducasse, and N. Schärli. Flattening traits. *Journal of Object Technology*, **5**(3), May 2006. (to appear).

[OAC+04] Odersky, M., P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. *The Scala Language Specification (Draft)*. Programming Methods Laboratory, EPFL, Switzerland, February 2004. Available from `lamp.epfl.ch/scala`.

[Qui04] Quitslund, P. J. Java traits — improving opportunities for reuse. *Technical Report CSE 04-005*, OGI School of Science & Engineering, September 2004.

[RS02] Riecke, J. G. and C. A. Stone. Privacy via subsumption. *Information and Computation*, **172**(1), January 2002, pp. 2–28. A preliminary version appeared in FOOL5.

[RT06] Reppy, J. and A. Turon. A foundation for trait-based metaprogramming. In *FOOL/WOOD'06*, January 2006.

[SD05] Smith, C. and S. Drossopoulou. Chai: Traits for Java-like languages. In *ECOOP'05*, LNCS, New York, NY, July 2005. Springer-Verlag.

[SDN+02] Schärli, N., S. Ducasse, O. Nierstrasz, R. Wuyts, and A. Black. Traits: The formal model. *Technical Report CSE 02-013*, OGI School of Science & Engineering, November 2002. (revised February 2003).

[SDNB03] Schärli, N., S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *ECOOP'03*, vol. 2743 of *LNCS*, New York, NY, July 2003. Springer-Verlag, pp. 248–274.

[Str94] Stroustrup, B. *The Design and Evolution of C++*. Addison-Wesley, Reading, MA, 1994.

A Syntactic forms

A.1 Calculus syntax

P	$::=$	$D;\ P \quad \mid \quad e$	program
D	$::=$	$t = (\vec{\alpha})T$	trait declaration
	$\mid$	$c = C$	class declaration
	$\mid$	$x = e$	expression declaration
T	$::=$	$t(\vec{\tau})$	polymorphic trait name
	$\mid$	$\langle\!\mid \mu;\ \rho \mid\!\rangle$	trait formation
	$\mid$	$T_1 + T_2$	trait composition
	$\mid$	$T\ \mathbf{exclude}\ m$	method exclusion
	$\mid$	$T\ \mathbf{hide}\ m$	method hiding
	$\mid$	$T\ \mathbf{alias}\ m\ \mathbf{as}\ m'$	method aliasing
	$\mid$	$T\ \mathbf{rename}\ r\ \mathbf{to}\ r'$	method renaming
μ	$::=$	$m\ (x : \tau_1) : \tau_2\ \{e\}$	method
ρ	$::=$	$\langle\!\langle r : \tau_r{}^{r \in \mathcal{R}} \rangle\!\rangle$	inlining assumptions
θ	$::=$	$\langle\!\mid m : \tau_m @ \rho_m{}^{m \in \mathcal{M}} \mid\!\rangle$	trait type
	$\mid$	$\Lambda(\vec{\alpha}).\theta$	polymorphic trait type
C	$::=$	c	class name
	$\mid$	$\mathbf{nil}$	empty class
	$\mid$	$I\ \mathbf{in}\ T\ \mathbf{extends}\ C$	subclass formation
I	$::=$	$\lambda(x : \tau).(\mathbf{super}\ e_1) \oplus e_2$	constructor
χ	$::=$	$\tau \to \{\!\mid l : \tau_l{}^{l \in \mathcal{L}} \mid\!\}$	class type
e	$::=$	x	variable
	$\mid$	$\lambda(x : \tau).e$	function abstraction
	$\mid$	$e_1\ e_2$	function application
	$\mid$	$\mathbf{new}\ C\ e$	object instantiation
	$\mid$	$\mathbf{self}$	host object
	$\mid$	$\mathbf{super}.m$	super-method dispatch
	$\mid$	$e.m$	method dispatch
	$\mid$	$e.f$	field selection
	$\mid$	$e_1.f := e_2$	field update
	$\mid$	$\{f = e_f{}^{f \in \mathcal{F}}\}$	field record
	$\mid$	$e_1 \oplus e_2$	field concatenation
	$\mid$	$()$	unit value
τ	$::=$	α	type variable
	$\mid$	$\langle l : \tau_l{}^{l \in \mathcal{L}} \rangle$	object type
	$\mid$	$\tau_1 \to \tau_2$	function type
	$\mid$	$\{f : \tau_f{}^{f \in \mathcal{F}}\}$	field record type
	$\mid$	$\mathbf{unit}$	unit type

A.2 Run-time value syntax

$$
\begin{array}{rcll}
tv & ::= & \langle\!|\ Mv;\ \phi_P;\ \phi_R\ |\!\rangle & \text{trait value} \\
\phi & ::= & \{r \mapsto i\ ^{r\in\mathcal{R}}\} & \text{dictionary} \\
Mv & ::= & \{i \mapsto \mu v_i\ ^{i\in\mathcal{I}}\} & \text{method suite value} \\
\mu v & ::= & [E;\ \phi_\mu;\ \lambda(x:\tau).e;\ \rho] & \text{method value} \\
cv & ::= & \{\!|\ \lambda v;\ Mv;\ \phi_C\ |\!\} & \text{class value} \\
ev & ::= & (a,\ \phi) & \text{object reference} \\
 & | & \lambda v & \text{function value} \\
 & | & fv & \text{field record value} \\
 & | & () & \text{unit value} \\
\lambda v & ::= & [E;\ \lambda(x:\tau).e] & \text{function value} \\
ov & ::= & \langle fv;\ Mv\rangle & \text{object value} \\
fv & ::= & \{f = ev_f\ ^{f\in\mathcal{F}}\} & \text{field record value}
\end{array}
$$

B Typing judgments

Context formation: $\boxed{\Gamma \vdash \text{ok}}$

$$
\frac{}{\emptyset \vdash \text{ok}} \qquad
\frac{\Gamma \vdash \text{ok} \quad \alpha \notin \Gamma}{\Gamma,\ \alpha \vdash \text{ok}} \qquad
\frac{\Gamma \vdash \theta\ \text{ok} \quad t \notin \text{dom}(\Gamma)}{\Gamma,\ t:\theta \vdash \text{ok}} \qquad
\frac{\Gamma \vdash \chi\ \text{ok} \quad c \notin \text{dom}(\Gamma)}{\Gamma,\ c:\chi \vdash \text{ok}} \qquad
\frac{\Gamma \vdash \tau\ \text{ok} \quad x \notin \text{dom}(\Gamma)}{\Gamma,\ x:\tau \vdash \text{ok}}
$$

Well-formed inlining assumptions: $\boxed{\Gamma \vdash \rho\ \text{ok}}$

$$
\frac{\Gamma \vdash \tau_r\ \text{ok for all}\ r \in \mathcal{R} \quad \tau_m = \tau_{\mathbf{super}.m}\ \text{for all}\ m\ \text{with}\ m \in \mathcal{R}, \mathbf{super}.m \in \mathcal{R}}{\Gamma \vdash \langle\!\langle r:\tau_r\ ^{r\in\mathcal{R}}\rangle\!\rangle\ \text{ok}}
$$

Well-formed trait types: $\boxed{\Gamma \vdash \theta\ \text{ok}}$

$$
\frac{\text{Inl}(\theta)\ \text{is defined}}{\Gamma \vdash \theta\ \text{ok}}
$$

Well-formed class types: $\boxed{\Gamma \vdash \chi\ \text{ok}}$

$$
\frac{\Gamma \vdash \tau\ \text{ok} \quad \Gamma \vdash \langle l:\tau_l\ ^{l\in\mathcal{L}}\rangle\ \text{ok}}{\Gamma \vdash \tau \to \{\!|\ l:\tau_l\ ^{l\in\mathcal{L}}\ |\!\}\ \text{ok}}
$$

Well-formed expression types: $\boxed{\Gamma \vdash \tau\ \text{ok}}$

$$
\frac{\Gamma \vdash \text{ok} \quad \alpha \in \Gamma}{\Gamma \vdash \alpha\ \text{ok}} \qquad
\frac{\Gamma \vdash \tau_1\ \text{ok} \quad \Gamma \vdash \tau_2\ \text{ok}}{\Gamma \vdash \tau_1 \to \tau_2\ \text{ok}} \qquad
\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \mathbf{unit}\ \text{ok}}
$$

$$
\frac{\Gamma \vdash \text{ok} \quad \Gamma \vdash \tau_l\ \text{ok} \quad \forall l \in \mathcal{L}}{\Gamma \vdash \langle l:\tau_l\ ^{l\in\mathcal{L}}\rangle\ \text{ok}} \qquad
\frac{\Gamma \vdash \text{ok} \quad \Gamma \vdash \tau_f\ \text{ok} \quad \forall f \in \mathcal{F}}{\Gamma \vdash \{f:\tau_f\ ^{f\in\mathcal{F}}\}\ \text{ok}}
$$

Subtyping: $\boxed{\Gamma \vdash \tau_1 \ <: \ \tau_2}$

$$\frac{\Gamma \vdash \tau \text{ ok}}{\Gamma \vdash \tau \ <: \ \tau} \qquad \frac{\Gamma \vdash \langle l : \tau_l \ ^{l \in \mathcal{L}_1} \rangle \text{ ok} \quad \mathcal{L}_2 \subset \mathcal{L}_1}{\Gamma \vdash \langle l : \tau_l \ ^{l \in \mathcal{L}_1} \rangle \ <: \ \langle l : \tau_l \ ^{l \in \mathcal{L}_2} \rangle} \qquad \frac{\Gamma \vdash \tau_2' \ <: \ \tau_1' \quad \Gamma \vdash \tau_1'' \ <: \ \tau_2''}{\Gamma \vdash \tau_1' \to \tau_1'' \ <: \ \tau_2' \to \tau_2''}$$

Program typing: $\boxed{\Gamma \vdash_P P \ : \ \tau}$

$$\frac{\Gamma \vdash D \Rightarrow \Gamma' \quad \Gamma' \vdash_P P \ : \ \tau}{\Gamma \vdash_P D; \ P \ : \ \tau} \qquad \frac{\Gamma \vdash e \ : \ \tau}{\Gamma \vdash_P e \ : \ \tau}$$

Declaration typing: $\boxed{\Gamma \vdash D \Rightarrow \Gamma'}$

$$\frac{t \notin \text{dom}(\Gamma) \quad \Gamma, \ \vec{\alpha} \vdash T \ : \ \theta}{\Gamma \vdash t = (\vec{\alpha})T \Rightarrow \Gamma, \ t : \Lambda(\vec{\alpha}).\theta} \qquad \frac{c \notin \text{dom}(\Gamma) \quad \Gamma \vdash C \ : \ \chi}{\Gamma \vdash c = C \Rightarrow \Gamma, \ c : \chi} \qquad \frac{x \notin \text{dom}(\Gamma) \quad \Gamma \vdash e \ : \ \tau}{\Gamma \vdash x = e \Rightarrow \Gamma, \ x : \tau}$$

Trait typing: $\boxed{\Gamma \vdash T \ : \ \theta}$

$$\frac{\begin{array}{c}\Gamma(t) = \Lambda(\vec{\alpha}).\theta \\ \Gamma \vdash \vec{\tau} \text{ ok} \quad |\vec{\tau}| = |\vec{\alpha}|\end{array}}{\Gamma \vdash t(\vec{\tau}) \ : \ \theta[\vec{\tau}/\vec{\alpha}]} \qquad \frac{\begin{array}{c}\mu = m \ (x : \tau_1) : \tau_2 \ \{e\} \quad \Gamma \vdash \rho \uplus \langle\!\langle m : \tau \rangle\!\rangle \text{ ok} \\ \Gamma, \ \textit{super} : \text{super}(\rho), \ \textit{self} : \text{self}(\rho) \vdash \mu \ : \ \tau\end{array}}{\Gamma \vdash \langle\!| \ \mu; \ \rho \ |\!\rangle \ : \ \langle\!| \ m : \tau @ \rho \ |\!\rangle}$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 \ : \ \theta_1 \quad \Gamma \vdash T_2 \ : \ \theta_2 \\ \text{dom}(\theta_1) \pitchfork \text{dom}(\theta_2) \quad \Gamma \vdash \theta_1 \uplus \theta_2 \text{ ok}\end{array}}{\Gamma \vdash T_1 + T_2 \ : \ \theta_1 \uplus \theta_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash T \ : \ \theta \quad m \in \theta \\ \Gamma \vdash \theta \setminus m \text{ ok}\end{array}}{\Gamma \vdash T \ \textbf{exclude} \ m \ : \ \theta \setminus m} \qquad \frac{\begin{array}{c}\Gamma \vdash T \ : \ \langle\!| \ m : \tau_m @ \rho_m \ ^{m \in \mathcal{M}} \ |\!\rangle \quad m' \in \mathcal{M} \\ \theta = \langle\!| \ m : \tau_m @ \text{hide}(\rho_m, m', \rho_{m'}) \ ^{m \in \mathcal{M} \setminus m'} \ |\!\rangle \quad \Gamma \vdash \theta \text{ ok}\end{array}}{\Gamma \vdash T \ \textbf{hide} \ m' \ : \ \theta}$$

$$\frac{\begin{array}{c}\Gamma \vdash T \ : \ \theta \quad m \in \theta \quad m' \notin \theta \\ \theta' = \theta \uplus \langle\!| \ m' : \theta(m) \ |\!\rangle \\ \Gamma \vdash \theta' \text{ ok}\end{array}}{\Gamma \vdash T \ \textbf{alias} \ m \ \textbf{as} \ m' \ : \ \theta'} \qquad \frac{\begin{array}{c}\Gamma \vdash T \ : \ \langle\!| \ m : \tau_m @ \rho_m \ ^{m \in \mathcal{M}} \ |\!\rangle \\ r, r' \notin \mathcal{F}_U \quad r' \notin \mathcal{M} \quad r \in \rho_m \text{ for some } \rho_m \in \mathcal{M} \\ \theta = \langle\!| \ m : \tau_m @ (\rho_m[r'/r]) \ ^{m \in \mathcal{M}} \ |\!\rangle [r'/r] \quad \Gamma \vdash \theta \text{ ok}\end{array}}{\Gamma \vdash T \ \textbf{rename} \ r \ \textbf{to} \ r' \ : \ \theta}$$

Method typing: $\boxed{\Gamma \vdash \mu \ : \ \tau_1 \to \tau_2}$

$$\frac{\Gamma, \ x : \tau_1 \vdash e \ : \ \tau_2}{\Gamma \vdash m \ (x : \tau_1) : \tau_2 \ \{e\} \ : \ \tau}$$

Class typing: $\boxed{\Gamma \vdash C \ : \ \chi}$

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash c \ : \ \Gamma(c)} \qquad \frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \textbf{nil} \ : \ \textbf{unit} \to \{\!| \ |\!\}}$$

$$\frac{\begin{array}{c}\Gamma,\ x:\tau \vdash e_{\text{cons}}\ :\ \tau_{\text{cons}} \qquad \Gamma,\ x:\tau \vdash e_F\ :\ \{f:\tau_f{}^{f\in\mathcal{F}}\} \\ \Gamma \vdash T\ :\ \theta \qquad \Gamma \vdash C\ :\ \tau_{\text{cons}} \rightarrow \{\!|\ l:\tau_l{}^{l\in\mathcal{L}_C}\ |\!\} \qquad \mathcal{F} \pitchfork \mathcal{L}_C \\ \mathcal{L} = \mathcal{L}_C \cup \mathcal{F} \cup \text{dom}(\theta) \\ \langle l:\tau_l{}^{l\in\mathcal{L}_{\mathbf{super}}}\rangle = \text{super}(\text{Inl}(\theta)) \quad \mathcal{L}_{\mathbf{super}} \subset \mathcal{L}_C \\ \langle l:\tau_l{}^{l\in\mathcal{L}_{\mathbf{self}}}\rangle = \text{self}(\text{Inl}(\theta)) \quad \mathcal{L}_{\mathbf{self}} \subset \mathcal{L}\end{array}}{\begin{array}{c}\Gamma \vdash \lambda(x:\tau).(\mathbf{super}\ e_{\text{cons}}) \oplus e_F\ \mathbf{in} \\ T\ \mathbf{extends}\ C\ :\ \tau \rightarrow \{\!|\ l:\tau_l{}^{l\in\mathcal{L}}\ |\!\}\end{array}}$$

Expression typing: $\boxed{\Gamma \vdash e\ :\ \tau}$

$$\frac{\Gamma \vdash \text{ok} \quad x \in \text{dom}(\Gamma)}{\Gamma \vdash x\ :\ \Gamma(x)} \qquad \frac{\Gamma,\ x:\tau_1 \vdash e\ :\ \tau_2}{\Gamma \vdash \lambda(x:\tau).e\ :\ \tau_1 \rightarrow \tau_2} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1\ :\ \tau' \rightarrow \tau \\ \Gamma \vdash e_2\ :\ \tau'\end{array}}{\Gamma \vdash e_1\, e_2\ :\ \tau}$$

$$\frac{\Gamma \vdash e\ :\ \tau \quad \Gamma \vdash \tau\ <:\ \tau'}{\Gamma \vdash e\ :\ \tau'} \qquad \frac{\begin{array}{c}\Gamma \vdash e\ :\ \tau \\ \Gamma \vdash \tau\ <:\ \langle l:\tau_l\rangle\end{array}}{\Gamma \vdash e.l\ :\ \tau_l} \qquad \frac{\Gamma \vdash \text{ok}}{\Gamma \vdash ()\ :\ \mathbf{unit}}$$

$$\frac{\begin{array}{c}\Gamma \vdash C\ :\ \tau \rightarrow \{\!|\ l:\tau_l{}^{l\in\mathcal{L}}\ |\!\} \\ \Gamma \vdash e\ :\ \tau\end{array}}{\Gamma \vdash \mathbf{new}\ C\ e\ :\ \langle l:\tau_l{}^{l\in\mathcal{L}}\rangle} \qquad \frac{\Gamma \vdash \Gamma(\mathit{super})\ <:\ \langle m:\tau_m\rangle}{\Gamma \vdash \mathbf{super}.m\ :\ \tau_m} \qquad \frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \mathbf{self}\ :\ \Gamma(\mathit{self})}$$

$$\frac{\Gamma \vdash \text{ok} \qquad \Gamma \vdash e_f\ :\ \tau_f \quad \forall f \in \mathcal{F}}{\Gamma \vdash \{f = e_f{}^{f\in\mathcal{F}}\}\ :\ \{f:\tau_f{}^{f\in\mathcal{F}}\}} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1\ :\ \{f:\tau_f{}^{f\in\mathcal{F}_1}\} \\ \Gamma \vdash e_2\ :\ \{f:\tau_f{}^{f\in\mathcal{F}_2}\}\end{array} \quad \mathcal{F}_1 \pitchfork \mathcal{F}_2}{\Gamma \vdash e_1 \oplus e_2\ :\ \{f:\tau_f{}^{f\in\mathcal{F}_1\cup\mathcal{F}_2}\}}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1\ :\ \tau \quad \Gamma \vdash \tau\ <:\ \langle f:\tau_f\rangle \\ \Gamma \vdash e_2\ :\ \tau_f\end{array}}{\Gamma \vdash e_1.f := e_2\ :\ \mathbf{unit}}$$

Auxilliary judgments:

$$\frac{\rho = \biguplus_{m\in\mathcal{M}}(\rho_m \uplus \langle\!\langle m:\tau_m\rangle\!\rangle) \qquad \Gamma \vdash \rho\ \text{ok}}{\text{Inl}(\{\!\!|\ m:\tau_m@\rho_m{}^{m\in\mathcal{M}}\ |\!\!\}) = \rho}$$

Auxilliary functions:

$$\begin{array}{l}\text{super}\ \langle\!\langle r:\tau_r{}^{r\in\mathcal{R}}\rangle\!\rangle = \langle m:\tau_{\mathbf{super}.m}{}^{\mathbf{super}.m\in\mathcal{R}}\rangle \\ \text{self}\ \langle\!\langle r:\tau_r{}^{r\in\mathcal{R}}\rangle\!\rangle = \langle l:\tau_l{}^{l\in\mathcal{R}\cap\mathcal{L}_U}\rangle\end{array}$$

$$\text{hide}(\rho, m', \rho_{m'}) = \begin{cases}(\rho \uplus \rho_{m'}) \setminus m' & m' \in \rho \\ \rho & \text{otherwise}\end{cases}$$

C Evaluation judgments

Program evaluation: $\boxed{E,\ S \vdash P \longrightarrow ev,\ E',\ S'}$

$$\frac{E,\ S \vdash D \longrightarrow E',\ S' \quad E',\ S' \vdash P \longrightarrow ev,\ E'',\ S''}{E,\ S \vdash D;\ P \longrightarrow ev,\ E'',\ S''} \qquad \frac{E,\ S \vdash e \longrightarrow ev,\ S'}{E,\ S \vdash e \longrightarrow ev,\ E,\ S'}$$

Declaration evaluation: $\boxed{E,\ S \vdash D \longrightarrow E',\ S'}$

$$\frac{E \vdash T \longrightarrow tv \quad t \notin \mathrm{dom}(E)}{E,\ S \vdash t = (\vec{\alpha})T \longrightarrow E[t \mapsto (\vec{\alpha})tv],\ S} \qquad \frac{E \vdash C \longrightarrow cv \quad c \notin \mathrm{dom}(E)}{E,\ S \vdash c = C \longrightarrow E[c \mapsto cv],\ S}$$

$$\frac{E,\ S \vdash e \longrightarrow ev \quad x \notin \mathrm{dom}(E)}{E,\ S \vdash x = e \longrightarrow E[x \mapsto ev],\ S'}$$

Trait evaluation: $\boxed{E \vdash T \longrightarrow tv}$

$$\frac{E(t) = (\vec{\alpha})tv}{E \vdash t(\vec{\tau}) \longrightarrow tv[\vec{\tau}/\vec{\alpha}]}$$

$$\frac{\begin{array}{c}\phi_P = \mathrm{NS}(\{m\},\ \emptyset) \quad \phi_R = \mathrm{NS}(\mathrm{dom}(\rho) \setminus \mathcal{F}_U,\ \{\phi_P(m)\}) \\ Mv = \{\phi_P(m) \mapsto [E;\ \phi_P \cup \phi_R;\ \lambda(x:\tau_1).e;\ \rho]\ \}\end{array}}{E \vdash \langle\!|\ m\ (x:\tau_1):\tau_2\ \{e\};\ \rho\ |\!\rangle \longrightarrow \langle\!|\ Mv;\ \phi_P;\ \phi_R\ |\!\rangle}$$

$$\frac{\begin{array}{c}E \vdash T_1 \longrightarrow \langle\!|\ {}^{\mathcal{I}_1}Mv_1;\ {}^{\mathcal{M}_1}\phi_{P_1};\ {}^{\mathcal{R}_1}\phi_{R_1}\ |\!\rangle \\ E \vdash T_2 \longrightarrow \langle\!|\ {}^{\mathcal{I}_2}Mv_2;\ {}^{\mathcal{M}_2}\phi_{P_2};\ {}^{\mathcal{R}_2}\phi_{R_2}\ |\!\rangle \\ \mathcal{M}_1 \pitchfork \mathcal{M}_2 \quad Mv_2 = \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \rho_i]\ ^{i\in\mathcal{I}_2}\} \\ \varphi_P = \{\phi_{P_2}(m) \mapsto \phi_{R_1}(m)\ ^{m\in\mathcal{M}_2\cap\mathcal{R}_1}\} \\ \varphi_R = \{\phi_{R_2}(m) \mapsto \phi_{P_1}(m)\ ^{m\in\mathcal{R}_2\cap\mathcal{M}_1}\} \\ \mathcal{I}_1' = \mathcal{I}_1 \cup \mathrm{rng}(\phi_{R_1}) \quad \mathcal{I}_2' = \mathcal{I}_2 \cup \mathrm{rng}(\phi_{R_2}) \\ \varphi_F = \mathrm{FS}(\mathcal{I}_2' \setminus \mathrm{dom}(\varphi_R \cup \varphi_P),\ \mathcal{I}_1') \\ \varphi = \varphi_P \cup \varphi_R \cup \varphi_F \quad \phi_P = \phi_{P_1} \cup (\varphi \circ \phi_{P_2}) \\ \phi_R = (\phi_{R_1} \cup (\varphi \circ \phi_{R_2})) \setminus (\mathcal{M}_1 \cup \mathcal{M}_2) \\ Mv = Mv_1 \cup \{\varphi(i) \mapsto [E_i;\ \varphi \circ \phi_i;\ e_i;\ \rho_i]\ ^{i\in\mathcal{I}_2}\}\end{array}}{E \vdash T_1 + T_2 \longrightarrow \langle\!|\ Mv;\ \phi_P;\ \phi_R\ |\!\rangle}$$

$$\frac{\begin{array}{c}E \vdash T \longrightarrow \langle\!|\ {}^{\mathcal{I}}Mv;\ {}^{\mathcal{M}}\phi_P;\ {}^{\mathcal{R}}\phi_R\ |\!\rangle \quad m \in \mathcal{M} \\ \langle\!|\ Mv \setminus \phi_P(m);\ \phi_P \setminus m;\ \phi_R[m \mapsto \phi_P(m)]\ |\!\rangle \hookrightarrow tv\end{array}}{E \vdash T\ \mathbf{exclude}\ m \longrightarrow tv}$$

$$\frac{\begin{array}{c}E \vdash T \longrightarrow \langle\!|\ {}^{\mathcal{I}}Mv;\ {}^{\mathcal{M}}\phi_P;\ {}^{\mathcal{R}}\phi_R\ |\!\rangle \\ m \in \mathcal{M} \quad j = \phi_P(m) \quad Mv = \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \rho_i]\ ^{i\in\mathcal{I}}\} \\ Mv' = \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \mathrm{hide}(\rho_i, m, \rho_j)]\ ^{i\in\mathcal{I}}\} \\ \langle\!|\ Mv';\ \phi_P \setminus m;\ \phi_R\ |\!\rangle \hookrightarrow tv\end{array}}{E \vdash T\ \mathbf{hide}\ m \longrightarrow tv}$$

$$\frac{\begin{array}{c} E \vdash T \longrightarrow \llparenthesis\, {}^{\mathcal{I}}Mv;\ {}^{\mathcal{M}}\phi_P;\ {}^{\mathcal{R}}\phi_R \,\rrparenthesis \\ m \in \mathcal{M} \quad m' \notin \mathcal{M} \quad m' \in \mathcal{R} \\ \phi'_P = \phi_P[m' \mapsto \phi_R(m')] \\ Mv' = Mv[\phi_R(m') \mapsto Mv(\phi_P(m))] \end{array}}{\begin{array}{c} E \vdash T \ \textbf{alias}\ m\ \textbf{as}\ m' \longrightarrow \\ \llparenthesis\, Mv';\ \phi'_P;\ \phi_R \setminus m' \,\rrparenthesis \end{array}} \qquad \frac{\begin{array}{c} E \vdash T \longrightarrow \llparenthesis\, {}^{\mathcal{I}}Mv;\ {}^{\mathcal{M}}\phi_P;\ {}^{\mathcal{R}}\phi_R \,\rrparenthesis \\ m \in \mathcal{M} \quad m' \notin \mathcal{M} \quad m' \notin \mathcal{R} \\ \phi'_P = \phi_P \cup \mathrm{NS}(\{m'\},\ \mathcal{I} \cup \mathrm{rng}(\phi_R)) \\ Mv' = Mv[\phi'_P(m') \mapsto Mv(\phi_P(m))] \end{array}}{\begin{array}{c} E \vdash T \ \textbf{alias}\ m\ \textbf{as}\ m' \longrightarrow \\ \llparenthesis\, Mv';\ \phi'_P;\ \phi_R \,\rrparenthesis \end{array}}$$

$$\frac{\begin{array}{c} E \vdash T \longrightarrow \llparenthesis\, {}^{\mathcal{I}}Mv;\ {}^{\mathcal{M}}\phi_P;\ {}^{\mathcal{R}}\phi_R \,\rrparenthesis \\ r \in \mathcal{M} \quad r' \notin \mathcal{M} \quad r' \notin \mathcal{R} \\ Mv = \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \rho_i]^{\ i \in \mathcal{I}}\} \\ \phi'_P = (\phi_P \setminus r)[r' \mapsto \phi_P(r)] \\ Mv' = \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \rho_i[r'/r]]^{\ i \in \mathcal{I}}\} \end{array}}{\begin{array}{c} E \vdash T \ \textbf{rename}\ r\ \textbf{to}\ r' \longrightarrow \\ \llparenthesis\, Mv';\ \phi'_P;\ \phi_R \,\rrparenthesis \end{array}} \qquad \frac{\begin{array}{c} E \vdash T \longrightarrow \llparenthesis\, {}^{\mathcal{I}}Mv;\ {}^{\mathcal{M}}\phi_P;\ {}^{\mathcal{R}}\phi_R \,\rrparenthesis \\ r \in \mathcal{R} \quad r' \notin \mathcal{M} \quad r' \notin \mathcal{R} \\ Mv = \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \rho_i]^{\ i \in \mathcal{I}}\} \\ \phi'_R = (\phi_R \setminus r)[r' \mapsto \phi_R(r)] \\ Mv' = \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \rho_i[r'/r]]^{\ i \in \mathcal{I}}\} \end{array}}{\begin{array}{c} E \vdash T \ \textbf{rename}\ r\ \textbf{to}\ r' \longrightarrow \\ \llparenthesis\, Mv';\ \phi_P;\ \phi'_R \,\rrparenthesis \end{array}}$$

$$\frac{\begin{array}{c} E \vdash T \longrightarrow \llparenthesis\, {}^{\mathcal{I}}Mv;\ {}^{\mathcal{M}}\phi_P;\ {}^{\mathcal{R}}\phi_R \,\rrparenthesis \\ r \in \mathcal{M} \quad r' \notin \mathcal{M} \quad r' \in \mathcal{R} \\ Mv = \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \rho_i]^{\ i \in \mathcal{I}}\} \\ \phi'_P = (\phi_P \setminus r)[r' \mapsto \phi_P(r)] \\ \varphi = \{i \mapsto i,\ \phi_R(r') \mapsto \phi_P(r)\} \\ Mv' = \{i \mapsto [E_i;\ \varphi \circ \phi_i;\ e_i;\ \rho_i[r'/r]]^{\ i \in \mathcal{I}}\} \end{array}}{E \vdash T \ \textbf{rename}\ r\ \textbf{to}\ r' \longrightarrow \llparenthesis\, Mv';\ \phi'_P;\ \phi_R \setminus m' \,\rrparenthesis}$$

Trait "dead code" elimination: $\boxed{tv \hookrightarrow tv'}$

$$\frac{Mv(i) = [E;\ \phi;\ e;\ \rho] \qquad i' \in \mathrm{rng}(\phi)}{Mv \vdash i \text{ uses } i'}$$

$$\frac{\begin{array}{c} Mv = \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \rho_i]^{\ i \in \mathrm{dom}(Mv)}\} \\ \mathcal{I} = \{i \mid j \in \mathrm{rng}(\phi_P) \text{ and } Mv \vdash j \text{ uses}^* i\} \qquad \mathcal{R} = \phi_R^{-1}(\mathcal{I}) \end{array}}{\llparenthesis\, Mv;\ \phi_P;\ \phi_R \,\rrparenthesis \hookrightarrow \llparenthesis\, Mv \cap \mathcal{I};\ \phi_P;\ \phi_R \cap \mathcal{R} \,\rrparenthesis}$$

Class evaluation: $\boxed{E \vdash C \longrightarrow cv}$

$$\frac{c \in E}{E \vdash c \longrightarrow E(c)} \qquad \frac{}{E \vdash \textbf{nil} \longrightarrow \{\!|\ [\emptyset;\ \lambda x.\{\}];\ \emptyset;\ \emptyset\ |\!\}}$$

$$\frac{\begin{array}{c} E \vdash T \longrightarrow tv \quad tv = \langle\!\!\langle\, {}^{\mathcal{I}}Mv;\ {}^{\mathcal{M}}\phi_P;\ {}^{\mathcal{R}}\phi_R \,\rangle\!\!\rangle \\ E \vdash C \longrightarrow cv \quad cv = \{\!\!|\ \lambda v_{\mathbf{super}};\ Mv_C;\ \phi_C\ |\!\!\} \\ cv \vdash tv \Longrightarrow \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \rho_i]^{\ i \in \mathcal{I}'}\} \\ \varphi_P = \{\phi_P(m) \mapsto \phi_C(m)^{\ m \in \mathcal{M} \cap \mathrm{dom}(\phi_C)}\} \\ \varphi_R = \{\phi_R(m) \mapsto \phi_C(m)^{\ m \in \mathcal{R} \cap \mathrm{dom}(\phi_C)}\} \\ \varphi_F = \mathrm{FS}\big(\mathcal{I}' \setminus \mathrm{dom}(\varphi_P),\ \mathrm{dom}(Mv_C)\big) \\ \varphi = \varphi_P \cup \varphi_R \cup \varphi_F \\ Mv'_T = \{\varphi(i) \mapsto [E_i;\ \varphi \circ \phi_i;\ e_i;\ \rho_i]^{\ i \in \mathcal{I}'}\} \\ Mv'_C = (Mv_C \setminus \mathrm{rng}(\varphi_P)) \cup Mv'_T \\ \mathit{super} \notin \mathrm{dom}(E) \quad E_{cons} = E[\mathit{super} \mapsto \lambda v_{\mathbf{super}}] \\ \lambda v_{cons} = [E_{con};\ \lambda(x:\tau).(\mathit{super}\ e_{cons}) \oplus e_F] \end{array}}{\begin{array}{c} E \vdash \lambda(x:\tau).(\mathbf{super}\ e_{\mathrm{cons}}) \oplus e_F\ \mathbf{in} \\ T\ \mathbf{extends}\ C \longrightarrow \{\!\!|\ \lambda v_{cons};\ Mv'_C;\ \phi_C \cup (\varphi \circ \phi_P)\ |\!\!\} \end{array}}$$

Super-method inclusion: $\boxed{cv \vdash tv \Longrightarrow Mv}$

$$\frac{Mv' = Mv \cup \{\phi_R(s) \mapsto Mv_C(\phi_C(s))^{\ s \in \mathrm{dom}(\phi_R) \cap \mathcal{S}_U}\}}{\{\!\!|\ \lambda v_{\mathbf{super}};\ Mv_C;\ \phi_C\ |\!\!\} \vdash \langle\!\!\langle\ Mv;\ \phi_P;\ \phi_R\ \rangle\!\!\rangle \Longrightarrow Mv'}$$

Expression evaluation: $\boxed{E,\ S \vdash e \longrightarrow ev,\ S'}$

$$\frac{}{E,\ S \vdash \{\} \longrightarrow \{\},\ S} \qquad \frac{}{E,\ S \vdash () \longrightarrow (),\ S} \qquad \frac{x \in E}{E,\ S \vdash x \longrightarrow E(x),\ S}$$

$$\frac{}{E,\ S \vdash \lambda(x:\tau).e \longrightarrow [E;\ \lambda(x:\tau).e],\ S} \qquad \frac{\begin{array}{c} E,\ S \vdash e_1 \longrightarrow [E';\ \lambda(x:\tau).e],\ S_1 \\ E,\ S_1 \vdash e_2 \longrightarrow ev_2,\ S_2 \\ E'[x \mapsto ev_2],\ S_2 \vdash e \longrightarrow ev,\ S_3 \end{array}}{E,\ S \vdash e_1\ e_2 \longrightarrow ev,\ S_3}$$

$$\frac{\begin{array}{c} E,\ S \vdash e \longrightarrow (a,\ \phi),\ S' \\ S'(a) = \langle fv;\ Mv\rangle \quad fv(f) = ev_f \end{array}}{E,\ S \vdash e.f \longrightarrow ev_f,\ S'} \qquad \frac{\begin{array}{c} E,\ S \vdash e_1 \longrightarrow (a,\ \phi),\ S_1 \\ E,\ S_1 \vdash e_2 \longrightarrow ev_2,\ S_2 \\ S_2(a) = \langle fv;\ Mv\rangle \\ S_3 = S_2[a \mapsto \langle fv[f = ev_2];\ Mv\rangle] \end{array}}{E,\ S \vdash e_1.f := e_2 \longrightarrow (),\ S_3}$$

$$\frac{\begin{array}{c} E,\ S \vdash e_1 \longrightarrow ev_1,\ S_1 \\ \vdots \\ E,\ S_{n-1} \vdash e_n \longrightarrow ev_n,\ S_n \end{array}}{\begin{array}{c} E,\ S \vdash \{f_1 = e_1,\ \ldots, f_n = e_n\} \longrightarrow \\ \{f_1 = ev_1,\ \ldots, f_n = ev_n\},\ S_n \end{array}} \qquad \frac{\begin{array}{c} E,\ S \vdash e_1 \longrightarrow \{f = ev_f^{\ f \in \mathcal{F}_1}\},\ S_1 \\ E,\ S_1 \vdash e_2 \longrightarrow \{f = ev_f^{\ f \in \mathcal{F}_2}\},\ S_2 \end{array}}{\begin{array}{c} E,\ S \vdash e_1 \oplus e_2 \longrightarrow \\ \{f = ev_f^{\ f \in \mathcal{F}_1 \cup \mathcal{F}_2}\},\ S_2 \end{array}}$$

$$\frac{E,\ S \vdash \mathbf{self}.(\mathbf{super}.m) \longrightarrow ev,\ S'}{E,\ S \vdash \mathbf{super}.m \longrightarrow ev,\ S'} \qquad \frac{}{E,\ S \vdash \mathbf{self} \longrightarrow E(\mathit{self}),\ S}$$

$$\frac{\begin{array}{l} E \vdash C \longrightarrow \{\!|\ [E_F;\ \lambda x.e_F];\ Mv;\ \phi_C\ |\!\} \\ E,\ S \vdash e \longrightarrow ev,\ S_1 \\ E_F[x \mapsto ev],\ S_1 \vdash e_F \longrightarrow fv,\ S_2 \quad a \notin \mathrm{dom}(S_2) \end{array}}{E,\ S \vdash \mathbf{new}\ C\ e \longrightarrow (a,\ \phi_C),\ S_2[a \mapsto \langle fv;\ Mv \rangle]}$$

$$\frac{\begin{array}{c} E,\ S \vdash e \longrightarrow (a,\ \phi),\ S' \quad S'(a) = \langle fv;\ Mv \rangle \\ Mv(\phi(m)) = [E_m;\ \phi_m;\ \lambda(x:\tau).e_m;\ \rho_m] \end{array}}{E,\ S \vdash e.m \longrightarrow [E_m[\mathit{self} \mapsto (a,\ \phi_m)];\ \lambda(x:\tau).e_m],\ S'}$$

D Run-time typing

Well-formed store types: $\boxed{\Sigma \vdash \mathrm{ok}}$

$$\frac{\emptyset \vdash \Sigma(a)\ \mathrm{ok} \quad \forall a \in \mathrm{dom}(\Sigma)}{\Sigma \vdash \mathrm{ok}}$$

Environment typing: $\boxed{\Sigma \vdash E\ :\ \Gamma}$

$$\frac{\Sigma \vdash \mathrm{ok}}{\Sigma \vdash \emptyset\ :\ \emptyset}$$

$$\frac{\Sigma \vdash E\ :\ \Gamma \qquad \Sigma \vdash_{(\vec{\alpha})} tv\ :\ \theta \qquad t \notin \mathrm{dom}(E)}{\Sigma \vdash E[t \mapsto (\vec{\alpha})tv]\ :\ \Gamma,\ t : \Lambda(\vec{\alpha}).\theta}$$

$$\frac{\Sigma \vdash E\ :\ \Gamma \qquad \Sigma \vdash cv\ :\ \chi \qquad c \notin \mathrm{dom}(E)}{\Sigma \vdash E[c \mapsto cv]\ :\ \Gamma,\ c : \chi}$$

$$\frac{\Sigma \vdash E\ :\ \Gamma \qquad \Sigma \vdash ev\ :\ \tau' \qquad \emptyset \vdash \tau'\ <:\ \tau \qquad x \notin \mathrm{dom}(E)}{\Sigma \vdash E[x \mapsto ev]\ :\ \Gamma,\ x : \tau}$$

Store typing: $\boxed{\vdash S\ :\ \Sigma}$

$$\frac{\Sigma \vdash \mathrm{ok} \qquad \mathrm{dom}(\Sigma) = \mathrm{dom}(S) \qquad \forall a \in \mathrm{dom}(S)\ \Sigma \vdash S(a)\ :\ \Sigma(a)}{\vdash S\ :\ \Sigma}$$

Trait value typing: $\boxed{\Sigma \vdash_{(\vec{\alpha})} tv\ :\ \theta}$

$$\frac{\begin{array}{c} Mv = \{i \mapsto [E_i;\ \phi_i;\ e_i;\ \rho_i]^{\ i \in \mathrm{dom}(Mv)}\} \\ \theta = \langle\!|\ m : \sigma(\phi_P(m))@\rho_{\phi_P(m)}{}^{\ m \in \mathrm{dom}(\phi_P)}\ |\!\rangle \\ \rho = \mathrm{Inl}(\theta) \qquad \phi = \phi_P \cup \phi_R \\ \tau_{\text{fields}} = \{f : \rho(f)^{\ f \in \mathrm{dom}(\rho) \cap \mathcal{F}_U}\} \\ \rho(r) = \sigma(\phi(r)) \quad \forall r \in \mathrm{dom}(\rho) \setminus \mathcal{F}_U \\ \Sigma \vdash_{(\vec{\alpha});\tau_{\text{fields}}} Mv\ :\ \sigma \end{array}}{\Sigma \vdash_{(\vec{\alpha})} \langle\!|\ Mv;\ \phi_P;\ \phi_R\ |\!\rangle\ :\ \theta}$$

Method suite typing: $\boxed{\Sigma \vdash_{(\vec{\alpha});\tau_{\text{fields}}} Mv \; : \; \sigma}$

$$\frac{\begin{array}{c}\text{dom}(Mv) = \text{dom}(\sigma) \\ Mv = \{ i \mapsto [E_i; \; \phi_i; \; e_i; \; \rho_i] \; ^{i \in \text{dom}(Mv)} \} \\ \forall i \in \text{dom}(Mv) \quad \Sigma \vdash_{(\vec{\alpha});\sigma;\tau_{\text{fields}}} Mv(i) \; : \; \sigma(i)\end{array}}{\Sigma \vdash_{(\vec{\alpha});\tau_{\text{fields}}} Mv \; : \; \sigma}$$

Dictionary application typing: $\boxed{\phi \vdash \sigma \; : \; \tau}$

$$\frac{}{\phi \vdash \sigma \; : \; \langle m : \sigma(\phi(m)) \; ^{m \in \text{dom}(\phi)} \rangle}$$

Method value typing: $\boxed{\Sigma \vdash_{(\vec{\alpha});\sigma;\tau_{\text{fields}}} \mu v \; : \; \tau}$

$$\frac{\begin{array}{c}\Sigma \vdash E \; : \; \Gamma \qquad \tau_{\text{fields}} = \{ f : \tau_f \; ^{f \in \mathcal{F}} \} \\ (\phi_\mu \cap \mathcal{M}_U) \vdash \sigma \; : \; \tau'_{\mathbf{self}} \quad (\phi_\mu \cap \mathcal{S}_U) \vdash \sigma \; : \; \tau_{\mathbf{super}} \\ \tau_{\mathbf{self}} = \tau'_{\mathbf{self}} \uplus \langle f : \tau_f \; ^{f \in \mathcal{F}} \rangle \\ \Gamma, \; \mathit{super} : \tau_{\mathbf{super}}, \; \mathit{self} : \tau_{\mathbf{self}}, \; x : \tau \vdash e \; : \; \tau'\end{array}}{\Sigma \vdash_{(\vec{\alpha});\sigma;\tau_{\text{fields}}} [E; \; \phi_\mu; \; \lambda(x : \tau).e; \; \rho] \; : \; \tau \rightarrow \tau'}$$

Class value typing: $\boxed{\Sigma \vdash cv \; : \; \chi}$

$$\frac{\begin{array}{c}\Sigma \vdash \lambda v \; : \; \tau \rightarrow \tau_{\text{fields}} \quad \tau_{\text{fields}} = \{ f : \tau_f \; ^{f \in \mathcal{F}} \} \\ \Sigma \vdash_{();\tau_{\text{fields}}} Mv \; : \; \sigma \qquad \phi_C \vdash \sigma \; : \; \langle l : \tau_l \; ^{l \in \mathcal{L}} \rangle\end{array}}{\Sigma \vdash \{\!| \; \lambda v; \; Mv; \; \phi_C \; |\!\} \; : \; \tau \rightarrow \{\!| \; l : \tau_l \; ^{l \in \mathcal{L}}, \; f : \tau_f \; ^{f \in \mathcal{F}} \; |\!\}}$$

Expression value typing: $\boxed{\Sigma \vdash ev \; : \; \tau}$

$$\frac{\begin{array}{c}\Sigma \vdash \text{ok} \qquad \Sigma(a) = \sigma, \; \{ f : \tau_f \; ^{f \in \mathcal{F}} \} \\ \phi \vdash \sigma \; : \; \langle l : \tau_l \; ^{l \in \mathcal{L}} \rangle\end{array}}{\Sigma \vdash (a, \phi) \; : \; \langle l : \tau_l \; ^{l \in \mathcal{L}}, \; f : \tau_f \; ^{f \in \mathcal{F}} \rangle}$$

$$\frac{\Sigma \vdash E \; : \; \Gamma \qquad \Gamma, \; x : \tau \vdash e \; : \; \tau'}{\Sigma \vdash [E; \; \lambda(x : \tau).e] \; : \; \tau \rightarrow \tau'}$$

$$\frac{\Sigma \vdash \text{ok} \qquad \Sigma \vdash ev_f \; : \; \tau'_f \qquad \emptyset \vdash \tau'_f \; <: \; \tau_f \qquad \forall f \in \mathcal{F}}{\Sigma \vdash \{ f = ev_f \; ^{f \in \mathcal{F}} \} \; : \; \{ f : \tau_f \; ^{f \in \mathcal{F}} \}}$$

$$\frac{\Sigma \vdash \text{ok}}{\Sigma \vdash () \; : \; \mathbf{unit}}$$

Object value typing: $\boxed{\Sigma \vdash ov \; : \; \sigma, \tau}$

$$\frac{\Sigma \vdash \mathit{fv} \; : \; \tau_{\text{fields}} \qquad \Sigma \vdash_{();\tau_{\text{fields}}} Mv \; : \; \sigma}{\Sigma \vdash \langle \mathit{fv}; \; Mv \rangle \; : \; \sigma, \tau_{\text{fields}}}$$

$$
\begin{aligned}
H_{E,\,S}(t = (\vec{\alpha})T;\ P) &= 1 + \begin{cases} H_{E',\,S}(P) & \text{if } E,\ S \vdash t = (\vec{\alpha})T \longrightarrow E',\ S \\ 1 & \text{otherwise} \end{cases} \\
H_{E,\,S}(c = C;\ P) &= 1 + \begin{cases} H_{E',\,S}(P) & \text{if } E,\ S \vdash c = C \longrightarrow E',\ S \\ 1 & \text{otherwise} \end{cases} \\
H_{E,\,S}(x = e;\ P) &= 1 + h_{E,\,S}(e) + \begin{cases} H_{E',\,S'}(P) & \text{if } E,\ S \vdash x = e \longrightarrow E',\ S' \\ 1 & \text{otherwise} \end{cases} \\
H_{E,\,S}(e) &= h_{E,\,S}(e)
\end{aligned}
$$

Figure 7: Program evaluation height

$$
\begin{array}{lcl}
h_{E,\,S}(x) & = & 1 \\
h_{E,\,S}(\lambda(x:\tau).e) & = & 1 \\
h_{E,\,S}(e_1\;e_2) & = & 1 \\
 & + & h_{E,\,S}(e_1) \\
 & + & \begin{cases} h_{E,\,S_1}(e_2) & \text{if } E,\;S \vdash e_1 \longrightarrow [E_1;\;\lambda(x:\tau).e],\;S_1 \\ 1 & \text{otherwise} \end{cases} \\
 & + & \begin{cases} h_{E_1[x\mapsto ev_2],\,S_2}(e) & \text{if } E,\;S \vdash e_1 \longrightarrow [E_1;\;\lambda(x:\tau).e],\;S_1 \\ & \text{and } E,\;S_1 \vdash e_2 \longrightarrow ev_2,\;S_2 \\ 1 & \text{otherwise} \end{cases} \\
h_{E,\,S}(\mathbf{new}\;C\;e) & = & 1 \\
 & + & h_{E,\,S}(e) \\
 & + & \begin{cases} h_{E_F[x\mapsto ev],\,S'}(e_F) & \text{if } E \vdash C \longrightarrow \{\!|\;[E_F;\;\lambda x.e_F];\;Mv;\;\phi_C\;|\!\} \\ & \text{and } E,\;S \vdash e \longrightarrow ev,\;S' \\ 1 & \text{otherwise} \end{cases} \\
h_{E,\,S}(\mathbf{self}) & = & 1 \\
h_{E,\,S}(\mathbf{super}.m) & = & 1 \\
h_{E,\,S}(e.m) & = & 1 + h_{E,\,S}(e) \\
h_{E,\,S}(e.f) & = & 1 + h_{E,\,S}(e) \\
h_{E,\,S}(e_1.f := e_2) & = & 1 \\
 & + & h_{E,\,S}(e_1) \\
 & + & \begin{cases} h_{E,\,S'}(e_2) & \text{if } E,\;S \vdash e_1 \longrightarrow ev_1,\;S' \\ 1 & \text{otherwise} \end{cases} \\
h_{E,\,S}(\{\}) & = & 1 \\
h_{E,\,S_0}(\{f_1 = e_1, \ldots, f_n = e_n\}) & = & 1 \\
 & + & h_{E,\,S_0}(e_1) \\
 & \vdots & \\
 & + & \begin{cases} h_{E,\,S_{n-1}}(e_n) & \text{if } E,\;S_0 \vdash e_1 \longrightarrow ev_1,\;S_1 \\ & \vdots \\ & \text{and } E,\;S_{n-2} \vdash e_{n-1} \longrightarrow ev_{n-1},\;S_{n-1} \\ 1 & \text{otherwise} \end{cases} \\
h_{E,\,S}(e_1 \oplus e_2) & = & 1 \\
 & + & h_{E,\,S}(e_1) \\
 & + & \begin{cases} h_{E,\,S'}(e_2) & \text{if } E,\;S \vdash e_1 \longrightarrow ev_1,\;S' \\ 1 & \text{otherwise} \end{cases} \\
h_{E,\,S}(()) & = & 1
\end{array}
$$