A typed calculus of traits

Kathleen Fisher
AT&T Labs — Research
kfisher@research.att.com

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

December 14, 2003

1 Introduction

Schärli *et al.* recently proposed a mechanism called *traits* as a way to foster code reuse in object-oriented programs [SDNB03]. They have prototyped the mechanism in the context of the Squeak implementation of Smalltalk. Using traits, they refactored the Smalltalk collection classes achieving a 25% reduction in the number of method implementations and a 10% reduction in source code size [BSD03]. This early experience suggests that traits are a promising mechanism for factoring class hierarchies and supporting code reuse. Smalltalk, however, is a dynamically-typed language, so it is not clear if this mechanism can be added to a statically-typed language such as Java [AG98] or Moby [FR99, FR03]. In this paper, we present a typed calculus of traits that can serve as the foundation for integrating traits into a statically-typed object-oriented language.

In languages with single inheritance, such as Smalltalk, it is often the case that inheritance does not provide sufficient flexibility when structuring a class hierarchy. Consider the case of two classes in different subtrees of the inheritance hierarchy and assume that they both implement some common protocol. Attempting to share the implementation of this protocol may result in having the common methods defined *too high* in the inheritance hierarchy (*i.e.*, these methods will be inherited by other classes that do not provide the protocol in question). On the other hand, keeping the methods at the "right" height results in code duplication. Multiple inheritance [Str94] is one possible solution to this problem, but it is a complicated mechanism that suffers from other problems (*e.g.*, multiple copies of instance variables). Mixins are a mechanism designed to give many of the benefits of multiple inheritance in single-inheritance languages There are strong similarities between traits and *mixins* [BC90, FKF98, OAC+03], which are another mechanism designed to address code sharing in single-inheritance languages. The main difference between mixins and traits is that mixins force a linear order in their composition. This order avoids the complexities of the diamond property, but it makes mixins a more fragile mechanismn [SDNB03].

A trait is collection of named methods. In Smalltalk traits, these methods cannot directly reference instance variables; instead, they must be "*pure behavior*." The methods defined in a trait are called the *provided methods*, while any methods that are referenced, but not provided, are called *required methods*. An important property of traits is that while they help structure the implementation of classes, they do not affect the inheritance hierarchy. Traits are formed by definition (*i.e.*, listing a collection of method definitions) or by using one of several trait operations:

Symmetric sum merges two disjoint traits to create a new trait.[1]

Override forms a new trait by layering additional methods over an existing trait. This operation is an asymmetric sum. When one of the new methods has the same name as a method in the original trait, the override operation replaces the original method.

[1] The most recent description of Smalltalk traits ([BSD03]) allows name conflicts, but replaces the conflicting methods with a special method body **conflict** that triggers a run-time error if evaluated.

Alias creates a new trait by adding a new name for an existing method.

Exclusion forms a new trait by removing a method from an existing trait. Combining the alias and exclusion operations yields a renaming operation, although the renaming is shallow.

The other important operation on traits is *inheritance*, the mechanism whereby traits are integrated with classes. This operation merges a class C, a trait, and additional fields and methods to form a new subclass of C. Often, the additional methods provide access to the newly added fields. The additional methods, plus the methods inherited from C, provide the required methods of the trait. An important aspect of traits is that the methods of a trait are only loosely coupled; they can be removed and replaced by other implementations. In this way traits are a lighter-weight mechanism than either multiple inheritance or mixins.

There is similarity between traits and the use of premethod collections to encode classes [AC96, RR96], but previous work on premethods focused on building a complete suite of methods and not on independent combinable traits. We have explored using the combination of modules, object types, and premethods to encode traits in Moby [FR03], but the encoding is cumbersome.

The term *traits* has been used in *delegation*-based (or *prototype*-based) languages, such as Self [US87], to describe objects that serve as repositories of methods. In Self, new objects are generated by cloning prototype objects, which, in turn, may delegate behavior to methods defined in trait objects. Like Smalltalk, Self is a dynamically typed language, so it does not address the issue of statically typing trait objects.

Bracha's Jigsaw framework is often cited as the first formal account of mixins [Bra92]. While his framework shares with traits the goal of replacing a monolithic class mechanism with simplier operators, it is a more powerful and complicated system with operators for global renaming of methods, static binding (or freezing), and visibility control. Traits can be viewed as a restricted subset of Jigsaw. While Bracha gave a dynamic semantics for Jigsaw and a type system, he did not prove type soundness.

The remainder of the paper is organized as follows. In the next section, we present the syntax and informal semantics of a typed object calculus that models traits. We take as a starting point for our calculus the untyped formal model of Smalltalk traits developed by Schärli *et al.* [SDN+02]. We then present a dynamic and static semantics for our calculus. In Section 5, we prove type soundness for our system using the standard technique of subject reduction and progress theorems. We conclude with a discussion of future directions.

2 A trait calculus

In this section, we present the syntax and informal semantics of a small calculus that models traits. Unlike the earlier work on traits, our calculus is designed to be statically typechecked. Because our main interest is in the factoring and assembly of classes, we have made simplifying choices in other areas of our design (*e.g.*, using functional objects and width-only subtyping).

2.1 Syntax

At the core of our calculus is a simple object-oriented language with immutable objects and first-class functions.[2] The language includes variables, function abstraction and application, object creation, self reference, method dispatch, super-method dispatch, field selection, and functional field update. The complete syntax and semantics of expressions are covered in Appendices A and C.

The most important parts of our calculus are the mechanisms for manipulating traits and classes and the corresponding type system. Before giving the syntax of these forms, we introduce some notation.

[2] We could have encoded functions as objects, but their presence clarifies the dynamic semantics.

Let $\mathcal{F}_U$ and $\mathcal{M}_U$ be disjoint, countable sets of field and method names, respectively. Collectively, we refer to method and field names as labels. We define the following notations:

f	$\in$	$\mathcal{F}_U$	field names
$\mathcal{F}$	$\overset{\text{fin}}{\subset}$	$\mathcal{F}_U$	finite sets of field names
m	$\in$	$\mathcal{M}_U$	method names
$\mathcal{M}$	$\overset{\text{fin}}{\subset}$	$\mathcal{M}_U$	finite sets of method names
$\mathcal{S}$	$\overset{\text{fin}}{\subset}$	$\mathcal{M}_U$	finite sets of super-method names
$\mathcal{L}_U$	$=$	$\mathcal{F}_U \cup \mathcal{M}_U$	universe of labels
l	$\in$	$\mathcal{L}_U$	labels (field or method names)
$\mathcal{L}$	$\overset{\text{fin}}{\subset}$	$\mathcal{L}_U$	finite sets of labels
$\mathcal{R}$	$\overset{\text{fin}}{\subset}$	$\mathcal{L}_U$	finite sets of required fields and methods

In addition, we assume disjoint, countable sets of trait names TNAME, class names CNAME, and expression variables VARIABLES. The syntax of our trait forms is taken from the formal model of Smalltalk traits developed by Schärli *et al.* [SDN+02].

T	$::=$	t	trait name: t $\in$ TNAME
	$\mid$	$\langle\!\langle\, M;\, \langle\theta\rangle;\, \mathcal{S}\,\rangle\!\rangle$	trait formation
	$\mid$	$T_1 + T_2$	symmetric concatenation
	$\mid$	$T - (m : \tau)$	method exclusion
	$\mid$	$T[m' \mapsto m]$	method alias
M	$::=$	$\langle {\mu_m}^{m \in \mathcal{M}} \rangle_{\mathtt{M}}$	method suite
μ	$::=$	$m\,(x : \tau) \Rightarrow e$	method definition

A trait expression can be the name of a previously defined trait, the formation of a base trait from a method suite and auxiliary information, the symmetric (or disjoint) concatenation of two traits, the exclusion of a method from a trait, or the addition of a method alias to a trait.

Our syntax differs from Schärli *et al.* in four significant ways. First, trait formation includes a row-type θ and a set of method names $\mathcal{S}$. The type θ specifies the names and types of the required methods and fields of the trait. The set $\mathcal{S}$ specifies the names of any super-methods that are referenced in M. This information is necessary to type check mixing a trait with a class. The second modification is that method exclusion includes a type annotation that specifies the type of the excluded method. This annotation is required for the evaluation of the exclusion operation, since an excluded method becomes required.[3] The third change is that we permit traits to require fields, *i.e.*, trait methods may reference fields directly, rather than using accessor functions. Traits still cannot define fields, however. As we will see below, allowing field references in traits provides a cleaner separation between traits and classes. The last difference is that our calculus does not have a direct analogue to the "D **with** T" form that adds a dictionary of methods D to a trait T, possibly overriding some of the methods in T. This effect can be achieved in our calculus using a combination of method removal and symmetric concatenation.

Our syntax for class definition is simple because we have stripped it of all but the features necessary to understand the interactions between classes and traits.

C	$::=$	c	class name: c $\in$ CNAME
	$\mid$	$\mathbf{nil}$	empty class
	$\mid$	$F\ \mathbf{in}\ T\ \mathbf{extends}\ c$	inheritance (subclass formation)
F	$::=$	$\langle f = e_f{}^{f \in \mathcal{F}} \rangle_{\mathtt{F}}$	fields definition

A class definition may be the name of an existing class, the empty class, or a new subclass formed by extending an existing class with a fields definition and a trait. A fields definition is a record of expressions labeled by field names. Our calculus does not model the passing of arguments at object construction time;

[3] This annotation could be avoided by adding return type annotations to methods or, perhaps, by changing the way we evaluate traits. We intend to explore these options in future work.

instead the values of fields are determined by the expressions in the fields definition. Note that in our minimal calculus, the definition of methods is left to traits. The class forms are concerned only with the inheritance hierarchy and field initialization.

This separation is not possible with Smalltalk traits because their trait methods cannot access instance variables (what we call fields), and instead must use accessor methods provided by host classes to reference instance variables. For example, to create a new class D, a Smalltalk user extends a class C with a trait T and a collection of instance variables and glue methods. These glue methods must provide the necessary access methods. While there may be pedagogical advantages to the Smalltalk design, we have chosen to allow field references in traits and to adopt the more restrictive form of class definition to simplify the calculus.

A program in our trait calculus is a sequence of trait and class declarations followed by an expression.

$$
\begin{array}{lcll}
P & ::= & D; P \quad | \quad e & \\
\\
D & ::= & t = T & \text{trait declaration} \\
 & | & c = C & \text{class declaration}
\end{array}
$$

Our type system distinguishes between two value types: function types and object types (θ). Object types assign types to a collection of field and method names. We use these types both to describe the type of object values and to specify aspects of traits and classes. Our type system also has types for field definitions, traits, and classes, although the corresponding terms are not first-class. The syntax of types is as follows:

$$
\begin{array}{lcll}
\theta & ::= & \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle & \text{row type} \\
\\
\tau & ::= & \tau_1 \rightarrow \tau_2 & \text{function type} \\
 & | & \theta & \text{object type} \\
 & | & \langle f : \tau_f{}^{f \in \mathcal{F}} \rangle_{\mathsf{F}} & \text{field record type} \\
 & | & \langle\!| \, \theta;\ \mathcal{S};\ \mathcal{R} \, |\!\rangle & \text{trait type} \\
 & | & \{\!| \, \theta \, |\!\} & \text{class type}
\end{array}
$$

In a trait type $\langle\!| \, \theta;\ \mathcal{S};\ \mathcal{R} \, |\!\rangle$, row type θ describes the types of all required and provided methods and fields. Set $\mathcal{S}$ contains the names of super-methods that must be available from any class that can be merged with the trait. The set $\mathcal{R}$ names the required methods and fields. A class type is similar to an object type in that it assigns types to the collection of fields and methods that it defines.

2.2 A small example

As an example of how traits are used in our calculus, consider the definition of a small two-level class hierarchy that contains a class `NameC` of objects with a `nameF` field and `showM` and `printM` methods, and a subclass `NamePC`, that overrides the `showM` method to parenthesize its result. The following code shows how this hierarchy is defined in our calculus:[4]

$$
\begin{array}{lcl}
\texttt{ShowNameT} & = & \langle\!| \, \langle \texttt{showM}() \Rightarrow \textbf{self}.\texttt{nameF} \rangle_{\mathsf{M}};\ \langle \texttt{nameF} : \texttt{string} \rangle;\ \{\} \, |\!\rangle \\
\texttt{PrintT} & = & \langle\!| \, \langle \texttt{printM}() \Rightarrow \texttt{print}(\textbf{self}.\texttt{showM}()) \rangle_{\mathsf{M}};\ \langle \texttt{showM} : () \rightarrow \texttt{string} \rangle;\ \{\} \, |\!\rangle \\
\texttt{NameC} & = & \langle \texttt{nameF} = \texttt{"Bob"} \rangle_{\mathsf{F}}\ \textbf{in}\ \texttt{ShowNameT} + \texttt{PrintT}\ \textbf{extends nil} \\
\texttt{ShowPT} & = & \langle\!| \, \langle \texttt{showM}() \Rightarrow \texttt{" ("} + \textbf{super}.\texttt{showM}() + \texttt{") "} \rangle_{\mathsf{M}};\ \langle\rangle;\ \{\texttt{showM}\} \, |\!\rangle \\
\texttt{NamePC} & = & \langle\rangle_{\mathsf{F}}\ \textbf{in}\ \texttt{ShowPT}\ \textbf{extends}\ \texttt{NameC}
\end{array}
$$

The `showNameT` trait defines the `showM` method for showing the name field of an object, while the `printT` trait defines the `printM` method for printing the result of `showM`. We concatenate these traits to form the method suite for the `NameC` class. The `NamePC` class is defined by using the `showPT` trait to override the `showM` method.

[4]For the purposes of this example, we have extended our calculus with strings, string concatenation, and a global `print` function.

Because we are using traits to factor the implementation, we can reuse our method definitions in a different class hierarchy. For example, we can define a class `HelloC` that is unrelated to the previous classes, but still shares the implementation of `printM`.

$$\texttt{HelloT} \;=\; \langle\!|\, \langle \texttt{showM}() \Rightarrow \texttt{"Hello, World!"} \rangle_{\texttt{M}};\ \langle\rangle;\ \{\} \,|\!\rangle$$

$$\texttt{HelloC} \;=\; \langle\rangle_{\texttt{F}}\ \textbf{in}\ \texttt{HelloT} + \texttt{PrintT}\ \textbf{extends nil}$$

3 Evaluation

We split the evaluation of programs in our calculus into two phases: *linking* and *execution*. In the linking phase, trait and class expressions are reduced to normal forms and bound to trait and class names in their respective environments. The resulting class environment is then used during the execution phase, which evaluates the body of the program. In this section, we describe the evaluation rules for our trait calculus. Our main focus is the linking phase, since traits have no run-time rôle in our system. We divide the linking phase into three parts: trait evaluation, class evaluation, and declaration evaluation.

3.1 Trait evaluation

Trait evaluation reduces trait expressions to trait values, which are of the form "$\langle\!|\, M;\ \theta;\ \mathcal{S} \,|\!\rangle$." We use TRAITVALS to denote the set of trait values and tv to denote elements of TRAITVALS. The trait evaluation judgment has the form "$\mathrm{TE} \vdash T \Rightarrow tv$," where TE is a trait environment (a finite map from trait names to trait values).

Trait variables are evaluated by looking them up in the trait environment.

$$\frac{t \in \mathrm{dom}(\mathrm{TE})}{\mathrm{TE} \vdash t \Rightarrow \mathrm{TE}(t)}$$

The trait formation expression is already a trait value.

$$\frac{}{\mathrm{TE} \vdash \langle\!|\, M;\ \theta;\ \mathcal{S} \,|\!\rangle \Rightarrow \langle\!|\, M;\ \theta;\ \mathcal{S} \,|\!\rangle}$$

The rule for symmetric concatenation of two traits ($T_1 + T_2$) forms a new trait value that is the union of T_1 and T_2's disjoint method suites.[5] Since some of the methods required by T_1 may be provided by T_2 (and *vice versa*), the set of required methods of the new trait is defined to be the union of T_1 and T_2's required methods after removing any overlap with the provided methods. The required super methods are the union of the super methods required by T_1 and T_2.

$$\frac{\begin{array}{c}\mathrm{TE} \vdash T_1 \Rightarrow \langle\!|\, \langle {\mu_m}^{m \in \mathcal{M}_1} \rangle_{\texttt{M}};\ \langle l : {\tau_l}^{l \in \mathcal{R}_1} \rangle;\ \mathcal{S}_1 \,|\!\rangle \quad \mathrm{TE} \vdash T_2 \Rightarrow \langle\!|\, \langle {\mu_m}^{m \in \mathcal{M}_2} \rangle_{\texttt{M}};\ \langle l : {\tau_l}^{l \in \mathcal{R}_2} \rangle;\ \mathcal{S}_2 \,|\!\rangle \\ \mathcal{M}_1 \pitchfork \mathcal{M}_2 \quad \mathcal{M}_3 = \mathcal{M}_1 \cup \mathcal{M}_2 \quad \mathcal{R}_3 = (\mathcal{R}_1 \cup \mathcal{R}_2) \setminus \mathcal{M}_3 \quad \mathcal{S}_3 = \mathcal{S}_1 \cup \mathcal{S}_2\end{array}}{\mathrm{TE} \vdash T_1 + T_2 \Rightarrow \langle\!|\, \langle {\mu_m}^{m \in \mathcal{M}_3} \rangle_{\texttt{M}};\ \langle l : {\tau_l}^{l \in \mathcal{R}_3} \rangle;\ \mathcal{S}_3 \,|\!\rangle}$$

Excluding a method from a trait causes its definition to be removed from the trait's methods, but it also causes the excluded method to be added to the list of required methods, which is necessary because the method may be mentioned in one of the trait's remaining methods.

$$\frac{\mathrm{TE} \vdash T \Rightarrow \langle\!|\, \langle {\mu_m}^{m \in \mathcal{M}} \rangle_{\texttt{M}};\ \langle l : {\tau_l}^{l \in \mathcal{R}} \rangle;\ \mathcal{S} \,|\!\rangle \qquad \tau_m = \tau \qquad m \in \mathcal{M}}{\mathrm{TE} \vdash T - (m : \tau) \Rightarrow \langle\!|\, \langle {\mu_m}^{m \in \mathcal{M} \setminus \{m\}} \rangle_{\texttt{M}};\ \langle l : {\tau_l}^{l \in \mathcal{R} \cup \{m\}} \rangle;\ \mathcal{S} \,|\!\rangle}$$

Lastly, the rule for method aliasing looks up the aliased method's definition, binds the definition to the new name `m'` and removes `m'` from the collection of required names.

$$\frac{\begin{array}{c}\mathrm{TE} \vdash T \Rightarrow \langle\!|\, \langle {\mu_m}^{m \in \mathcal{M}} \rangle_{\texttt{M}};\ \langle l : {\tau_l}^{l \in \mathcal{R}} \rangle;\ \mathcal{S} \,|\!\rangle \\ m \in \mathcal{M} \quad \mu_m = m\,(x : \tau) \Rightarrow e \quad m' \notin \mathcal{M}\end{array}}{\mathrm{TE} \vdash T[m' \mapsto m] \Rightarrow \langle\!|\, \langle {\mu_m}^{m \in \mathcal{M}},\ m'\,(x : \tau) \Rightarrow e \rangle_{\texttt{M}};\ \langle l : {\tau_l}^{l \in \mathcal{R} \setminus \{m'\}} \rangle;\ \mathcal{S} \,|\!\rangle}$$

[5] We use the notation $S_1 \pitchfork S_2$ to denote that sets S_1 and S_2 are disjoint.

3.2 Class evaluation

The second part of linking is the evaluation of class expressions. To express class evaluation, we add a new syntactic form for classes, called a ***flattened class***, which consists of a record of fields and a method suite.

$$\begin{array}{rcll} C & ::= & \ldots & \text{previous forms} \\ & | & \{\!|\, F;\ M \,|\!\} & \text{flattened class} \end{array}$$

Flattened classes serve as *class values* and are the result of evaluating a class expression. We use CLASSVALS to denote the set of class values and cv to denote an element of CLASSVALS. A class environment CE is a finite map from class names to class values. We write $\mathrm{TE}, \mathrm{CE} \vdash C \Rightarrow cv$ to mean that a class expression C evaluates to class value cv. We define this judgment form by the rules given below.

For class names, we lookup the class value in the class environment.

$$\frac{c \in \mathrm{dom}(\mathrm{CE})}{\mathrm{TE}, \mathrm{CE} \vdash c \Rightarrow \mathrm{CE}(c)}$$

The root class evaluates to the "empty" class value.

$$\mathrm{TE}, \mathrm{CE} \vdash \mathbf{nil} \Rightarrow \{\!|\, \langle\rangle_{\mathtt{F}};\ \langle\rangle_{\mathtt{M}} \,|\!\}$$

Forming a new subclass from the combination of a record of fields, a trait expression, and a superclass is the heart of our system. One complication in the assembly of a class is the static resolution of superclass methods. We address this complication by eliminating references to **super** in method bodies before constructing a subclass. The judgment form $M \vdash M_1 \Longrightarrow M_2$ specifies that the method suite M_1 is rewritten with respect to the super-class method suite M to produce the suite M_2. This rewriting has the effect of statically resolving super-method dispatch by replacing instances of super-class method dispatch with the application of the statically determined super-class method to self. The full details of this judgment form are described in Appendix B. The following rule defines the inheritance form:

$$\frac{\begin{array}{c} \mathrm{TE} \vdash T \Rightarrow \{\!|\, \langle {\mu'}_m{}^{m \in \mathcal{M}_T} \rangle_{\mathtt{M}};\ \langle l : {\tau_l}^{l \in \mathcal{R}_T} \rangle;\ \mathcal{S}_T \,|\!\} \\ \mathrm{TE}, \mathrm{CE} \vdash c \Rightarrow \{\!|\, \langle f = {e_f}^{f \in \mathcal{F}_c} \rangle_{\mathtt{F}};\ \langle {\mu_m}^{m \in \mathcal{M}_c} \rangle_{\mathtt{M}} \,|\!\} \\ \langle {\mu_m}^{m \in \mathcal{M}_c} \rangle_{\mathtt{M}} \vdash \langle {\mu'}_m{}^{m \in \mathcal{M}_T} \rangle_{\mathtt{M}} \Longrightarrow \langle {\mu_m}^{m \in \mathcal{M}_T} \rangle_{\mathtt{M}} \\ \mathcal{R}_T \subseteq \mathcal{F} \cup \mathcal{F}_c \cup \mathcal{M}_c \qquad \mathcal{S}_T \subseteq \mathcal{M}_c \end{array}}{\mathrm{TE}, \mathrm{CE} \vdash \langle f = {e_f}^{f \in \mathcal{F}} \rangle_{\mathtt{F}} \ \mathbf{in}\ T\ \mathbf{extends}\ c \Rightarrow \{\!|\, \langle f = {e_f}^{f \in \mathcal{F} \cup \mathcal{F}_c} \rangle_{\mathtt{F}};\ \langle {\mu_m}^{m \in \mathcal{M}_T \cup \mathcal{M}_c} \rangle_{\mathtt{M}} \,|\!\}}$$

The first two lines specify the normalization of the trait and superclass expressions. The third line rewrites the trait's methods to remove references to super by in-lining methods from `C`, and the fourth line ensures that the required and super methods are provided.

3.3 Declaration evaluation

The final part of linking is evaluation of trait and class declarations. The basic judgment form is $\mathrm{TE}, \mathrm{CE} \vdash D \Rightarrow \mathrm{TE}', \mathrm{CE}'$, which extends the trait and class environments TE and CE with the declaration D to produce TE' and CE'. The rules for this judgment can be found in Appendix C.

3.4 Program evaluation

The evaluation of programs combines the linking and execution phases into a single evaluation judgment. The definitions of the program evaluation judgments are left to Appendix C, but we summarize the notation here.

The judgment form "$\mathrm{TE}, \mathrm{CE} \vdash P \Rightarrow \mathrm{CE}', e$" says that linking the program P yields a pair of the class environment CE and the expression e. Executing an expression e for n steps is denoted $e \rightarrow^n_{\mathrm{CE}} e'$, where CE is the class environment used to define object creation and e' is the result of evaluation. Finally, the judgment "$\mathrm{TE}, \mathrm{CE} \vdash P \rightarrow^n e$" denotes linking the program P and then executing the resulting expression for n steps producing e as a result.

4 The static semantics of traits

In this section, we give an overview of the type system for our calculus, focusing on the typing rules for traits and classes. Appendix D contains the complete type system.

4.1 Contexts

All of our typing judgments are written in terms of a context Γ, which maps trait names, class names, and variables to associated types. We use the following typing judgments to formulate our type system:

$\Gamma \vdash ok$	well-formed context Γ
$\Gamma \vdash \tau$	well-formed type τ
$\Gamma \vdash R \ : \ \tau$	well-typed expression, trait, class, field record, or method
$\Gamma \vdash \tau_1 \ <: \ \tau_2$	type τ_1 is a subtype of type τ_2
$\Gamma \vdash D \ \Rightarrow \ \Gamma'$	well-formed declaration D, yielding extended environment Γ'
$\Gamma \vdash_P P \ : \ \tau$	well-typed program P

4.2 Trait typing

In this section, we describe selected rules for typing trait expressions. The first rule types trait formation.

$$\frac{\begin{array}{c}\tau_{\mathbf{super}} = \langle l : \tau_l{}^{l \in \mathcal{S}} \rangle \qquad \tau_{\mathbf{self}} = \langle l : \tau_l{}^{l \in \mathcal{M} \cup \mathcal{R}} \rangle \\ \Gamma,\ \mathbf{super} : \tau_{\mathbf{super}},\ \mathbf{self} : \tau_{\mathbf{self}} \vdash \mu_{m'} \ : \ \tau_{m'} \quad \text{forall } m' \in \mathcal{M} \\ \Gamma \vdash \tau_{l'} \quad \text{forall } l' \in \mathcal{R} \quad \Gamma \vdash ok \\ \Gamma \vdash \tau_{\mathbf{self}} \ <: \ \tau_{\mathbf{super}} \quad \mathcal{M} \pitchfork \mathcal{R}\end{array}}{\Gamma \vdash \langle\!| \langle \mu_m{}^{m \in \mathcal{M}} \rangle_{\mathtt{M}};\ \langle l : \tau_l{}^{l \in \mathcal{R}} \rangle;\ \mathcal{S} |\!\rangle \ : \ \langle\!| \tau_{\mathbf{self}};\ \mathcal{S};\ \mathcal{R} |\!\rangle}$$

This rule checks each of the method bodies in the trait's method suite under assumptions about the type of $\mathbf{super}$ and $\mathbf{self}$. The type of $\mathbf{self}$, $\tau_{\mathbf{self}}$, contains each of the methods of the method suite with the type inferred for that method. It also contains all of the required methods of the trait with their programmer-supplied types. The type of $\mathbf{super}$ is the restriction of $\tau_{\mathbf{self}}$ to methods declared to come from the superclass. The inference rule also ensures that the types of the required methods are well-formed, that the type of $\mathbf{self}$ is a subtype of the type of $\mathbf{super}$, and that no method implemented in the trait is marked as required. The resulting type for the trait is a triple of the types of all of the trait's fields and methods (both provided and required), the set of methods that must be provided by any host superclass ($\mathcal{S}$), and the set of required methods ($\mathcal{R}$).

The rule for type checking symmetric concatenation of two traits is

$$\frac{\begin{array}{c}\Gamma \vdash T_1 \ : \ \langle\!| \langle l : \tau_l{}^{l \in \mathcal{L}_1} \rangle;\ \mathcal{S}_1;\ \mathcal{R}_1 |\!\rangle \\ \Gamma \vdash T_2 \ : \ \langle\!| \langle l : \tau_l{}^{l \in \mathcal{L}_2} \rangle;\ \mathcal{S}_2;\ \mathcal{R}_2 |\!\rangle \\ \mathcal{M}_1 = \mathcal{L}_1 \setminus \mathcal{R}_1 \quad \mathcal{M}_2 = \mathcal{L}_2 \setminus \mathcal{R}_2 \quad \mathcal{M}_1 \pitchfork \mathcal{M}_2 \\ \mathcal{R}'_1 = \mathcal{R}_1 \setminus \mathcal{M}_2 \quad \mathcal{R}'_2 = \mathcal{R}_2 \setminus \mathcal{M}_1\end{array}}{\Gamma \vdash T_1 + T_2 \ : \ \langle\!| \langle l : \tau_l{}^{l \in \mathcal{L}_1 \cup \mathcal{L}_2} \rangle;\ \mathcal{S}_1 \cup \mathcal{S}_2;\ \mathcal{R}'_1 \cup \mathcal{R}'_2 |\!\rangle}$$

This rule requires that the methods provided by the two traits are disjoint ($\mathcal{M}_1 \ \pitchfork \ \mathcal{M}_2$). The new collection of required methods is the union of the methods required by T_1 but not implemented in T_2 (*i.e.*, $\mathcal{R}'_1$) and those required by T_2 but not implemented in T_1 (*i.e.*, $\mathcal{R}'_2$). Shared abstract methods are required to have the same type.

To type check method exclusion, we ensure that the method being removed was provided by the trait (*i.e.*, $m \in \mathcal{L} \setminus \{m\}$).

$$\frac{\Gamma \vdash T \ : \ \langle\!| \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle;\ \mathcal{S};\ \mathcal{R} |\!\rangle \quad m \in \mathcal{L} \setminus \mathcal{R} \quad \tau_m = \tau}{\Gamma \vdash T - (m : \tau) \ : \ \langle\!| \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle;\ \mathcal{S};\ \mathcal{R} \cup \{m\} |\!\rangle}$$

The evaluation semantics requires that the method being removed be annotated with its type. Here, we check that the type annotation matches the type of the method in the trait.

Finally, the typing rule for method aliasing checks that the new name m' does not have a binding ($m' \notin \mathcal{L} \setminus \mathcal{R}$), while ensuring that the old name m does have one ($m \in \mathcal{L} \setminus \mathcal{R}$).

$$\frac{\Gamma \vdash T \ : \ \langle\!| \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle; \ \mathcal{S}; \ \mathcal{R} \ |\!\rangle \quad m \in \mathcal{L} \setminus \mathcal{R} \quad m' \notin \mathcal{L} \setminus \mathcal{R} \quad \tau_{m'} = \tau_m}{\Gamma \vdash T[m' \mapsto m] \ : \ \langle\!| \langle l : \tau_l{}^{l \in \mathcal{L} \cup \{m'\}} \rangle; \ \mathcal{S}; \ \mathcal{R} \setminus \{m'\} \ |\!\rangle}$$

The requirement that $\tau_{m'} = \tau_m$ is subtle: if $m' \in \mathcal{L}$, then the condition requires that the type given for m' in the type for T match the type τ_m. If $m' \notin \mathcal{L}$, then the condition defines $\tau_{m'}$ to be τ_m.

4.3 Class typing

The key typing rule for classes is the rule for subclass formation.

$$\frac{\begin{array}{c}\Gamma \vdash F \ : \ \langle f : \tau_f{}^{f \in \mathcal{F}} \rangle_{\mathsf{F}} \\ \Gamma \vdash T \ : \ \langle\!| \langle l : \tau_l{}^{l \in \mathcal{L}_T} \rangle; \ \mathcal{S}_T; \ \mathcal{R}_T \ |\!\rangle \\ \Gamma \vdash c \ : \ \{\!| \langle l : \tau_l{}^{l \in \mathcal{L}_c} \rangle \ |\!\} \\ \mathcal{S}_T \subseteq \mathcal{L}_c \qquad \mathcal{R}_T \subseteq (\mathcal{L}_c \cup \mathcal{F}) \quad \mathcal{F} \pitchfork \mathcal{L}_c \qquad \mathcal{L} = \mathcal{F} \cup \mathcal{L}_T \cup \mathcal{L}_c\end{array}}{\Gamma \vdash F \ \mathbf{in} \ T \ \mathbf{extends} \ c \ : \ \{\!| \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle \ |\!\}}$$

This rule infers types for the field definitions F, trait T, and class c that are being merged to form a new class and checks that these pieces fit together appropriately. To verify that c satisfies all of the trait's superclass requirements, we check that $\mathcal{S}_T \subseteq \mathcal{L}_c$. Condition $\mathcal{R}_T \subseteq (\mathcal{L}_c \cup \mathcal{F})$ ensures that either F or c provides all the methods and fields required by the trait. To guarantee that new fields do not conflict with existing fields, we check that $\mathcal{F} \pitchfork \mathcal{L}_c$. The label set $\mathcal{L}$ collects together the names of all the fields and methods of the new class. The formation of the class type ensures that if the trait requires a given field or method l with a type τ_l, then the supplier of l (either F or c) must give the syntactically identical type to l.

5 Type soundness

We prove type soundness using the standard technique of subject reduction and progress theoreoms. To establish both of these theorems, we need to introduce an auxiliary judgment to relate trait and class environments with a typing context:

$$\frac{\begin{array}{ll}\Gamma \vdash \mathrm{TE}(t) \ : \ \Gamma(t) & \text{forall } t \in \mathrm{dom}(\mathrm{TE}) \\ \Gamma \vdash \mathrm{CE}(c) \ : \ \Gamma(c) & \text{forall } c \in \mathrm{dom}(\mathrm{CE})\end{array}}{\Gamma \models \mathrm{TE}, \ \mathrm{CE}}$$

In addition, we need to define what it means for one context to refine another.

Definition 5.1 *Context Γ' refines Γ, written $\Gamma' \leq \Gamma$, if the following conditions hold: $\Gamma' \vdash ok$, $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\Gamma')$, and for all $id \in \mathrm{dom}(\Gamma)$, $\Gamma' \vdash \Gamma'(id) \ <: \ \Gamma(id)$.*

Given these definitions, we can state subject reduction and progress theorems, from which a type soundness theorem follows directly.

Theorem 5.1 (Subject Reduction) *If we may derive a typing $\Gamma \vdash_P P \ : \ \tau$, and an evaluation $\mathrm{TE}, \mathrm{CE} \vdash P \rightarrow^n e$, and show that $\Gamma \models \mathrm{TE}, \ \mathrm{CE}$ where $\Gamma \pitchfork$ VARIABLES, then there exists a context Γ' such that $\Gamma' \leq \Gamma$ and $\Gamma' \vdash e \ : \ \tau$.*

Theorem 5.2 (Progress) *If we may derive a typing $\Gamma \vdash_P P \ : \ \tau$ where $\Gamma \pitchfork$ VARIABLES and show that $\Gamma \models \mathrm{TE}, \ \mathrm{CE}$ for trait and class environments TE and CE, then either there exists a natural number n such that $\mathrm{TE}, \ \mathrm{CE} \vdash P \rightarrow^n ev$ or for all n there exists an expression e such that $\mathrm{TE}, \ \mathrm{CE} \vdash P \rightarrow^n e$.*

Theorem 5.3 (Type Soundness) *If P is a closed, well-typed program ($\epsilon \vdash_P P : \tau$), then either P evaluates to a value with type τ ($\emptyset, \emptyset \vdash P \rightarrow^n ev$ for some n and there exists a context Γ $\Cap$ VARIABLES such that $\Gamma \vdash ev : \tau$) or P runs forever (for all natural numbers n, we may derive an evaluation $\emptyset, \emptyset \vdash P \rightarrow^n e$).*

6 Conclusion

Traits are a promising new mechanism for constructing class hierarchies from reusable components [SDNB03]. While this mechanism has been designed for Smalltalk, we expect it to be useful for statically-typed object-oriented programming languages as well.

This paper is the first step in developing statically-typed traits as a programming language mechanism. In it, we have presented a statically-typed calculus of traits and classes and have shown type soundness for our calculus. There are a number of refinements to this calculus that we plan to explore and we discuss these in the remainder of this section.

We have purposefully omitted a number of common features from our calculus in the interest of simplicity. These features include depth subtyping, imperative objects, and mechanisms for object initialization. We do not expect any of these features to have a significant interaction with traits, but adding them to the calculus will move our model closer to practical programming languages. Another feature that we omitted is any form of privacy. In our previous work, we support privacy using signature ascription at the module level [FR99]. In principal, this technique should apply to our trait calculus, but we have not worked out the details of trait signatures.

The typing system for traits maintains information about the required fields and methods. In our system, we associate this information with individual traits, but it would also be possible to keep per-method information about required fields and methods, which would improve the typing precision for method exclusion.[6]

In our current design, the addition of types does limit code reuse in some situations. For example, Schärli *et al.* give an example of a synchronization trait that overrides read and write methods to implement synchronized read and write methods [SDNB03]. While we can define such a trait in our calculus, the types of the read and write methods in the trait restrict the trait to similarly typed classes (*e.g.,* we cannot define a synchronization trait that works for both string and integer-valued read/write methods). The obvious solution to this problem is some form of parametric polymorphism, which could be provided either by allowing traits to be parameterized over types or by the use of parameterized modules.

A more speculative direction is to make traits and classes first-class. We have not yet explored this possibility in detail.

References

[AC96] Abadi, M. and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, NY, 1996.

[AG98] Arnold, K. and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 2nd edition, 1998.

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

[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. (to appear).

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

[6] Of course, a programming style in which all traits have a single method achieves the same result.

[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. Object-oriented aspects of Moby. *Technical Report TR-2003-10*, Dept. of Computer Science, U. of Chicago, Chicago, IL, September 2003.

[OAC+03] Odersky, M., P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. *The Scala Language Specification (Draft)*. Switzerland, October 2003. Available from `lamp.epfl.ch/scala`.

[RR96] Reppy, J. H. and J. G. Riecke. Classes in Object ML via modules. In *FOOL3*, July 1996.

[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*, LNCS, New York, NY, July 2003. Springer-Verlag. (to appear).

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

[US87] Ungar, D. and R. B. Smith. Self: The power of simplicity. In *OOPSLA'87*, October 1987, pp. 227–242.

A The expression language

This appendix gives the syntax for the expression language underlying our trait-calculus. We assume a countable set of variables VARIABLES and use x to denote an arbitrary variable. The syntax of the language is as follows:

e	$::=$	x	variable: x $\in$ VARIABLES
	$\mid$	$\lambda(x:\tau).e$	function abstraction
	$\mid$	$e_1 e_2$	function application
	$\mid$	$\mathbf{new}\ c$	object instantiation
	$\mid$	$\mathbf{self}$	host object
	$\mid$	$e.m$	method dispatch
	$\mid$	$\mathbf{super}.m$	super-method dispatch
	$\mid$	$e.f$	field selection
	$\mid$	$e_1.f := e_2$	field update

The language contains variables (x), function abstraction and application, object creation, self reference, method dispatch (including super-method dispatch), field selection, and field update. Because objects are immutable, updating a field f does not modify the state of its argument, but rather produces a new object with the same field values except for f, which holds the new value.

B Method-suite rewriting

In this appendix, we present the details a method suite rewriting, which is used to resolve super-method dispatch statically. To eliminate references to $\mathbf{super}$, we rewrite method bodies before they are installed in classes, essentially inlining the superclass's method bodies. We rewrite method bodies with respect to a superclass method suite. For this purpose, we treat method suites as finite maps from their index set to their method bodies, *i.e.*, $\langle {\mu_m}^{m\in\mathcal{M}}\rangle_{\mathtt{M}}(m') = \mu_{m'}$ as long as $m' \in \mathcal{M}$.

Method-suite rewriting judgment forms:

$M \vdash M_1 \Longrightarrow M_2$	Method suite rewriting
$M \vdash m\,(x:\tau) \Rightarrow e_1 \Longrightarrow m\,(x:\tau) \Rightarrow e_2$	Method body rewriting
$M \vdash e_1 \Longrightarrow e_2$	Expression rewriting

We have the following rules for rewriting terms to remove **super** references:

$$\frac{M \vdash \mu_{m'} \Longrightarrow \mu'_{m'} \quad \text{forall } m' \in \mathcal{M}}{M \vdash \langle {\mu_m}^{m \in \mathcal{M}} \rangle_{\mathtt{M}} \Longrightarrow \langle {\mu'}_m{}^{m \in \mathcal{M}} \rangle_{\mathtt{M}}}$$

$$\frac{M \vdash e_1 \Longrightarrow e_2}{M \vdash m\,(x : \tau) \Rightarrow e_1 \Longrightarrow m\,(x : \tau) \Rightarrow e_2}$$

$$M \vdash x \Longrightarrow x$$

$$\frac{M \vdash e_1 \Longrightarrow e_2}{M \vdash \lambda(x : \tau).e_1 \Longrightarrow \lambda(x : \tau).e_2}$$

$$\frac{M \vdash e_1 \Longrightarrow e'_1 \qquad M \vdash e_2 \Longrightarrow e'_2}{M \vdash e_1 e_2 \Longrightarrow e'_1 e'_2}$$

$$M \vdash \mathbf{new}\ c \Longrightarrow \mathbf{new}\ c$$

$$M \vdash \mathbf{self} \Longrightarrow \mathbf{self}$$

$$\frac{M \vdash e_1 \Longrightarrow e_2}{M \vdash e_1.m \Longrightarrow e_2.m}$$

$$\frac{M(m) = m\,(x : \tau) \Rightarrow e}{M \vdash \mathbf{super}.m \Longrightarrow \lambda(x : \tau).e}$$

$$\frac{M \vdash e_1 \Longrightarrow e_2}{M \vdash e_1.f \Longrightarrow e_2.f}$$

$$\frac{M \vdash e_1 \Longrightarrow e'_1 \qquad M \vdash e_2 \Longrightarrow e'_2}{M \vdash e_1.f := e_2 \Longrightarrow e'_1.f := e'_2}$$

C Other evaluation rules

This appendix collects together those evaluation rules that were omitted from the main text.

C.1 Declaration evaluation rules

$$\frac{\mathrm{TE} \vdash T \Rightarrow tv \qquad t \notin \mathrm{dom}(\mathrm{TE})}{\mathrm{TE},\ \mathrm{CE} \vdash t = T \Rightarrow \mathrm{TE} \cup \{t \mapsto tv\},\ \mathrm{CE}}$$

$$\frac{\mathrm{TE},\ \mathrm{CE} \vdash C \Rightarrow cv \qquad c \notin \mathrm{dom}(\mathrm{CE})}{\mathrm{TE},\ \mathrm{CE} \vdash c = C \Rightarrow \mathrm{TE},\ \mathrm{CE} \cup \{c \mapsto cv\},}$$

C.2 Program linking rules

$$\frac{\mathrm{TE},\ \mathrm{CE} \vdash D \Rightarrow \mathrm{TE}',\ \mathrm{CE}' \quad \mathrm{TE}',\ \mathrm{CE}' \vdash P \Rightarrow \mathrm{TE}'',\ \mathrm{CE}'',\ e}{\mathrm{TE},\ \mathrm{CE} \vdash D; P \Rightarrow \mathrm{TE}'',\ \mathrm{CE}'',\ e}$$

$$\mathrm{TE},\ \mathrm{CE} \vdash e \Rightarrow \mathrm{TE},\ \mathrm{CE},\ e$$

C.3 Expression evaluation rules

To evaluate expressions, we need to add a run-time form to represent dyamic objects:

e	::=	...	previous forms
	|	$\langle F;\ M \rangle$	dynamic object
ev	=	$\lambda(x : \tau).e \mid \langle fv;\ M \rangle$	
fv	=	$\langle f = {ev_f}^{f \in \mathcal{F}} \rangle_{\mathtt{F}}$	

Evaluation contexts:

$$\begin{array}{rcl} E & ::= & [\,] \mid E\,e \mid ev\,E \mid E.l \mid E.f := e \mid ev.f := E \\ & \mid & \langle f = {ev_f}^{f \in \mathcal{F}_1}, f = E, f = {e_f}^{f \in \mathcal{F}_2} \rangle_{\mathtt{F}} \mid \langle E;\ M \rangle \end{array}$$

Redexes :

$$\begin{array}{rcll} r & ::= & \lambda(x:\tau).e_1\ ev & \\ & | & \mathbf{new}\ c & \\ & | & \langle F;\ M\rangle.m & m \in \mathrm{dom}(M) \\ & | & \langle\langle f = ev_f{}^{f\in\mathcal{F}}\rangle_{\mathsf{F}};\ M\rangle.f' & f' \in \mathcal{F} \\ & | & \langle\langle f = ev_f{}^{f\in\mathcal{F}}\rangle_{\mathsf{F}};\ M\rangle.f' := ev & f' \in \mathcal{F} \end{array}$$

We use the notation $e_2[x \mapsto e_1]$ to denote the capture-avoiding substitution of e_1 for x in e_2.

Evaluation axioms:

$$\begin{array}{rcll} \lambda(x:\tau).e_1\ \ ev & \to_{CE} & e_1[x \mapsto ev] & \\ \mathbf{new}\ c & \to_{CE} & \langle F;\ M\rangle & \text{if } \mathrm{CE}(c) = \{\!|\ F;\ M\ |\!\} \\ \langle F;\ M\rangle.m & \to_{CE} & \lambda(x:\tau).e[\mathbf{self} \mapsto \langle F;\ M\rangle] & \text{if } M(m) = m\,(x:\tau) \Rightarrow e \\ \langle\langle f = ev_f{}^{f\in\mathcal{F}}\rangle_{\mathsf{F}};\ M\rangle.f' & \to_{CE} & ev_{f'} & \text{if } f' \in \mathcal{F} \\ \langle\langle f = ev_f{}^{f\in\mathcal{F}}\rangle_{\mathsf{F}};\ M\rangle.f' := ev & \to_{CE} & \langle\langle f = ev_f{}^{f\in\mathcal{F}\setminus\{f'\}},\ f' = ev\rangle_{\mathsf{F}};\ M\rangle & \text{if } f' \in \mathcal{F} \end{array}$$

Congruence closure:

$$\frac{e = E[r] \qquad r \to_{CE} r' \qquad e' = E[r']}{e \to_{CE} e'}$$

Note: Superclass invocations have been inlined during class evaluation.

The notation E[r] denotes filling the hole in context E with redex r. We use the notation $e \to^n_{CE} e'$ to denote that under class environment CE, expression e reduces to e' in n steps.

C.4 Program evaluation rules

$$\frac{\mathrm{TE},\ \mathrm{CE} \vdash P \Rightarrow \mathrm{TE}',\ \mathrm{CE}',\ e \quad e \to^n_{\mathrm{CE}'} e'}{\mathrm{TE},\ \mathrm{CE} \vdash P \to^n e'}$$

D The type system

This appendix gives the details of the type system for our calculus. All of our typing judgments are written in terms of contexts, which map trait names, class names, and variables to associated types.

$$\begin{array}{rcl} id & \in & \textsc{TName} \cup \textsc{CName} \cup \textsc{Variables} \\ \Gamma & ::= & \epsilon \mid \Gamma,\ id : \tau \end{array}$$

For convenience, we often treat the keywords **self** and **super** as variable names and allow them to be bound in contexts. We assume that the sets TNAME, CNAME, and VARIABLES are mutually disjoint. We use R as a meta-variable ranging over expressions, traits, classes, field records, and methods:

$$R \quad ::= \quad e \mid F \mid T \mid C \mid \mu$$

The rules for context formation are standard and we omit them here.

D.1 Well-formed types

$$\frac{\Gamma \vdash \tau_1 \quad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash \tau_f \quad \text{forall } f \in \mathcal{F} \quad \Gamma \vdash ok}{\Gamma \vdash \langle f : \tau_f{}^{f\in\mathcal{F}}\rangle_{\mathsf{F}}}$$

$$\frac{\Gamma \vdash \tau_l \quad \text{forall } l \in \mathcal{L} \quad \Gamma \vdash ok}{\Gamma \vdash \langle l : \tau_l{}^{l\in\mathcal{L}}\rangle} \qquad \frac{\Gamma \vdash \langle l : \tau_l{}^{l\in\mathcal{L}}\rangle \quad \mathcal{S} \cup \mathcal{R} \subseteq \mathcal{L}}{\Gamma \vdash \{\!|\ \langle l : \tau_l{}^{l\in\mathcal{L}}\rangle;\ \mathcal{S};\ \mathcal{R}\ |\!\}}$$

$$\frac{\Gamma \vdash \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle}{\Gamma \vdash \{\!|\, \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle \,|\!\}}$$

D.2 Subtyping

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \tau \mathrel{<:} \tau} \qquad \frac{\Gamma \vdash \langle l : \tau_l{}^{l \in \mathcal{L}_1} \rangle \qquad \mathcal{L}_2 \subseteq \mathcal{L}_1}{\Gamma \vdash \langle l : \tau_l{}^{l \in \mathcal{L}_1} \rangle \mathrel{<:} \langle l : \tau_l{}^{l \in \mathcal{L}_2} \rangle}$$

$$\frac{\Gamma \vdash \tau_2' \mathrel{<:} \tau_1' \quad \Gamma \vdash \tau_1'' \mathrel{<:} \tau_2''}{\Gamma \vdash \tau_1' \to \tau_1'' \mathrel{<:} \tau_2' \to \tau_2''}$$

D.3 Well-typed expressions

$$\frac{\Gamma \vdash ok}{\Gamma \vdash x \,:\, \Gamma(x)} \qquad \frac{\Gamma \vdash e \,:\, \tau \qquad \Gamma \vdash \tau \mathrel{<:} \tau'}{\Gamma \vdash e \,:\, \tau'}$$

$$\frac{\Gamma,\, x : \tau_1 \vdash e \,:\, \tau_2}{\Gamma \vdash \lambda(x : \tau_1).e \,:\, \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 \,:\, \tau_2 \to \tau \quad \Gamma \vdash e_2 \,:\, \tau_2}{\Gamma \vdash e_1\, e_2 \,:\, \tau}$$

$$\frac{\Gamma \vdash ok \quad \Gamma(c) = \{\!|\, \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle \,|\!\}}{\Gamma \vdash \mathbf{new}\ c \,:\, \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle} \qquad \frac{\Gamma \vdash e \,:\, \tau \quad \Gamma \vdash \tau \mathrel{<:} \langle l : \tau_l \rangle}{\Gamma \vdash e.l \,:\, \tau_l}$$

$$\frac{\Gamma \vdash e_1 \,:\, \tau \quad \Gamma \vdash \tau \mathrel{<:} \langle f : \tau_f \rangle \quad \Gamma \vdash e_2 \,:\, \tau_f}{\Gamma \vdash e_1.f := e_2 \,:\, \tau}$$

$$\frac{\begin{array}{c}\Gamma \vdash F \,:\, \langle f : \tau_f{}^{f \in \mathcal{F}} \rangle_{\mathtt{F}} \qquad \tau_{\mathbf{self}} = \langle l : \tau_l{}^{l \in \mathcal{F} \cup \mathcal{M}} \rangle \\ \Gamma,\, \mathbf{self} : \tau_{\mathbf{self}} \vdash \mu_{m'} \,:\, \tau_{m'} \quad \text{forall } m' \in \mathcal{M}\end{array}}{\Gamma \vdash \langle F;\ \langle \mu_m{}^{m \in \mathcal{M}} \rangle_{\mathtt{M}} \rangle \,:\, \tau_{\mathbf{self}}}$$

D.4 Field record typing

$$\frac{\Gamma \vdash e_f \,:\, \tau_f \quad \text{forall } f \in \mathcal{F} \quad \Gamma \vdash ok}{\Gamma \vdash \langle f = e_f{}^{f \in \mathcal{F}} \rangle_{\mathtt{F}} \,:\, \langle f : \tau_f{}^{f \in \mathcal{F}} \rangle_{\mathtt{F}}}$$

D.5 Method body typing

$$\frac{\Gamma,\, x : \tau \vdash e \,:\, \tau'}{\Gamma \vdash m\,(x : \tau) \Rightarrow e \,:\, \tau \to \tau'}$$

D.6 Trait typing

$$\frac{\Gamma \vdash ok}{\Gamma \vdash t \,:\, \Gamma(t)}$$

$$\frac{\begin{array}{c}\tau_{\mathbf{super}} = \langle l : \tau_l{}^{l \in \mathcal{S}} \rangle \qquad \tau_{\mathbf{self}} = \langle l : \tau_l{}^{l \in \mathcal{M} \cup \mathcal{R}} \rangle \\ \Gamma,\ \mathbf{super} : \tau_{\mathbf{super}},\ \mathbf{self} : \tau_{\mathbf{self}} \vdash \mu_{m'} \ :\ \tau_{m'} \quad \text{forall } m' \in \mathcal{M} \\ \Gamma \vdash \tau_{l'} \quad \text{forall } l' \in \mathcal{R} \quad \Gamma \vdash ok \\ \Gamma \vdash \tau_{\mathbf{self}} \ <:\ \tau_{\mathbf{super}} \quad \mathcal{M} \pitchfork \mathcal{R}\end{array}}{\Gamma \vdash \langle\!| \langle \mu_m{}^{m \in \mathcal{M}} \rangle_{\mathtt{M}};\ \langle l : \tau_l{}^{l \in \mathcal{R}} \rangle;\ \mathcal{S} |\!\rangle \ :\ \langle\!| \tau_{\mathbf{self}};\ \mathcal{S};\ \mathcal{R} |\!\rangle}$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 \ :\ \langle\!| \langle l : \tau_l{}^{l \in \mathcal{L}_1} \rangle;\ \mathcal{S}_1;\ \mathcal{R}_1 |\!\rangle \quad \Gamma \vdash T_2 \ :\ \langle\!| \langle l : \tau_l{}^{l \in \mathcal{L}_2} \rangle;\ \mathcal{S}_2;\ \mathcal{R}_2 |\!\rangle \\ \mathcal{M}_1 = \mathcal{L}_1 \setminus \mathcal{R}_1 \quad \mathcal{M}_2 = \mathcal{L}_2 \setminus \mathcal{R}_2 \quad \mathcal{M}_1 \pitchfork \mathcal{M}_2 \\ \mathcal{R}'_1 = \mathcal{R}_1 \setminus \mathcal{M}_2 \quad \mathcal{R}'_2 = \mathcal{R}_2 \setminus \mathcal{M}_1\end{array}}{\Gamma \vdash T_1 + T_2 \ :\ \langle\!| \langle l : \tau_l{}^{l \in \mathcal{L}_1 \cup \mathcal{L}_2} \rangle;\ \mathcal{S}_1 \cup \mathcal{S}_2;\ \mathcal{R}'_1 \cup \mathcal{R}'_2 |\!\rangle}$$

$$\frac{\Gamma \vdash T \ :\ \langle\!| \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle;\ \mathcal{S};\ \mathcal{R} |\!\rangle \quad m \in \mathcal{L} \setminus \mathcal{R} \quad \tau_m = \tau}{\Gamma \vdash T - (m : \tau) \ :\ \langle\!| \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle;\ \mathcal{S};\ \mathcal{R} \cup \{m\} |\!\rangle}$$

$$\frac{\Gamma \vdash T \ :\ \langle\!| \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle;\ \mathcal{S};\ \mathcal{R} |\!\rangle \quad m \in \mathcal{L} \setminus \mathcal{R} \quad m' \notin \mathcal{L} \setminus \mathcal{R} \quad \tau_{m'} = \tau_m}{\Gamma \vdash T[m' \mapsto m] \ :\ \langle\!| \langle l : \tau_l{}^{l \in \mathcal{L} \cup \{m'\}} \rangle;\ \mathcal{S};\ \mathcal{R} \setminus \{m'\} |\!\rangle}$$

D.7 Class typing

$$\frac{\Gamma \vdash ok}{\Gamma \vdash c \ :\ \Gamma(c)} \qquad \frac{\Gamma \vdash ok}{\Gamma \vdash \mathbf{nil} \ :\ \{\!| \langle\rangle |\!\}}$$

$$\frac{\begin{array}{c}\Gamma \vdash F \ :\ \langle f : \tau_f{}^{f \in \mathcal{F}} \rangle_{\mathtt{F}} \qquad \Gamma \vdash T \ :\ \langle\!| \langle l : \tau_l{}^{l \in \mathcal{L}_T} \rangle;\ \mathcal{S}_T;\ \mathcal{R}_T |\!\rangle \qquad \Gamma \vdash c \ :\ \{\!| \langle l : \tau_l{}^{l \in \mathcal{L}_c} \rangle |\!\} \\ \mathcal{S}_T \subseteq \mathcal{L}_c \qquad \mathcal{R}_T \subseteq (\mathcal{L}_c \cup \mathcal{F}) \qquad \mathcal{F} \pitchfork \mathcal{L}_c \qquad \mathcal{L} = \mathcal{F} \cup \mathcal{L}_T \cup \mathcal{L}_c\end{array}}{\Gamma \vdash F\ \mathbf{in}\ T\ \mathbf{extends}\ c \ :\ \{\!| \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle |\!\}}$$

$$\frac{\begin{array}{c}\Gamma \vdash F \ :\ \langle f : \tau_f{}^{f \in \mathcal{F}} \rangle_{\mathtt{F}} \qquad \tau_{\mathbf{self}} = \langle l : \tau l^{l \in \mathcal{F} \cup \mathcal{M}} \rangle \\ \Gamma,\ \mathbf{self} : \tau_{\mathbf{self}} \vdash \mu_{m'} \ :\ \tau_{m'} \quad \text{forall } m' \in \mathcal{M}\end{array}}{\Gamma \vdash \{\!| F;\ \langle \mu_m{}^{m \in \mathcal{M}} \rangle_{\mathtt{M}} |\!\} \ :\ \{\!| \tau_{\mathbf{self}} |\!\}}$$

D.8 Declaration typing

$$\frac{t \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash T \ :\ \tau}{\Gamma \vdash t = T \ \Rightarrow\ \Gamma,\ t : \tau} \qquad \frac{c \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash C \ :\ \tau}{\Gamma \vdash c = C \ \Rightarrow\ \Gamma,\ c : \tau}$$

D.9 Program typing

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