Statically typed traits

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

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

December 7, 2003

Abstract

Traits are mechanism, recently proposed by Scharli et al, for factoring Smalltalk class hierarchies. By separating the issue of code reuse from the inheritance hierarchy, traits allow one to avoid problems with methods defined too high in the inheritance hierarchy while sharing common implementation. Early experience with traits in Smalltalk shows that traits are an effective way to improve code sharing in class hierarchies. This positive experience with traits in the untyped Smalltalk world suggests that traits may be useful in statically typed languages too. In this paper, we present a statically typed calculus of traits, classes, and objects, which can serve as the foundation for extending statically-typed class-based languages, such as JAVA, with traits.

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 and may be useful for statically typed languages too. The main contribution of this paper is to present a typed calculus of traits that can serve as the foundation for integrating traits into a statically-typed object-oriented language such as JAVA [AG98] or Moby [FR99, FR03].

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]

[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.

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.

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.

The remainder of the paper is organized as follows. In the next section, we briefly review the motivation given by Schärli *et al.* for traits as a language feature and give an informal example illustrating the utility of traits. We then present the syntax and semantics of our typed trait calculus in Section 3. In Section 4, we prove type soundness for our system using the standard technique of subject reduction and progress theorems. We conclude with a discussion of related work and future directions.

2 Background

While the purpose of this paper is not to argue the merits of traits *per se* (see [SDNB03, BSD03] for such arguments), it is helpful to understand the motivation for traits. In languages with single inheritance, such as Smalltalk, it is often the case that inheritance does not provide sufficient flexibility for 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. If this protocol is not implemented by a common superclass, then each class must provide its own implementation, which results in code duplication. On the other hand, if we lift the implementation of the protocol up to the common superclass, we pollute the interface of the superclass, which affects all of its subclasses. Furthermore, if the protocol is defined by building on methods defined in intermediate classes, we will have to add these methods to the common superclass as well. This problem results from the dual nature of classes. Classes serve as both object generators and as superclasses. In the former case, the implementation of a class must be complete, whereas in the latter case the class implementation may have *abstract* methods that are implemented by a subclass. Traits provide a mechanism to separate these two aspects of classes and allow code to be reused across the class hierarchy. Multiple inheritance [Str94] and mixins [BC90, FKF98] represent two other attempts to solve this problem, but they both introduce semantic complexities and ambiguities (*e.g.*, multiple copies of instance variables in the case of multiple inheritance, and problems with the order of method definition in the case of mixins) [SDNB03].

To illustrate these issues, consider the class hierarchy given in Figure 1. The root of this hierarchy is the `CDevice` class that implements I/O on some file descriptor. It has two subclasses for reading and writing integers on the device (respectively). Defining a class that supports both reading and writing (`CIntRW`) requires reimplementing one or the other of the methods (denoted by the

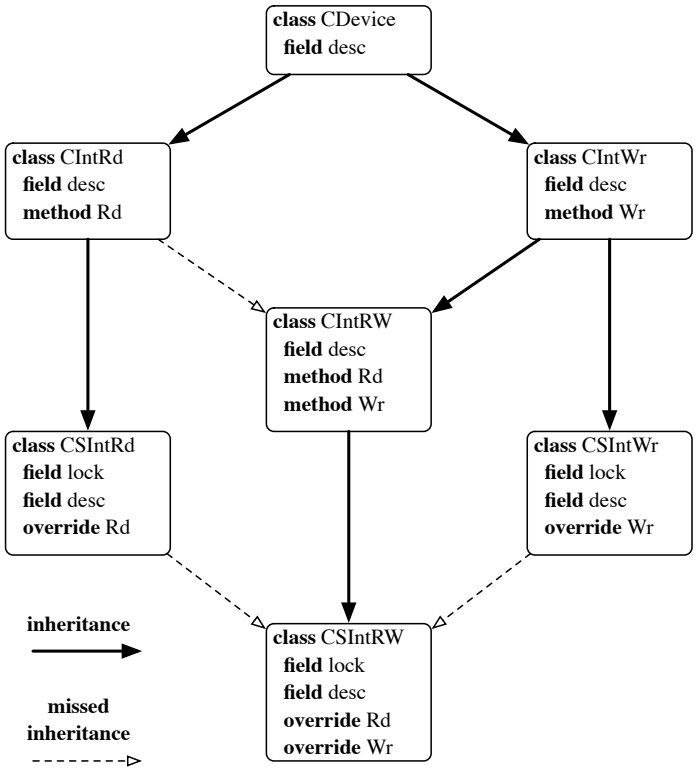

Figure 1: The readers and writers example

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 other subclasses (*e.g.*, a class for reading booleans). This class hierarchy is further extended with support for *synchronized* reading and writing by adding a lock. Single inheritance again forces us to reimplement methods.

Traits, however, allow us to reuse code without having to define methods too high. Figure 2 illustrates the trait version of this hierarchy. In this version of the code, we have four traits that are used to generate six classes.

3 A typed trait calculus

In this section, we present the syntax and semantics of our typed trait calculus.

3.1 Syntax

We give the collected syntax of our calculus in Figure 3. Before describing this syntax, we introduce some notation. 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 sets and naming

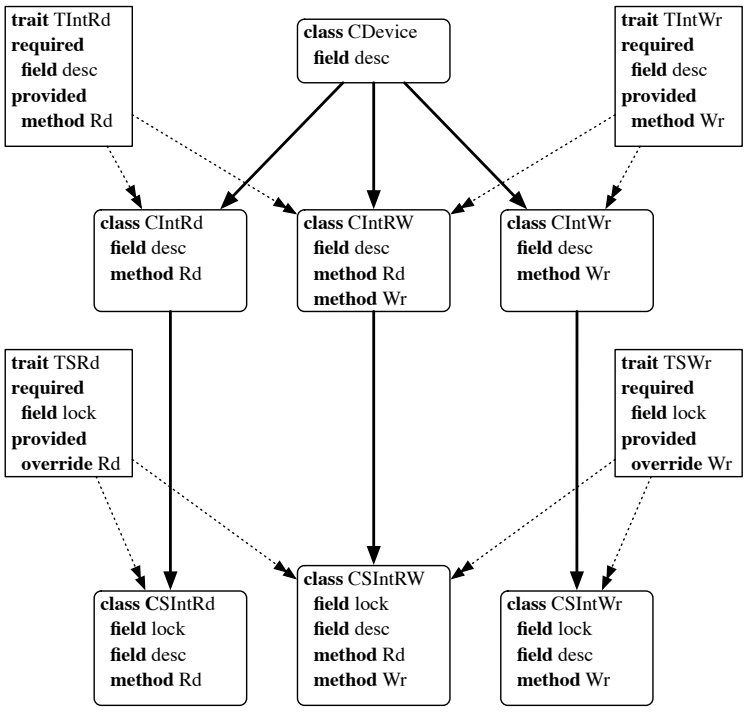

Figure 2: The readers and writers example using traits

conventions:

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 field and method names

In addition, we assume disjoint, countable sets of trait names TNAMES, class names CNAMES, expression names/variables VARIABLES, and type variables TYVARIABLES.

In our calculus, a program is a sequence of declarations followed by an expression. Each declaration binds a trait, class, or expression to a name. To enhance the reuseability of traits, we permit them to be implicitly parameterized by type variables. The declaration form for traits makes this dependence explicit by abstracting the free type variables.

The syntax of our trait forms is inspired by the formal model of Smalltalk traits developed by Schärli *et al.* [SDN+02]. A trait expression can be the name of a previously defined trait, instantiated with appropriate type arguments; the formation of a base trait from a method suite and auxiliary typing information; the symmetric (or disjoint) concatenation of two traits; the exclusion

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	::=	$t(\vec{r})$	polymorphic trait name: t $\in$ TNAMES
	$\mid$	$\langle\!\langle\, M;\, \langle\theta\rangle \,\rangle\!\rangle$	trait formation
	$\mid$	$T_1 + T_2$	symmetric concatenation
	$\mid$	$T \setminus m$	method exclusion
	$\mid$	$T[m' \mapsto m]$	method alias: bind new m' to old m
M	::=	$\langle {\mu_m}^{m \in \mathcal{M}} \rangle_{\mathsf{M}}$	method suite
μ	::=	$m\,(x : \tau_1) : \tau_2\{e\}$	method definition
C	::=	c	class name: c $\in$ CNAMES
	$\mid$	**nil**	empty class
	$\mid$	I **in** T **extends** C	inheritance (subclass formation)
I	::=	$\lambda(x : \tau).(\mathbf{super}\ e_1) \oplus e_2$	constructor
e	::=	x	expression name/variable: x $\in$ VARIABLES
	$\mid$	$\lambda(x : \tau).e$	function abstraction
	$\mid$	$e_1 e_2$	function application
	$\mid$	**new** $c\ e$	object instantiation
	$\mid$	**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
	$\mid$	$\langle f = e_f{}^{f \in \mathcal{F}} \rangle_{\mathsf{F}}$	field record definition
	$\mid$	$e_1 \oplus e_2$	field record concatentation
	$\mid$	$()$	unit value
θ	::=	$l : \tau_l{}^{l \in \mathcal{L}}$	row
τ	::=	α	type variable: $\alpha \in$ TYVARIABLES
	$\mid$	$\Lambda(\vec{\alpha}).\tau$	polymorphic type
	$\mid$	$\langle\!\langle\, \langle\theta\rangle;\, \mathcal{S};\, \mathcal{R} \,\rangle\!\rangle$	trait type
	$\mid$	$\{\!\{\, \theta \,\}\!\}$	class body type
	$\mid$	$\langle\theta\rangle$	object and method suite type
	$\mid$	$\langle f : \tau_f{}^{f \in \mathcal{F}} \rangle_{\mathsf{F}}$	field record type
	$\mid$	$\tau_1 \rightarrow \tau_2$	function type
	$\mid$	$Unit$	unit type

Figure 3: Trait calculus syntax

of a method from a trait; or the addition of a method alias to a trait.

We have a simple model of class definition that supports single inheritance, object initialization, and method definition via traits. Our classes do not support abstract methods (traits provide that capability), nor do they permit private, protected, or static members. 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 additional fields and a trait. Each subclass defines a constructor function, responsible for initializing all the fields of the class. To obtain initialized inherited fields, the constructor applies the superclass constructor function to a specified expression. It then concatenates the resulting fields with those defined directly in the subclass. 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 object initialization.

At the core of our calculus is a simple object-oriented language with objects and first-class functions. The language includes variables, function abstraction and application, object creation, self reference, super-method dispatch, method dispatch, field selection, field update, and unit. In addition, it includes operations for defining a collection of fields and concatenating two disjoint records of fields, which we use to model object initialization.

3.2 Type syntax

Our type system, the syntax of which appears in Figure 3, has four different expression types: object types, field record types, function types, and the Unit type. The type system also has types for traits and class bodies, although the corresponding terms are not first-class. To accomodate polymorphic traits, we add type variables and polymorphic types. Syntactic restrictions limit polymorphic types to trait names. The types for objects, classes, and traits are written in terms of *rows* (θ), which assign types to a collection of labels.

Trait types document the types of provided methods and specify the types of required methods and super-class methods.[2] Syntactically, trait types have the form $\langle\!|\, \langle\theta\rangle;\ \mathcal{S};\ \mathcal{R} \,|\!\rangle$, where row θ describes the types of all required, super-class, and provided methods and the types of all required fields. The set $\mathcal{S}$ contains the names of the super-methods that must be available from any class that can incorporate the trait, while the set $\mathcal{R}$ names the required methods and fields. All of the names in $\mathcal{S} \cup \mathcal{R}$ must be given types in θ.

We give trait declarations polymorphic types of the form $\Lambda(\vec{\alpha}).\tau$, where $\vec{\tau}$ is a shorthand for a possibly non-empty, comma separated list of types. This form binds the type variables $\vec{\alpha}$ in the trait type τ. When a trait or class refers to a named trait, it must supply appropriate type arguments to instantiate the polymorphic trait.

The type we give to a class is a function type, whose domain is the type of the class's constructor argument and whose range is a class body type, written $\{\!|\, \theta \,|\!\}$. In this type, row θ assigns types to the collection of fields and methods defined in the associated class.

[2] Note that the set of required methods and the set of super-class methods are independent, in that neither is necessarily a subset of the other. For example, a trait may define a method `m` but require any incorporating class to have previously defined a method of the same name, making `m` a super-class method but not required.

3.3 Example: Synchronized readers and writers

To illustrate our calculus, we show how to code the synchronized readers and writers example from the previous section. To improve the readability of the code, we introduce some standard syntactic sugar and some abbreviations. In particular, we use ';' for sequencing expressions and '`e1 before e2`' for first executing `e1`, then executing `e2`, and then returning the result of `e1`. We also use the following abbreviations:

$$\texttt{DescTy} \;=\; \langle \texttt{ReadInt} : \texttt{Unit} \to \texttt{Int},\, \texttt{WriteInt} : \texttt{Int} \to \texttt{Unit}, \ldots \rangle$$

$$\texttt{LockTy} \;=\; \langle \texttt{Acquire} : \texttt{Unit} \to \texttt{Unit},\, \texttt{Release} : \texttt{Unit} \to \texttt{Unit}, \ldots \rangle$$

Intuitively, an object with type `DescTy` models an I/O device that supports reading and writing for various primitive types; objects with type `LockTy` are semaphores. Finally, we assume that we have previously coded a class `CLock` that generates objects of type `LockTy` when instantiated with the unit value.

$$\begin{array}{lcl}
\texttt{CDevice} & = & \lambda(\texttt{x} : \texttt{Unit}).(\textbf{super}\ \texttt{()}) \oplus \langle \texttt{desc} = \ldots \rangle_{\texttt{F}}\ \textbf{in}\ \ldots\ \textbf{extends}\ \ldots \\[2ex]
\texttt{TIntRd} & = & ()\{\!|\, \langle \texttt{Rd}\,(\texttt{x} : \texttt{Unit}) : \texttt{Int}\{\textbf{self}.\texttt{desc}.\texttt{ReadInt}\ ()\}\rangle_{\texttt{M}};\ \langle \texttt{desc} : \texttt{DescTy}\rangle\, |\!\} \\[2ex]
\texttt{TIntWr} & = & ()\{\!|\, \langle \texttt{Wr}\,(\texttt{x} : \texttt{Int}) : \texttt{Unit}\{\textbf{self}.\texttt{desc}.\texttt{WriteInt}\ \texttt{x}\}\rangle_{\texttt{M}};\ \langle \texttt{desc} : \texttt{DescTy}\rangle\, |\!\} \\[2ex]
\texttt{CIntRd} & = & \lambda(\texttt{x} : \texttt{Unit}).(\textbf{super}\ \texttt{()}) \oplus \langle\rangle_{\texttt{F}}\ \textbf{in}\ \texttt{TIntRd}\ \textbf{extends}\ \texttt{CDevice} \\[2ex]
\texttt{CIntRW} & = & \lambda(\texttt{x} : \texttt{Unit}).(\textbf{super}\ \texttt{()}) \oplus \langle\rangle_{\texttt{F}}\ \textbf{in}\ \texttt{TIntRd + TIntWr}\ \textbf{extends}\ \texttt{CDevice} \\[2ex]
\texttt{CIntWr} & = & \lambda(\texttt{x} : \texttt{Unit}).(\textbf{super}\ \texttt{()}) \oplus \langle\rangle_{\texttt{F}}\ \textbf{in}\ \texttt{TIntWr}\ \textbf{extends}\ \texttt{CDevice} \\[2ex]
\texttt{TSRd} & = & (\alpha)\ \{\!|\ \langle \texttt{Rd}\,(\texttt{x} : \texttt{Unit}) : \alpha\ \{\ \textbf{self}.\texttt{lock}.\texttt{Acquire}\ (); \\
& & \qquad\qquad\qquad\qquad\quad \textbf{super}.\texttt{Rd}\ ()\ \texttt{before} \\
& & \qquad\qquad\qquad\qquad\quad\ \ \textbf{self}.\texttt{lock}.\texttt{Release}\ ()\ \ \}\rangle_{\texttt{M}}; \\
& & \qquad \langle \texttt{desc} : \texttt{DescTy}, \texttt{lock} : \texttt{LockTy}\rangle\ |\!\} \\[2ex]
\texttt{TSWr} & = & (\alpha)\ \{\!|\ \langle \texttt{Wr}\,(\texttt{x} : \alpha) : \texttt{Unit}\ \{\ \textbf{self}.\texttt{lock}.\texttt{Acquire}\ (); \\
& & \qquad\qquad\qquad\qquad\quad \textbf{super}.\texttt{Wr}\ \texttt{x}; \\
& & \qquad\qquad\qquad\qquad\quad \textbf{self}.\texttt{lock}.\texttt{Release}\ ()\ \ \}\rangle_{\texttt{M}}; \\
& & \qquad \langle \texttt{desc} : \texttt{DescTy}, \texttt{lock} : \texttt{LockTy}\rangle\ |\!\} \\[2ex]
\texttt{CSIntRd} & = & \lambda(\texttt{x} : \texttt{Unit}).(\textbf{super}\ \texttt{()}) \oplus \langle \texttt{lock} = \textbf{new}\ \texttt{CLock}\ ()\rangle_{\texttt{F}}\ \textbf{in} \\
& & \quad \texttt{TSRd(Int)}\ \textbf{extends}\ \texttt{CIntRd} \\[2ex]
\texttt{CSIntRW} & = & \lambda(\texttt{x} : \texttt{Unit}).(\textbf{super}\ \texttt{()}) \oplus \langle \texttt{lock} = \textbf{new}\ \texttt{CLock}\ ()\rangle_{\texttt{F}}\ \textbf{in} \\
& & \quad \texttt{TSRd(Int) + TSWr(Int)}\ \textbf{extends}\ \texttt{CIntRW} \\[2ex]
\texttt{CSIntWr} & = & \lambda(\texttt{x} : \texttt{Unit}).(\textbf{super}\ \texttt{()}) \oplus \langle \texttt{lock} = \textbf{new}\ \texttt{CLock}\ ()\rangle_{\texttt{F}}\ \textbf{in} \\
& & \quad \texttt{TSWr(Int)}\ \textbf{extends}\ \texttt{CIntWr}
\end{array}$$

Figure 4: Synchronous reader/writer example written in formal calculus.

With these assumptions, Figure 4 shows how to express the synchronized readers and writers from the previous section. The calculus is expressive enough to capture the full extent of reuse

in the example: the structure of this code is isomorphic to the picture in Figure 1. Note that the `TSReader` and `TSWriter` traits are polymorphic. Consequently, they can be reused to define synchronous versions of reader and writers for types other than `Int`. Figure 5 illustrates using `TSReader` to create a synchronized boolean reader.

$$\texttt{TBoolRd} \;=\; ()\{\!|\, \langle \texttt{Rd}\,(\texttt{x} : \texttt{Unit}) : \texttt{Bool}\{\mathbf{self}.\texttt{desc}.\texttt{ReadBool}\,()\}\rangle_{\texttt{M}};\, \langle \texttt{desc} : \texttt{DescTy}\rangle \,|\!\}$$

$$\texttt{CBoolRd} \;=\; \lambda(\texttt{x} : \texttt{Unit}).(\mathbf{super}\ ()) \oplus \langle\rangle_{\texttt{F}}\ \mathbf{in}\ \texttt{TBoolRd}\ \mathbf{extends}\ \texttt{CDevice}$$

$$\texttt{CSBoolRd} \;=\; \lambda(\texttt{x} : \texttt{Unit}).(\mathbf{super}\ ()) \oplus \langle \texttt{lock} = \mathbf{new}\ \texttt{CLock}\ ()\rangle_{\texttt{F}}\ \mathbf{in}\ \texttt{TSRd(Bool)}\ \mathbf{extends}\ \texttt{CBoolRd}$$

Figure 5: Boolean synchronous reader in formal calculus.

3.4 Dynamic semantics

We have developed a big-step operational semantics for our calculus. We write the evaluation rules in terms of environments E, which map identifers (trait names, class names, and expression variables) to values, and stores S, which map addresses to object values. Values include trait values, class values, and expression values: addresses, closures, records of field values, and the unit value. The run-time representation of an object is its address, created by executing the **new** expression. We find the object's associated method suite and fields by looking up the object's address in the store. We adopted a store-based semantics so that we could model imperative field update. In the remainder of this section, we describe the evaluation rules related to traits and classes, leaving the remaining, standard rules to Appendix B.

The judgment form $E,\ S \vdash P \longrightarrow ev \bullet E' \bullet S'$ says that evaluating a program P in an environment E and store S yields an expression value ev, an extended environment E', and a modified store S'. Similarly, the judgment form $E,\ S \vdash D \longrightarrow E' \bullet S'$ indicates that evaluating a declaration D will yield an extended environment and a possibly updated store. The instance of this judgment for trait declaration evaluation is as follows:[3]

$$\frac{E,\ S \vdash T \longrightarrow tv \bullet S \qquad t \notin \mathrm{dom}(E)}{E,\ S \vdash t = (\vec{\alpha})T \longrightarrow E \pm \{t \mapsto (\vec{\alpha})tv\} \bullet S} \tag{6}$$

This rule says that if we can reduce a trait T to a trait value tv, then we may augment the environment with a binding from the fresh trait name t to the abstracted trait value $(\vec{\alpha})tv$. Note that this rule does not change the store.[4]

3.4.1 Trait evaluation

Trait evaluation reduces trait expressions to trait values, which have the form $\{\!|\, Mv;\, \langle\theta\rangle \,|\!\}$, where Mv is a method suite value and θ lists the names and types of all the required and provided methods and fields. The judgment $E,\ S \vdash T \longrightarrow tv \bullet S$ indicates that trait T evaluates to trait value tv in environment E and store S. For syntactic consistency, this judgment form returns the store, although trait evaluation cannot modify it. We define this judgment form using the rules given below.

[3]The rule number refers to the occurrence of the rule in the appendices.

[4]The store can only be modified during expression evaluation.

A named trait applied to type arguments is evaluated by looking up the trait name in the environment and substituting the type arguments for the bound type variables in the associated value:

$$\frac{E(t) = (\vec{\alpha})tv}{E,\ S \vdash t(\vec{\tau}) \longrightarrow tv[\vec{\tau}/\vec{\alpha}] \bullet S} \tag{9}$$

We use the syntax $X[\vec{\tau}/\vec{\alpha}]$ to denote the simultaneous substitution of $\vec{\tau}$ for $\vec{\alpha}$ in X, where X is either a type τ or a trait value tv.

To convert a trait formation expression into a value, we need only evaluate the method suite. Method suite evaluation uses the environment E to create closures for each of the method bodies.

$$\frac{E,\ S \vdash M \longrightarrow Mv \bullet S}{E,\ S \vdash \{\!|\, M;\ \langle\theta\rangle \,|\!\} \longrightarrow \{\!|\, Mv;\ \langle\theta\rangle \,|\!\} \bullet S} \tag{10}$$

The rule for symmetric concatenation of two traits ($T_1 + T_2$) merges the disjoint method suites from the two traits. 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.

$$\frac{\begin{array}{c} E,\ S \vdash T_1 \longrightarrow \{\!|\, \langle {\mu v_m}^{m \in \mathcal{M}_1} \rangle_{\mathtt{M}};\ \langle {l : \tau_l}^{l \in \mathcal{R}_1} \rangle \,|\!\} \bullet S \\ E,\ S \vdash T_2 \longrightarrow \{\!|\, \langle {\mu v_m}^{m \in \mathcal{M}_2} \rangle_{\mathtt{M}};\ \langle {l : \tau_l}^{l \in \mathcal{R}_2} \rangle \,|\!\} \bullet S \\ \mathcal{M}_1 \pitchfork \mathcal{M}_2 \quad \mathcal{M}_3 = \mathcal{M}_1 \cup \mathcal{M}_2 \quad \mathcal{R}_3 = (\mathcal{R}_1 \cup \mathcal{R}_2) \setminus \mathcal{M}_3 \end{array}}{E,\ S \vdash T_1 + T_2 \longrightarrow \{\!|\, \langle {\mu v_m}^{m \in \mathcal{M}_3} \rangle_{\mathtt{M}};\ \langle {l : \tau_l}^{l \in \mathcal{R}_3} \rangle \,|\!\} \bullet S} \tag{9}$$

We use the notation $\mathcal{M}_1 \pitchfork \mathcal{M}_2$ to denote that the sets $\mathcal{M}_1$ and $\mathcal{M}_2$ are disjoint.

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.

$$\frac{\begin{array}{c} E,\ S \vdash T \longrightarrow \{\!|\, \langle {\mu v_m}^{m \in \mathcal{M}} \rangle_{\mathtt{M}};\ \langle {l : \tau_l}^{l \in \mathcal{R}} \rangle \,|\!\} \bullet S \\ m \in \mathcal{M} \qquad \mu v_m = (m : \tau_m = \lambda v) \end{array}}{E,\ S \vdash T \setminus m \longrightarrow \{\!|\, \langle {\mu v_m}^{m \in \mathcal{M} \setminus \{m\}} \rangle_{\mathtt{M}};\ \langle {l : \tau_l}^{l \in \mathcal{R} \cup \{m\}} \rangle \,|\!\} \bullet S} \tag{12}$$

We believe that a more refined (but more verbose!) type system could track such dependencies and remove the method entirely if no further references exist, but we have not pursued this avenue.

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.

$$\frac{\begin{array}{c} E,\ S \vdash T \longrightarrow \{\!|\, \langle {\mu v_m}^{m \in \mathcal{M}} \rangle_{\mathtt{M}};\ \langle {l : \tau_l}^{l \in \mathcal{R}} \rangle \,|\!\} \bullet S \\ m \in \mathcal{M} \qquad m' \notin \mathcal{M} \qquad \mu v_m = (m : \tau_m = \lambda v) \end{array}}{E,\ S \vdash T[m' \mapsto m] \longrightarrow \{\!|\, \langle {\mu v_m}^{m \in \mathcal{M}},\ (m' : \tau_m = \lambda v) \rangle_{\mathtt{M}};\ \langle {l : \tau_l}^{l \in \mathcal{R} \setminus \{m'\}} \rangle \,|\!\} \bullet S} \tag{13}$$

3.4.2 Class evaluation

Class evaluation reduces class expressions to class values, which are of the form $\{\!|\, \lambda v;\ Mv \,|\!\}$, where λv is a closure representing the constructor function for the class and Mv is its method suite. The class evaluation judgment has the form $E,\ S \vdash C \longrightarrow cv \bullet S$, indicating that class C evaluates to

class value cv in environment E and store S. Like the trait evaluation judgment form, this judgment form returns the store, although class evaluation cannot modify it.

To evaluate class names, we lookup the corresponding class value in the environment.

$$\frac{c \in \mathrm{dom}(E)}{E,\ S \vdash c \longrightarrow E(c) \bullet S} \tag{16}$$

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

$$\frac{}{E,\ S \vdash \mathbf{nil} \longrightarrow \{\!|\ [\epsilon;\ \lambda x.\langle\rangle_{\mathtt{F}}];\ \langle\rangle_{\mathtt{M}}\ |\!\} \bullet S} \tag{17}$$

Forming a new subclass from the combination of a syntactic constructor function, a trait expression, and a superclass is the heart of our system. Although method suites in traits may contain references to **super**, we compile such references away during class construction to simplify the representation of class values. This simplification is possible because invocations through **super** can be uniquely resolved at the point of class definition. The judgment form $Mv \vdash Mv_1 \Longrightarrow Mv_2$ specifies that the method suite Mv_1 can be rewritten with respect to the super-class method suite Mv to produce the **super**-free method suite Mv_2. The rules defining this judgment form appear in Appendix B. The following rule defines the inheritance form for classes:

$$\frac{\begin{array}{c} E,\ S \vdash T \longrightarrow \langle\!|\ \langle {\mu v'}_m{}^{m \in \mathcal{M}_T} \rangle_{\mathtt{M}};\ \langle l : \tau_l{}^{l \in \mathcal{R}_T} \rangle\ |\!\rangle \bullet S \\ E,\ S \vdash C \longrightarrow \{\!|\ ev_{super};\ \langle {\mu v}_m{}^{m \in \mathcal{M}_C} \rangle_{\mathtt{M}}\ |\!\} \bullet S \\ \langle {\mu v}_m{}^{m \in \mathcal{M}_c} \rangle_{\mathtt{M}} \vdash \langle {\mu v'}_m{}^{m \in \mathcal{M}_T} \rangle_{\mathtt{M}} \Longrightarrow \langle {\mu v''}_m{}^{m \in \mathcal{M}_T} \rangle_{\mathtt{M}} \\ ev_{con} = [E \pm \{super \mapsto ev_{super}\};\ \lambda x.(super\ e_{args}) \oplus e_F] \qquad super \notin \mathrm{dom}(E) \end{array}}{\begin{array}{c} E,\ S \vdash \lambda(x : \tau).(\mathbf{super}\ e_{args}) \oplus e_F\ \mathbf{in}\ T\ \mathbf{extends}\ C \longrightarrow \\ \{\!|\ ev_{con};\ \langle {\mu v}_m{}^{m \in \mathcal{M}_C \setminus \mathcal{M}_T},\ {\mu v''}_m{}^{m \in \mathcal{M}_T} \rangle_{\mathtt{M}}\ |\!\} \bullet S \end{array}} \tag{18}$$

The first two lines specify the evaluation of the nested trait and class expressions. The third line rewrites the trait's methods to remove references to **super** by inlining methods from C. The fourth line constructs the appropriate closure to represent the constructor function for the new class. The environment for this closure binds the fresh variable $super$ to the constructor function for class C; its body merges the fields obtained from C with those defined directly in the new class. We added the field concatenation operation to our expression calculus to support this usage.

3.5 Static semantics

In this section, we give an overview of the type system for our calculus, again focusing on the typing rules for traits and classes as the other parts of our calculus are routine. For reference, Appendix C contains the complete type system.

3.5.1 Contexts

All our typing judgments are written in terms of an ordered context Γ, which lists free type variables and 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
$\Gamma \vdash \tau$	well-formed type
$\Gamma \vdash \tau_1 <: \tau_2$	subtyping
$\Gamma \vdash T : \tau$	trait T has type τ
$\Gamma \vdash {}_{\tau_{\mathbf{super}};\tau_{\mathbf{self}}} M : \tau$	method suite M has type τ assuming $\mathbf{super} : \tau_{\mathbf{super}}$ and $\mathbf{self} : \tau_{\mathbf{self}}$
$\Gamma \vdash \mu : \tau$	method body μ has type τ
$\Gamma \vdash C : \tau$	class C has type τ
$\Gamma \vdash e : \tau$	expression e has type τ
$\Gamma \vdash t = (\vec{\alpha})T \Rightarrow \Gamma'$	well-formed trait declaration
$\Gamma \vdash c = C \Rightarrow \Gamma'$	well-formed class declaration, yielding new environment Γ'
$\Gamma \vdash x = e \Rightarrow \Gamma'$	well-formed expression declaration, yielding new environment Γ'
$\Gamma \vdash_P P : \tau$	well-typed program, yielding new environment Γ'

3.5.2 Trait typing

In this section, we describe the rules for typing trait expressions. The first rule types trait names applied to type arguments by instantiating the context's type for the trait with the supplied type arguments, assuming those types are well-formed:

$$\frac{\Gamma(t) = \Lambda(\vec{\alpha}).\tau \qquad \Gamma \vdash \vec{\tau} \qquad |\vec{\tau}| = |\vec{\alpha}|}{\Gamma \vdash t(\vec{\tau}) : \tau[\vec{\tau}/\vec{\alpha}]} \tag{60}$$

The second rule types trait formation:

$$\frac{\tau_{\mathbf{super}} = \langle l : \tau_l{}^{l \in \mathcal{S}} \rangle \qquad \tau_{\mathbf{self}} = \langle l : \tau_l{}^{l \in \mathcal{M} \cup \mathcal{R}} \rangle \qquad \Gamma \vdash {}_{\tau_{\mathbf{super}};\tau_{\mathbf{self}}} M : \langle m : \tau_m{}^{m \in \mathcal{M}} \rangle \qquad \mathcal{M} \Cap \mathcal{R}}{\Gamma \vdash \langlef\!\langle M;\ \langle l : \tau_l{}^{l \in \mathcal{R}} \rangle \rangle\!\rangle : \langlef\!\langle \langle l : \tau_l{}^{l \in \mathcal{M} \cup \mathcal{R}} \rangle;\ \mathcal{S};\ \mathcal{R} \rangle\!\rangle} \tag{61}$$

It assigns a type to M, the method suite for the trait, under assumptions about the types of $\mathbf{super}$ and $\mathbf{self}$. The type of $\mathbf{self}$, $\tau_{\mathbf{self}}$, contains each of the methods of the methd 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 a restriction of $\tau_{\mathbf{self}}$ to methods that must come from the superclass. The method-suite typing judgment ensures that $\tau_{\mathbf{self}}$ is a subtype of $\tau_{\mathbf{super}}$, a consequence of which is that both types are well-formed, as are the programmer-supplied types. The condition $\mathcal{M} \Cap \mathcal{R}$ guarantees that no method provided by 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

$$\frac{\begin{array}{c}\Gamma \vdash T_1 : \langlef\!\langle \langle l : \tau_l{}^{l \in \mathcal{L}_1} \rangle;\ \mathcal{S}_1;\ \mathcal{R}_1 \rangle\!\rangle \\ \Gamma \vdash T_2 : \langlef\!\langle \langle l : \tau_l{}^{l \in \mathcal{L}_2} \rangle;\ \mathcal{S}_2;\ \mathcal{R}_2 \rangle\!\rangle \\ \mathcal{M}_1 = \mathcal{L}_1 \setminus \mathcal{R}_1 \quad \mathcal{M}_2 = \mathcal{L}_2 \setminus \mathcal{R}_2 \quad \mathcal{M}_1 \Cap \mathcal{M}_2 \\ \mathcal{R}'_1 = \mathcal{R}_1 \setminus \mathcal{M}_2 \quad \mathcal{R}'_2 = \mathcal{R}_2 \setminus \mathcal{M}_1\end{array}}{\Gamma \vdash T_1 + T_2 : \langlef\!\langle \langle l : \tau_l{}^{l \in \mathcal{L}_1 \cup \mathcal{L}_2} \rangle;\ \mathcal{S}_1 \cup \mathcal{S}_2;\ \mathcal{R}'_1 \cup \mathcal{R}'_2 \rangle\!\rangle} \tag{62}$$

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.*, $m \in \mathcal{L} \setminus \mathcal{R}$).

$$\frac{\Gamma \vdash T \ : \ \{\!|\, \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle;\ \mathcal{S};\ \mathcal{R} \,|\!\}\quad m \in \mathcal{L} \setminus \mathcal{R}}{\Gamma \vdash T \setminus m \ : \ \{\!|\, \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle;\ \mathcal{S};\ \mathcal{R} \cup \{m\} \,|\!\}} \tag{63}$$

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

$$\frac{\begin{array}{c}\Gamma \vdash T \ : \ \{\!|\, \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle;\ \mathcal{S};\ \mathcal{R} \,|\!\} \\ m \in \mathcal{L} \setminus \mathcal{R} \qquad m' \notin \mathcal{L} \setminus \mathcal{R} \qquad \tau_{m'} = \tau_m\end{array}}{\Gamma \vdash T[m' \mapsto m] \ : \ \{\!|\, \langle l : \tau_l{}^{l \in \mathcal{L}},\ m' : \tau_{m'} \rangle;\ \mathcal{S};\ \mathcal{R} \setminus \{m'\} \,|\!\}} \tag{64}$$

The requirement that $\tau_{m'} = \tau_m$ is subtle: if m' is a required method in T, (*i.e.* $m' \in \mathcal{R}$), then the condition ensures that the type assumed for m' in T matches the type of method m. If m' is not required, (*i.e.*, $m' \notin \mathcal{R}$), then the condition defines $\tau_{m'}$ to be τ_m. Since we have added a binding for m', we remove m' from the set of required methods in the resulting trait.

3.5.3 Class typing

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

$$\frac{\begin{array}{c}\Gamma,\ x : \tau \vdash e_{super} \ : \ \tau_{super} \qquad \Gamma,\ x : \tau \vdash e_F \ : \ \langle f : \tau_f{}^{f \in \mathcal{F}} \rangle_{\mathsf{F}} \\ \Gamma \vdash T \ : \ \{\!|\, \langle l : \tau_l{}^{l \in \mathcal{L}_T} \rangle;\ \mathcal{S}_T;\ \mathcal{R}_T \,|\!\} \qquad \Gamma \vdash C \ : \ \tau_{super} \to \{\!|\, l : \tau_l{}^{l \in \mathcal{L}_c} \,|\!\} \\ \mathcal{S}_T \subseteq \mathcal{L}_c \qquad \mathcal{R}_T \subseteq (\mathcal{L}_c \cup \mathcal{F}) \qquad \mathcal{F} \pitchfork \mathcal{L}_c \qquad \mathcal{L} = \mathcal{F} \cup \mathcal{L}_T \cup \mathcal{L}_c\end{array}}{\Gamma \vdash \lambda(x : \tau).(\mathbf{super}\ e_{super}) \oplus e_F \ \mathbf{in}\ T\ \mathbf{extends}\ C \ : \ \tau \to \{\!|\, l : \tau_l{}^{l \in \mathcal{L}} \,|\!\}} \tag{69}$$

This rule infers types for the argument to the superclass constructor function, the record of fields defined in this class, the trait T to be incorporated, and the superclass C. It ensures that the type of the superclass argument matches the domain of the superclass constructor function. 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 C or $\mathcal{F}$ provides all the methods and fields required by T. 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 type τ_l, then the supplier of l (either $\mathcal{F}$ or C) must give the syntactically identical type to l.

3.6 Trait override

Although override is one of the primitive operations of Smalltalk traits, we did not include it as a primitive in our calculus because it is derivable as a combination of method removal and symmetric concatenation operations. Using the syntax $T_1 \triangleright T_2$ to denote overriding the trait T_2 with T_1, we define the operation as follows:

$$T_1 \triangleright T_2 \quad = \quad T_1 + ((T_2 \setminus m_1) \ldots \setminus m_n)$$

where $m_1, \ldots, m_n = \{m_i \,|\, m_i \in Methods(T_1)\}$ and $Methods(T)$ denotes the set of method names with definitions in trait T.

The derived evaluation rule for override is:

$$\frac{\begin{array}{c} E,\ S \vdash T_1 \longrightarrow \langle\!|\, \langle \mu v_m{}^{m \in \mathcal{M}_1} \rangle_{\mathtt{M}};\ \langle l : \tau_l{}^{l \in \mathcal{R}_1} \rangle \,|\!\rangle \bullet S \\ E,\ S \vdash T_2 \longrightarrow \langle\!|\, \langle \mu v'_m{}^{m \in \mathcal{M}_2} \rangle_{\mathtt{M}};\ \langle l : \tau_l{}^{l \in \mathcal{R}_2} \rangle \,|\!\rangle \bullet S \\ \mathcal{M}_3 = \mathcal{M}_1 \cup \mathcal{M}_2 \quad \mathcal{R}_3 = (\mathcal{R}_1 \cup \mathcal{R}_2) \setminus \mathcal{M}_3 \end{array}}{E,\ S \vdash T_1 \triangleright T_2 \longrightarrow \langle\!|\, \mu v_m{}^{m \in \mathcal{M}_1},\ \mu v'_m{}^{m \in \mathcal{M}_2 \setminus \mathcal{M}_1};\ \langle l : \tau_l{}^{l \in \mathcal{R}_3} \rangle \,|\!\rangle \bullet S} \tag{1}$$

Note that unlike the rule for trait sum, this rule does not require that the method suites for the two traits be disjoint.

The derived typing rule for override is:

$$\frac{\begin{array}{c} \Gamma \vdash T_1 \ :\ \langle\!|\, \langle l : \tau_l{}^{l \in \mathcal{L}_1} \rangle;\ \mathcal{S}_1;\ \mathcal{R}_1 \,|\!\rangle \\ \Gamma \vdash T_2 \ :\ \langle\!|\, \langle l : \tau_l{}^{l \in \mathcal{L}_2} \rangle;\ \mathcal{S}_2;\ \mathcal{R}_2 \,|\!\rangle \\ \mathcal{M}_1 = \mathcal{L}_1 \setminus \mathcal{R}_1 \quad \mathcal{M}_2 = \mathcal{L}_2 \setminus (\mathcal{R}_2 \cup \mathcal{M}_1) \\ \mathcal{R}'_1 = \mathcal{R}_1 \setminus \mathcal{M}_2 \quad \mathcal{R}'_2 = \mathcal{R}_2 \setminus \mathcal{M}_1 \end{array}}{\Gamma \vdash T_1 \triangleright T_2 \ :\ \langle\!|\, \langle l : \tau_l{}^{l \in \mathcal{L}_1 \cup \mathcal{L}_2} \rangle;\ \mathcal{S}_1 \cup \mathcal{S}_2;\ \mathcal{R}'_1 \cup \mathcal{R}'_2 \,|\!\rangle} \tag{2}$$

4 Type Soundness

To show that the operational semantics is consistent with the type system, we must give types to values, including object addresses. We cannot infer types for addresses by recursively examining their subcomponents because stores may contain cycles. To solve this problem, we introduce the notion of a store typing Σ, which maps object addresses to object types. This technique allows us to type values independently of a particular store. This approach is reasonable because type-safe computations always store results of different types in different locations. Judgments for typing run-time values are written in terms of store typings:

$\Sigma \vdash ok$	well-formed store type
$\Sigma \vdash E : \Gamma$	environment E "has type" Γ
$\vdash S : \Sigma$	store S "has type" Σ
$\Sigma \vdash_{(\vec{\alpha})} tv : \tau$	well-typed trait value with free type variables $\vec{\alpha}$
$\Sigma \vdash_{(\vec{\alpha});\tau_{\mathbf{super}};\tau_{\mathbf{self}}} Mv : \tau'$	well-typed method suite value
$\Sigma \vdash_{(\vec{\alpha});\tau_{\mathbf{super}};\tau_{\mathbf{self}}} \mu v : \tau'$	well-typed method value
$\Sigma \vdash cv : \tau$	well-typed class value
$\Sigma \vdash ev : \tau$	well-typed expression value
$\Sigma \vdash ov : \tau$	well-typed object value

Intuitively, all object addresses are typechecked "globally," with respect to the whole store:

$$\frac{\begin{array}{c} \Sigma \vdash ok \qquad \mathrm{dom}(\Sigma) = \mathrm{dom}(S) \\ \forall a \in \mathrm{dom}(S)\ \Sigma \vdash S(a) : \Sigma(a) \end{array}}{\vdash S : \Sigma} \tag{3}$$

The other rules related to typing run-time forms are largely straightforward. For completeness, they appear in Appendix D.

We show soundness by first proving a subject reduction theorem, as is standard. To state subject reduction, we must define some auxiliary terms first.

Definition 4.1 (Context extension) *Context Γ' extends Γ, which we write $\Gamma' \preceq \Gamma$, if $\Gamma' \vdash ok$ and for all $id \in \mathrm{dom}(\Gamma)$, $\Gamma' \vdash \Gamma'(id) <: \Gamma(id)$ and for all $\alpha \in \mathrm{dom}(\Gamma)$, $\alpha \in \mathrm{dom}(\Gamma')$.*

Definition 4.2 (Store typing extension) *Store typing Σ' extends Σ, which we write $\Sigma' \preceq \Sigma$, if $\Sigma' \vdash ok$ and for all $a \in \mathrm{dom}(\Sigma)$, $\Sigma'(a) = \Sigma(a)$.*

Theorem 4.1 (Subject reduction) *If $\Gamma \vdash_P P : \tau$ and $E,\ S \vdash P \longrightarrow ev \bullet E' \bullet S'$ and $\Sigma \vdash E : \Gamma$ and $\vdash S : \Sigma$, then there exist context $\Gamma' \preceq \Gamma$ and store typing $\Sigma' \preceq \Sigma$ such that $\Sigma' \vdash E' : \Gamma'$ and $\vdash S' : \Sigma'$ and $\Sigma' \vdash ev : \tau'$ and $\Gamma' \vdash \tau' <: \tau$.*

The proof is by induction on the structure of P. It follows from a subsidiary lemma that expression evaluation preserves types. □

To prove type soundness, we need a way to characterize whether a program terminates in our semantics. In small-step semantics, non-termination corresponds to an infinite sequence of reduction steps. In a big-step semantics, however, non-termination corresponds to a derivation tree of infinite height.

Definition 4.3 (Program evaluation height) *We define the evaluation height of a program expression P in environment E and store S to be $h^P_{E,\,S}(e)$, where $h^P_{E,\,S}(e)$ is defined in Appendix E.*

Theorem 4.2 (Soundness of program evaluation) *If $\Gamma \vdash_P P : \tau$ and $\Sigma \vdash E : \Gamma$ and $\vdash S : \Sigma$ and there exists an n such that $h^P_{E,\,S}(P) = n$, then there exist a store typing $\Sigma' \preceq \Sigma$, a store S', a context $\Gamma' \preceq \Gamma$, an environment E', a type τ', and an expression value ev such that $\Sigma' \vdash E' : \Gamma'$ and $\vdash S' : \Sigma'$ and $E,\ S \vdash P \longrightarrow ev \bullet E' \bullet S'$ and $\Sigma' \vdash ev : \tau'$ and $\epsilon \vdash \tau' <: \tau$.*

The proof is by induction n. □

Definition 4.4 (Divergence) *We say a program P* diverges *if there is no n such that $h^P_{\epsilon,\,\epsilon}(P) = n$.*

Corollary 4.1 (Type soundness) *If $\epsilon \vdash_P P : \tau$ 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 $\epsilon,\ \epsilon \vdash P \longrightarrow ev \bullet E \bullet S$ and $\Sigma \vdash ev : \tau'$ and $\epsilon \vdash \tau' <: \tau$.*

In English, if closed program P has type τ, then either P diverges or evaluates to an expression value whose type improves upon the statically determined type τ.

5 Related work

Our first attempt formalize traits in a typed setting is reported in a workshop paper [FR04]. This paper builds on this previous work in several significant ways. Most importantly, the caclulus in described in this paper supports polymorphic traits, which are crucial to support examples such as the synchronized readers given in Section 2. The calculus in this paper also has a more realistic object model, with stateful objects and object initialization. We have also fixed a number of minor warts in the previous design.

The starting point for our calculus was the formal model of Smalltalk traits described by Schärli *et al.* [SDN+02], but our work differs in many significant ways. The two semantics have a very different style and scope: we have taken an operational approach to specifying the semantics of a minimal, but complete, calculus of traits, whereas they use a series of abstract mathematical objects (*e.g.*, finite maps) to define the semantics of method dispatch, but do not address the semantics of a complete language. Most importantly, we address the issue of incorporating traits into a statically-typed class-based language. The other differences are mainly syntactic. To keep our calculus small and regular, we localized method definitions to traits, which in turn required allowing methods in traits to refer to fields, and we defined override as a derived form. Although minimal, our calculus subsumes the Smalltalk trait model (*i.e.*, any term in their model can be translated to a term in our model if one ignores the issue of typing).

There is similarity between our calculus of 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.

While the purpose of this paper is developing the foundations of typed traits, we briefly survey other work related to the design of traits as a language feature.

There are strong similarities between traits and *mixins* [BC90, FKF98, OAC+03], which are another mechanism designed to give many of the benefits of multiple inheritance in single-inheritance languages without 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]. 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.

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 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.

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 believe it could 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 described a statically-typed core calculus of traits, demonstrated its expressiveness via an example, and shown a type soundness result. A number of interesting questions about typed traits remain.

We briefly outline some of these questions in the remainder of this section.

In the interests of simplicity, we purposefully omitted a number of common features from our calculus, including depth subtyping and privacy controls. We do not believe that the introduction of depth subtyping will cause significant problems for type soundness. In previous work unrelated to traits, we supported 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.

Our type system maintains information about required labels at the trait level. Although the type system would be more verbose, we could also track per-method information about required labels using techniques similar to those developed in work typing extensible objects [BL95]. This more precise information would allow us to drop non-referenced labels from the required set in typing method exclusion, achieving more flexibility.[5]

Another, more speculative direction, is 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.

[BL95] Bono, V. and L. Liquori. A subtyping for the Fisher-Honsell-Mitchell lambda calculus of objects. In *CSL'94*, vol. 933 of *LNCS*, New York, NY, 1995. Springer-Verlag, pp. 16–30.

[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. (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.

[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.

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

[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`.

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

[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 Shorthands

We use a collection of shorthand notations for sequences: $\vec{\tau}$ is a possibly non-empty, comma separated list of types; $X[\vec{\tau}/\vec{\alpha}]$ denotes the simultaneous substitution of $\vec{\tau}$ for $\vec{\alpha}$ in X, where X is either a type τ or a trait value tv; $|\vec{\tau}|$ denotes the length of the sequence $\vec{\tau}$; and $\Gamma \vdash \vec{\tau}$ denotes the sequence of judgements $\Gamma \vdash \tau_1, \ldots, \Gamma \vdash \tau_n$, where $\vec{\tau} = \tau_1, \ldots, \tau_n$.

We use the notation $\langle f = ev_f{}^{f \in \mathcal{F}} \rangle_{\mathsf{F}}(f') = ev_{f'}$ to denote selecting the f' field from the record of fields $\langle f = ev_f{}^{f \in \mathcal{F}} \rangle_{\mathsf{F}}$. Similarly, we treat method suites as finite maps from their index set to their method bodies, *i.e.*, $\langle (m : \tau_m = \lambda v_m)^{m \in \mathcal{M}} \rangle_{\mathtt{M}}(m') = \lambda v_{m'}$ as long as $m' \in \mathcal{M}$. Finally, we use the function $Ty((m : \tau_m = \lambda v_m), m') = \tau_{m'}$ as long as $m' \in \mathcal{M}$ to extract the type associated with a method in a method suite.

B Evaluation

For purposes of evaluation, we treat **self** as an expression variable, *i.e.*, $\mathbf{self} \in$ VARIABLES.

B.1 Additional syntax to support evaluation

E	$::=$	$\emptyset \mid E \pm \{t \mapsto (\vec{\alpha})tv\} \mid E \pm \{c \mapsto cv\} \mid E \pm \{x \mapsto ev\}$	environment
S	$::=$	$\emptyset \mid S \pm \{a \mapsto ov\}$	store
tv	$::=$	$\langle\!\langle\, Mv;\, \langle\theta\rangle \,\rangle\!\rangle$	trait value
Mv	$::=$	$\langle \mu v_m{}^{m \in \mathcal{M}} \rangle_{\mathtt{M}}$	method suite value
μv	$::=$	$(m : \tau = \lambda v)$	method definition value
cv	$::=$	$\{\!\{\, \lambda v;\ Mv \,\}\!\}$	class value
ev	$::=$	a	object address
	$\mid$	λv	function value
	$\mid$	fv	fields record value
	$\mid$	$()$	unit value
λv	$::=$	$[E;\ \lambda x.e]$	function value
fv	$::=$	$\langle f = ev_f{}^{f \in \mathcal{F}} \rangle_{\mathsf{F}}$	field record value
ov	$::=$	$\langle fv;\ Mv \rangle$	object value

B.2 Evaluation judgement forms

$E,\ S \vdash P \longrightarrow ev \bullet E' \bullet S'$	program evaluation
$E,\ S \vdash D \longrightarrow E' \bullet S'$	declaration evaluation
$E,\ S \vdash T \longrightarrow tv \bullet S$	trait evaluation
$E,\ S \vdash M \longrightarrow Mv \bullet S$	method suite "compilation"
$E,\ S \vdash \mu \longrightarrow \mu v \bullet S$	method "compilation"
$E,\ S \vdash C \longrightarrow cv \bullet S$	class evaluation
$E,\ S \vdash e \longrightarrow ev \bullet S'$	expression evaluation
$Mv \vdash Mv_1 \Longrightarrow Mv_2$	Method suite:super-inliing
$Mv \vdash \mu v_1 \Longrightarrow \mu v_2$	Method body:super-inling
$Mv,\ E \vdash e_1 \Longrightarrow e_2 \bullet E'$	Expression:super-inlining

B.3 Evaluation rules

$$\frac{E,\ S \vdash D \longrightarrow E' \bullet S' \qquad E',\ S' \vdash P \longrightarrow ev \bullet E'' \bullet S''}{E,\ S \vdash D; P \longrightarrow ev \bullet E'' \bullet S''} \quad (4) \qquad \frac{E,\ S \vdash e \longrightarrow ev \bullet S'}{E,\ S \vdash e \longrightarrow ev \bullet E \bullet S'} \quad (5)$$

$$\frac{E,\ S \vdash T \longrightarrow tv \bullet S \qquad t \notin \mathrm{dom}(E)}{E,\ S \vdash t = (\vec{\alpha})T \longrightarrow E \pm \{t \mapsto (\vec{\alpha})tv\} \bullet S} \quad (6)$$

$$\frac{E,\ S \vdash C \longrightarrow cv \bullet S \qquad c \notin \mathrm{dom}(E)}{E,\ S \vdash c = C \longrightarrow E \pm \{c \mapsto cv\} \bullet S} \quad (7) \qquad \frac{E,\ S \vdash e \longrightarrow ev \bullet S' \qquad x \notin \mathrm{dom}(E)}{E,\ S \vdash x = e \longrightarrow E \pm \{x \mapsto ev\} \bullet S'} \quad (8)$$

$$\frac{E(t) = (\vec{\alpha})tv}{E,\ S \vdash t(\vec{\tau}) \longrightarrow tv[\vec{\tau}/\vec{\alpha}] \bullet S} \quad (9) \qquad \frac{E,\ S \vdash M \longrightarrow Mv \bullet S}{E,\ S \vdash \langle\!|\ M;\ \langle\theta\rangle\ |\!\rangle \longrightarrow \langle\!|\ Mv;\ \langle\theta\rangle\ |\!\rangle \bullet S} \quad (10)$$

$$\frac{\begin{array}{c}E,\ S \vdash T_1 \longrightarrow \langle\!|\ \langle {\mu v_m}^{m \in \mathcal{M}_1}\rangle_{\mathtt{M}};\ \langle {l : \tau_l}^{l \in \mathcal{R}_1}\rangle\ |\!\rangle \bullet S \\ E,\ S \vdash T_2 \longrightarrow \langle\!|\ \langle {\mu v_m}^{m \in \mathcal{M}_2}\rangle_{\mathtt{M}};\ \langle {l : \tau_l}^{l \in \mathcal{R}_2}\rangle\ |\!\rangle \bullet S \\ \mathcal{M}_1 \pitchfork \mathcal{M}_2 \quad \mathcal{M}_3 = \mathcal{M}_1 \cup \mathcal{M}_2 \quad \mathcal{R}_3 = (\mathcal{R}_1 \cup \mathcal{R}_2) \setminus \mathcal{M}_3\end{array}}{E,\ S \vdash T_1 + T_2 \longrightarrow \langle\!|\ \langle {\mu v_m}^{m \in \mathcal{M}_3}\rangle_{\mathtt{M}};\ \langle {l : \tau_l}^{l \in \mathcal{R}_3}\rangle\ |\!\rangle \bullet S} \quad (11)$$

$$\frac{\begin{array}{c}E,\ S \vdash T \longrightarrow \langle\!|\ \langle {\mu v_m}^{m \in \mathcal{M}}\rangle_{\mathtt{M}};\ \langle {l : \tau_l}^{l \in \mathcal{R}}\rangle\ |\!\rangle \bullet S \\ m \in \mathcal{M} \qquad \mu v_m = (m : \tau_m = \lambda v)\end{array}}{E,\ S \vdash T \setminus m \longrightarrow \langle\!|\ \langle {\mu v_m}^{m \in \mathcal{M} \setminus \{m\}}\rangle_{\mathtt{M}};\ \langle {l : \tau_l}^{l \in \mathcal{R} \cup \{m\}}\rangle\ |\!\rangle \bullet S} \quad (12)$$

$$\frac{\begin{array}{c}E,\ S \vdash T \longrightarrow \langle\!|\ \langle {\mu v_m}^{m \in \mathcal{M}}\rangle_{\mathtt{M}};\ \langle {l : \tau_l}^{l \in \mathcal{R}}\rangle\ |\!\rangle \bullet S \\ m \in \mathcal{M} \qquad m' \notin \mathcal{M} \qquad \mu v_m = (m : \tau_m = \lambda v)\end{array}}{E,\ S \vdash T[m' \mapsto m] \longrightarrow \langle\!|\ \langle {\mu v_m}^{m \in \mathcal{M}},\ (m' : \tau_m = \lambda v)\rangle_{\mathtt{M}};\ \langle {l : \tau_l}^{l \in \mathcal{R} \setminus \{m'\}}\rangle\ |\!\rangle \bullet S} \quad (13)$$

$$\frac{}{E,\ S \vdash m\,(x : \tau_1) : \tau_2\{e\} \longrightarrow (m : \tau_1 \to \tau_2 = [E;\ \lambda(x : \tau_1).e]) \bullet S} \quad (14)$$

$$\frac{E,\ S \vdash \mu_m \longrightarrow \mu v_m \bullet S \quad \text{forall } m \in \mathcal{M}}{E,\ S \vdash \langle {\mu_m}^{m \in \mathcal{M}}\rangle_{\mathtt{M}} \longrightarrow \langle {\mu v_m}^{m \in \mathcal{M}}\rangle_{\mathtt{M}} \bullet S} \quad (15)$$

$$\frac{c \in \mathrm{dom}(E)}{E,\ S \vdash c \longrightarrow E(c) \bullet S} \quad (16) \qquad \frac{}{E,\ S \vdash \mathbf{nil} \longrightarrow \{\!|\ [\epsilon;\ \lambda x.\langle\rangle_{\mathtt{F}}];\ \langle\rangle_{\mathtt{M}}\ |\!\} \bullet S} \quad (17)$$

$$\frac{\begin{array}{c} E,\ S \vdash T \longrightarrow \{\!|\ \langle \mu v'_m{}^{m \in \mathcal{M}_T} \rangle_{\mathtt{M}};\ \langle l : \tau_l{}^{l \in \mathcal{R}_T} \rangle\ |\!\} \bullet S \\ E,\ S \vdash C \longrightarrow \{\!|\ ev_{super};\ \langle \mu v_m{}^{m \in \mathcal{M}_C} \rangle_{\mathtt{M}}\ |\!\} \bullet S \\ \langle \mu v_m{}^{m \in \mathcal{M}_c} \rangle_{\mathtt{M}} \vdash \langle \mu v'_m{}^{m \in \mathcal{M}_T} \rangle_{\mathtt{M}} \Longrightarrow \langle \mu v''_m{}^{m \in \mathcal{M}_T} \rangle_{\mathtt{M}} \\ ev_{con} = [E \pm \{super \mapsto ev_{super}\};\ \lambda x.(super\ e_{args}) \oplus e_F] \qquad super \notin \mathrm{dom}(E) \end{array}}{\begin{array}{c} E,\ S \vdash \lambda(x:\tau).(\mathbf{super}\ e_{args}) \oplus e_F\ \mathbf{in}\ T\ \mathbf{extends}\ C \longrightarrow \\ \{\!|\ ev_{con};\ \langle \mu v_m{}^{m \in \mathcal{M}_C \setminus \mathcal{M}_T},\ \mu v''_m{}^{m \in \mathcal{M}_T} \rangle_{\mathtt{M}}\ |\!\} \bullet S \end{array}} \quad (18)$$

$$\frac{}{E,\ S \vdash x \longrightarrow E(x) \bullet S} \quad (19) \qquad \frac{}{E,\ S \vdash \lambda(x:\tau).e \longrightarrow [E;\ \lambda x.e] \bullet S} \quad (20)$$

$$\frac{\begin{array}{l} E,\ S \vdash e_1 \longrightarrow [E_1;\ \lambda x.e] \bullet S_1 \\ E,\ S_1 \vdash e_2 \longrightarrow ev_2 \bullet S_2 \\ E_1 \pm \{x \mapsto ev_2\},\ S_2 \vdash e \longrightarrow ev \bullet S_3 \end{array}}{E,\ S \vdash e_1\, e_2 \longrightarrow ev \bullet S_3} \quad (21)$$

$$\frac{\begin{array}{l} E,\ S \vdash c \longrightarrow \{\!|\ [E_F;\ \lambda x.e_F];\ Mv\ |\!\} \bullet S \\ E,\ S \vdash e \longrightarrow ev \bullet S_1 \\ E_F \pm \{x \mapsto ev\},\ S_1 \vdash e_F \longrightarrow fv \bullet S_2 \\ a \notin dom(S_2) \end{array}}{E,\ S \vdash \mathbf{new}\ c\ e \longrightarrow a \bullet S_2 \pm \{a \mapsto \langle fv;\ Mv \rangle\}} \quad (22)$$

$$\frac{}{E,\ S \vdash \mathbf{self} \longrightarrow E(\mathbf{self}) \bullet S} \quad (23)$$

$$\frac{E,\ S \vdash e \longrightarrow a \bullet S_1 \quad S_1(a) = \langle fv;\ Mv \rangle \quad Mv(m) = [E_m;\ \lambda x.e_m]}{E,\ S \vdash e.m \longrightarrow [E_m \pm \{\mathbf{self} \mapsto a\};\ \lambda x.e_m] \bullet S_1} \quad (24)$$

$$\frac{E,\ S \vdash e \longrightarrow a \bullet S_1 \quad S_1(a) = \langle fv;\ Mv \rangle \quad fv(f) = ev_f}{E,\ S \vdash e.f \longrightarrow ev_f \bullet S_1} \quad (25)$$

$$\frac{E,\ S \vdash e_1 \longrightarrow a \bullet S_1 \quad E,\ S_1 \vdash e_2 \longrightarrow ev_2 \bullet S_2 \quad S_2(a) = \langle \langle f = ev_f{}^{f \in \mathcal{F}} \rangle_{\mathtt{F}};\ Mv \rangle}{E,\ S \vdash e_1.f' := e_2 \longrightarrow () \bullet S_2 \pm \{a \mapsto \langle \langle f = ev_f{}^{f \in \mathcal{F} \setminus \{f'\}},\ f' = ev_2 \rangle_{\mathtt{F}};\ Mv \rangle\}} \quad (26)$$

$$\frac{}{E,\ S \vdash \langle\rangle_{\mathtt{F}} \longrightarrow \langle\rangle_{\mathtt{F}} \bullet S} \quad (27)$$

$$\frac{E,\ S \vdash e_1 \longrightarrow ev_1 \bullet S_1 \quad \cdots \quad E,\ S_{n-1} \vdash e_n \longrightarrow ev_n \bullet S_n}{E,\ S \vdash \langle f_1 = e_1, \ldots, f_n = e_n \rangle_{\mathtt{F}} \longrightarrow \langle f_1 = ev_1, \ldots, f_n = ev_n \rangle_{\mathtt{F}} \bullet S_n} \quad (28)$$

$$\frac{E,\ S \vdash e_1 \longrightarrow \langle f = ev_f{}^{f \in \mathcal{F}_1} \rangle_{\mathtt{F}} \bullet S_1 \quad E,\ S_1 \vdash e_2 \longrightarrow \langle f = ev_f{}^{f \in \mathcal{F}_2} \rangle_{\mathtt{F}} \bullet S_2}{E,\ S \vdash e_1 \oplus e_2 \longrightarrow \langle f = ev_f{}^{f \in \mathcal{F}_1 \cup \mathcal{F}_2} \rangle_{\mathtt{F}} \bullet S_2} \quad (29)$$

$$\frac{}{E,\ S \vdash () \longrightarrow () \bullet S} \quad (30)$$

B.4 Super in-lining

In this section, we present the details a method suite rewriting, which is used to resolve super-method dispatch statically. To eliminate references to **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. The rewriting works by replacing each super class reference $\mathbf{super}.m$ with a fresh variable x_m and augmenting the containing method's closure with a mapping from x_m to the closure associated with the super class method.

$$\frac{Mv \vdash \mu v_m \Longrightarrow \mu v'_m \quad \text{forall } m \in \mathcal{M}}{Mv \vdash \langle {\mu v_m}^{m \in \mathcal{M}} \rangle_{\mathtt{M}} \Longrightarrow \langle {\mu v'_m}^{m \in \mathcal{M}} \rangle_{\mathtt{M}}} \tag{31}$$

$$\frac{Mv,\ E \vdash e \Longrightarrow e' \bullet E'}{Mv \vdash (m : \tau = [E;\ \lambda x.e]) \Longrightarrow (m : \tau = [E';\ \lambda x.e'])} \tag{32}$$

$$\frac{}{Mv,\ E \vdash x \Longrightarrow x \bullet E} \tag{33} \qquad \frac{Mv,\ E \vdash e \Longrightarrow e' \bullet E'}{Mv,\ E \vdash \lambda(x : \tau).e \Longrightarrow \lambda(x : \tau).e' \bullet E'} \tag{34}$$

$$\frac{Mv,\ E \vdash e_1 \Longrightarrow e'_1 \bullet E' \qquad Mv,\ E' \vdash e_2 \Longrightarrow e'_2 \bullet E''}{Mv,\ E \vdash e_1 e_2 \Longrightarrow e'_1 e'_2 \bullet E''} \tag{35}$$

$$\frac{Mv,\ E \vdash e \Longrightarrow e' \bullet E'}{Mv,\ E \vdash \mathbf{new}\ c\ e \Longrightarrow \mathbf{new}\ c\ e' \bullet E'} \tag{36} \qquad \frac{}{Mv,\ E \vdash \mathbf{self} \Longrightarrow \mathbf{self} \bullet E} \tag{37}$$

$$\frac{x_m \notin \mathrm{dom}(E) \quad Ty(Mv, m) = \tau'_m \to \tau''_m \quad Mv(m) = [E_m;\ \lambda x.e_m]}{Mv,\ E \vdash \mathbf{super}.m \Longrightarrow x_m\,\mathbf{self} \bullet E \pm \{x_m \mapsto [E_m;\ \lambda \mathbf{self}.\lambda(x : \tau'_m).e_m]\}} \tag{38}$$

$$\frac{Mv,\ E \vdash e \Longrightarrow e' \bullet E'}{Mv,\ E \vdash e.m \Longrightarrow e'.m \bullet E'} \tag{39} \qquad \frac{M,\ E \vdash e \Longrightarrow e' \bullet E'}{Mv,\ E \vdash e.f \Longrightarrow e'.f \bullet E'} \tag{40}$$

$$\frac{Mv,\ E \vdash e_1 \Longrightarrow e'_1 \bullet E' \qquad Mv,\ E' \vdash e_2 \Longrightarrow e'_2 \bullet E''}{Mv,\ E \vdash e_1.f := e_2 \Longrightarrow e'_1.f := e'_2 \bullet E''} \tag{41}$$

$$\frac{}{Mv,\ E \vdash \langle\rangle_{\mathtt{F}} \Longrightarrow \langle\rangle_{\mathtt{F}} \bullet E} \tag{42}$$

$$\frac{Mv,\ E \vdash e_1 \Longrightarrow e'_1 \bullet E_1 \quad \cdots \quad Mv,\ E_{n-1} \vdash e_n \Longrightarrow e'_n \bullet E_n}{Mv,\ E \vdash \langle f_1 = e_1, \ldots, f_n = e_n \rangle_{\mathtt{F}} \Longrightarrow \langle f_1 = e'_1, \ldots, f_n = e'_n \rangle_{\mathtt{F}} \bullet E_n} \tag{43}$$

$$\frac{Mv,\ E \vdash e_1 \Longrightarrow e'_1 \bullet E' \qquad Mv,\ E' \vdash e_2 \Longrightarrow e'_2 \bullet E''}{Mv,\ E \vdash e_1 \oplus e_2 \Longrightarrow e'_1 \oplus e'_2 \bullet E''} \tag{44}$$

$$\frac{}{Mv,\ E \vdash () \Longrightarrow () \bullet E} \tag{45}$$

C The type system

All of our typing judgments are written in terms of contexts, which map trait names, class names, and variables to associated types. They also record free type variables.

$$\begin{array}{lcl} id & \in & \text{TNAMES} \cup \text{CNAMES} \cup \text{VARIABLES} \cup \{\mathbf{self},\ \mathbf{super}\} \\ \alpha & \in & \text{TYVARIABLES} \\ \Gamma & ::= & \epsilon \mid \Gamma,\ id : \tau \mid \Gamma,\ \alpha \end{array}$$

We assume that the sets TNAMES, CNAMES, VARIABLES, $\{\mathbf{self},\ \mathbf{super}\}$, and TYVARIABLES are mutually disjoint.

C.1 Context formation rules

$$\frac{}{\epsilon \vdash ok} \quad (46) \qquad \frac{\Gamma \vdash \tau \quad id \notin \mathrm{dom}(\Gamma)}{\Gamma,\ id : \tau \vdash ok} \quad (47)$$

$$\frac{\Gamma \vdash ok \quad \alpha \notin \Gamma}{\Gamma,\ \alpha \vdash ok} \quad (48)$$

C.2 Well-formed types

$$\frac{\Gamma \vdash ok \qquad \alpha \in \Gamma}{\Gamma \vdash \alpha} \quad (49) \qquad \frac{\Gamma, \vec{\alpha} \vdash \tau}{\Gamma \vdash \Lambda(\vec{\alpha}).\tau} \quad (50)$$

$$\frac{\Gamma \vdash \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle \quad \mathcal{S} \cup \mathcal{R} \subseteq \mathcal{L}}{\Gamma \vdash \langle\!| \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle;\ \mathcal{S};\ \mathcal{R} |\!\rangle} \quad (51) \qquad \frac{\Gamma \vdash \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle}{\Gamma \vdash \{\!| l : \tau_l{}^{l \in \mathcal{L}} |\!\}} \quad (52)$$

$$\frac{\Gamma \vdash ok \qquad \Gamma \vdash \tau_l \quad \text{forall } l \in \mathcal{L}}{\Gamma \vdash \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle} \quad (53) \qquad \frac{\Gamma \vdash ok \qquad \Gamma \vdash \tau_f \quad \text{forall } f \in \mathcal{F}}{\Gamma \vdash \langle f : \tau_f{}^{f \in \mathcal{F}} \rangle_{\mathsf{F}}} \quad (54)$$

$$\frac{\Gamma \vdash \tau_1 \quad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2} \quad (55) \qquad \frac{\Gamma \vdash ok}{\Gamma \vdash Unit} \quad (56)$$

C.3 Subtyping

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \tau \ <:\ \tau} \quad (57) \qquad \frac{\Gamma \vdash \langle l : \tau_l{}^{l \in \mathcal{L}_1} \rangle \qquad \mathcal{L}_2 \subseteq \mathcal{L}_1}{\Gamma \vdash \langle l : \tau_l{}^{l \in \mathcal{L}_1} \rangle \ <:\ \langle l : \tau_l{}^{l \in \mathcal{L}_2} \rangle} \quad (58)$$

$$\frac{\Gamma \vdash \tau_2' \ <:\ \tau_1' \quad \Gamma \vdash \tau_1'' \ <:\ \tau_2''}{\Gamma \vdash \tau_1' \to \tau_1'' \ <:\ \tau_2' \to \tau_2''} \quad (59)$$

C.4 Trait typing

$$\frac{\Gamma(t) = \Lambda(\vec{\alpha}).\tau \qquad \Gamma \vdash \vec{\tau} \qquad |\vec{\tau}| = |\vec{\alpha}|}{\Gamma \vdash t(\vec{\tau})\ :\ \tau[\vec{\tau}/\vec{\alpha}]} \quad (60)$$

$$\frac{\tau_{\mathbf{super}} = \langle l : \tau_l{}^{l \in \mathcal{S}} \rangle \qquad \tau_{\mathbf{self}} = \langle l : \tau_l{}^{l \in \mathcal{M} \cup \mathcal{R}} \rangle \quad \Gamma \vdash {}_{\tau_{\mathbf{super}};\, \tau_{\mathbf{self}}} M\ :\ \langle m : \tau_m{}^{m \in \mathcal{M}} \rangle \qquad \mathcal{M} \Cap \mathcal{R}}{\Gamma \vdash \langle\!| M;\ \langle l : \tau_l{}^{l \in \mathcal{R}} \rangle\ |\!\rangle\ :\ \langle\!| \langle l : \tau_l{}^{l \in \mathcal{M} \cup \mathcal{R}} \rangle;\ \mathcal{S};\ \mathcal{R} |\!\rangle} \quad (61)$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1\ :\ \langle\!| \langle l : \tau_l{}^{l \in \mathcal{L}_1} \rangle;\ \mathcal{S}_1;\ \mathcal{R}_1 |\!\rangle \quad \Gamma \vdash T_2\ :\ \langle\!| \langle l : \tau_l{}^{l \in \mathcal{L}_2} \rangle;\ \mathcal{S}_2;\ \mathcal{R}_2 |\!\rangle \\ \mathcal{M}_1 = \mathcal{L}_1 \setminus \mathcal{R}_1 \quad \mathcal{M}_2 = \mathcal{L}_2 \setminus \mathcal{R}_2 \quad \mathcal{M}_1 \Cap \mathcal{M}_2 \\ \mathcal{R}_1' = \mathcal{R}_1 \setminus \mathcal{M}_2 \quad \mathcal{R}_2' = \mathcal{R}_2 \setminus \mathcal{M}_1\end{array}}{\Gamma \vdash T_1 + T_2\ :\ \langle\!| \langle l : \tau_l{}^{l \in \mathcal{L}_1 \cup \mathcal{L}_2} \rangle;\ \mathcal{S}_1 \cup \mathcal{S}_2;\ \mathcal{R}_1' \cup \mathcal{R}_2' |\!\rangle} \quad (62)$$

$$\frac{\Gamma \vdash T \ : \ \{\!|\, \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle;\ \mathcal{S};\ \mathcal{R} \,|\!\}\quad m \in \mathcal{L} \setminus \mathcal{R}}{\Gamma \vdash T \setminus m \ : \ \{\!|\, \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle;\ \mathcal{S};\ \mathcal{R} \cup \{m\} \,|\!\}} \tag{63}$$

$$\frac{\begin{array}{c}\Gamma \vdash T \ : \ \{\!|\, \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle;\ \mathcal{S};\ \mathcal{R} \,|\!\} \\ m \in \mathcal{L} \setminus \mathcal{R} \qquad m' \notin \mathcal{L} \setminus \mathcal{R} \qquad \tau_{m'} = \tau_m\end{array}}{\Gamma \vdash T[m' \mapsto m] \ : \ \{\!|\, \langle l : \tau_l{}^{l \in \mathcal{L}},\ m' : \tau_{m'} \rangle;\ \mathcal{S};\ \mathcal{R} \setminus \{m'\} \,|\!\}} \tag{64}$$

C.5 Method suite typing

$$\frac{\begin{array}{c}\Gamma,\ \mathbf{super} : \tau_{\mathbf{super}},\ \mathbf{self} : \tau_{\mathbf{self}} \vdash \mu_m \ : \ \tau_m \quad \text{forall } m \in \mathcal{M} \\ \Gamma \vdash \tau_{\mathbf{self}} \ <: \ \tau_{\mathbf{super}} \qquad \Gamma \vdash \tau_{\mathbf{self}} \ <: \ \langle m : \tau_m{}^{m \in \mathcal{M}} \rangle\end{array}}{\Gamma \vdash {}_{\tau_{\mathbf{super}};\, \tau_{\mathbf{self}}}\langle \mu_m{}^{m \in \mathcal{M}} \rangle_{\mathtt{M}} \ : \ \langle m : \tau_m{}^{m \in \mathcal{M}} \rangle} \tag{65}$$

C.6 Method body typing

$$\frac{\Gamma,\ x : \tau_1 \vdash e \ : \ \tau_2}{\Gamma \vdash m\,(x : \tau_1) : \tau_2\{e\} \ : \ \tau_1 \to \tau_2} \tag{66}$$

C.7 Class typing

$$\frac{\Gamma \vdash ok}{\Gamma \vdash c \ : \ \Gamma(c)} \tag{67}$$

$$\frac{\Gamma \vdash ok}{\Gamma \vdash \mathbf{nil} \ : \ Unit \to \{\!|\ |\!\}} \tag{68}$$

$$\frac{\begin{array}{ccc}\Gamma,\ x : \tau \vdash e_F \ : \ \langle f : \tau_f{}^{f \in \mathcal{F}} \rangle_{\mathtt{F}} & & \Gamma,\ x : \tau \vdash e_{super} \ : \ \tau_{super} \\ \Gamma \vdash T \ : \ \{\!|\, \langle l : \tau_l{}^{l \in \mathcal{L}_T} \rangle;\ \mathcal{S}_T;\ \mathcal{R}_T \,|\!\} & & \Gamma \vdash C \ : \ \tau_{super} \to \{\!|\, l : \tau_l{}^{l \in \mathcal{L}_c} \,|\!\} \\ \mathcal{S}_T \subseteq \mathcal{L}_c \qquad \mathcal{R}_T \subseteq (\mathcal{L}_c \cup \mathcal{F}) & & \mathcal{F} \pitchfork \mathcal{L}_c \qquad \mathcal{L} = \mathcal{F} \cup \mathcal{L}_T \cup \mathcal{L}_c\end{array}}{\Gamma \vdash \lambda(x : \tau).(\mathbf{super}\ e_{super}) \oplus e_F \ \mathbf{in}\ T\ \mathbf{extends}\ C \ : \ \tau \to \{\!|\, l : \tau_l{}^{l \in \mathcal{L}} \,|\!\}} \tag{69}$$

C.8 Well-typed expressions

$$\frac{\Gamma \vdash ok}{\Gamma \vdash x \ : \ \Gamma(x)} \quad (70) \qquad \frac{\Gamma,\ x : \tau_1 \vdash e \ : \ \tau_2}{\Gamma \vdash \lambda(x : \tau_1).e \ : \ \tau_1 \to \tau_2} \quad (71)$$

$$\frac{\Gamma \vdash e_1 \ : \ \tau_2 \to \tau \quad \Gamma \vdash e_2 \ : \ \tau_2}{\Gamma \vdash e_1\ e_2 \ : \ \tau} \quad (72) \qquad \frac{\Gamma \vdash c \ : \ \tau \to \{\!|\, l : \tau_l{}^{l \in \mathcal{L}} \,|\!\} \quad \Gamma \vdash e \ : \ \tau}{\Gamma \vdash \mathbf{new}\ c\ e \ : \ \langle l : \tau_l{}^{l \in \mathcal{L}} \rangle} \quad (73)$$

$$\frac{\Gamma \vdash ok}{\Gamma \vdash \mathbf{self} \ : \ \Gamma(\mathbf{self})} \quad (74) \qquad \frac{\Gamma \vdash \Gamma(\mathbf{super}) \ <: \ \langle m : \tau_m \rangle}{\Gamma \vdash \mathbf{super}.m \ : \ \tau_m} \quad (75)$$

$$\frac{\Gamma \vdash e \ : \ \tau \quad \Gamma \vdash \tau \ <: \ \langle l : \tau_l \rangle}{\Gamma \vdash e.l \ : \ \tau_l} \quad (76) \qquad \frac{\Gamma \vdash e_1 \ : \ \tau \quad \Gamma \vdash \tau \ <: \ \langle f : \tau_f \rangle \quad \Gamma \vdash e_2 \ : \ \tau_f}{\Gamma \vdash e_1.f := e_2 \ : \ Unit} \quad (77)$$

$$\frac{\Gamma \vdash ok \qquad \Gamma \vdash e_f \ : \ \tau_f \quad \text{forall } f \in \mathcal{F}}{\Gamma \vdash \langle f = e_f{}^{f \in \mathcal{F}} \rangle_{\mathsf{F}} \ : \ \langle f : \tau_f{}^{f \in \mathcal{F}} \rangle_{\mathsf{F}}} \quad (78)$$

$$\frac{\Gamma \vdash e_1 \ : \ \langle f : \tau_f{}^{f \in \mathcal{F}_1} \rangle_{\mathsf{F}} \quad \Gamma \vdash e_2 \ : \ \langle f : \tau_f{}^{f \in \mathcal{F}_2} \rangle_{\mathsf{F}} \quad F_1 \pitchfork F_2}{\Gamma \vdash e_1 \oplus e_2 \ : \ \langle f : \tau_f{}^{f \in \mathcal{F}_1 \cup \mathcal{F}_2} \rangle_{\mathsf{F}}} \quad (79)$$

$$\frac{\Gamma \vdash ok}{\Gamma \vdash () \ : \ Unit} \quad (80) \qquad \frac{\Gamma \vdash e \ : \ \tau \quad \Gamma \vdash \tau \ <: \ \tau'}{\Gamma \vdash e \ : \ \tau'} \quad (81)$$

C.9 Declaration typing

$$\frac{t \notin \mathrm{dom}(\Gamma) \qquad \Gamma, \vec{\alpha} \vdash T \ : \ \tau}{\Gamma \vdash t = (\vec{\alpha})T \ \Rightarrow \ \Gamma,\ t : \Lambda(\vec{\alpha}).\tau} \quad (82) \qquad \frac{c \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash C \ : \ \tau}{\Gamma \vdash c = C \ \Rightarrow \ \Gamma,\ c : \tau} \quad (83)$$

$$\frac{x \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash e \ : \ \tau}{\Gamma \vdash x = e \ \Rightarrow \ \Gamma,\ x : \tau} \quad (84)$$

C.10 Program typing

$$\frac{\Gamma \vdash D \ \Rightarrow \ \Gamma' \qquad \Gamma' \vdash_P P \ : \ \tau}{\Gamma \vdash_P D; P \ : \ \tau} \quad (85) \qquad \frac{\Gamma \vdash e \ : \ \tau}{\Gamma \vdash_P e \ : \ \tau} \quad (86)$$

D Run-time typing

We introduce the notion of a store type, denoted by the meta-variable Σ. Store types map addresses to closed types τ.

D.1 Well-formed store type

$$\frac{\forall a \in \mathrm{dom}(\Sigma),\ \epsilon \vdash \Sigma(a)}{\Sigma \vdash ok} \tag{87}$$

D.2 Well-typed environment

$$\frac{\Sigma \vdash ok}{\Sigma \vdash \epsilon : \epsilon} \tag{88}$$

$$\frac{\Sigma \vdash E : \Gamma \qquad \Sigma \vdash_{(\vec{\alpha})} tv : \tau \qquad t \notin \mathrm{dom}(E)}{\Sigma \vdash E \pm \{t \mapsto (\vec{\alpha})tv\} : (\Gamma,\ t\ :\ \Lambda(\vec{\alpha}).\tau)} \tag{89}$$

$$\frac{\Sigma \vdash E : \Gamma \qquad \Sigma \vdash cv : \tau \qquad c \notin \mathrm{dom}(E)}{\Sigma \vdash E \pm \{c \mapsto cv\} : (\Gamma,\ c\ :\ \tau)} \tag{90}$$

$$\frac{\Sigma \vdash E : \Gamma \qquad \Sigma \vdash ev : \tau' \qquad \epsilon \vdash \tau' <: \tau \qquad x \notin \mathrm{dom}(E)}{\Sigma \vdash E \pm \{x \mapsto ev\} : (\Gamma,\ x\ :\ \tau)} \tag{91}$$

D.3 Store typing

$$\frac{\Sigma \vdash ok \qquad \mathrm{dom}(\Sigma) = \mathrm{dom}(S) \qquad \forall a \in \mathrm{dom}(S)\ \Sigma \vdash S(a) : \Sigma(a)}{\vdash S : \Sigma} \tag{92}$$

D.4 Typing rules

$$\frac{\mathcal{R} \pitchfork \mathcal{M} \qquad \tau_{\mathbf{self}} = \langle l : {\tau_l}^{l \in \mathcal{R} \cup \mathcal{M}} \rangle \qquad \tau_{\mathbf{super}} = \langle l : {\tau_l}^{l \in \mathcal{S}} \rangle \quad \Sigma \vdash_{(\vec{\alpha});\tau_{\mathbf{super}};\tau_{\mathbf{self}}} Mv : \langle m : {\tau_m}^{m \in \mathcal{M}} \rangle}{\Sigma \vdash_{(\vec{\alpha})} \{\!|\, Mv;\ \langle l : {\tau_l}^{l \in \mathcal{R}} \rangle \,|\!\} : \{\!|\, \langle l : {\tau_l}^{l \in \mathcal{R} \cup \mathcal{M}} \rangle;\ \mathcal{S};\ \mathcal{R} \,|\!\}} \tag{93}$$

$$\frac{\Sigma \vdash ok \quad \vec{\alpha} \vdash \tau_{\mathbf{self}} <: \tau_{\mathbf{super}} \qquad \vec{\alpha} \vdash \tau_{\mathbf{self}} <: \langle m : {\tau_m}^{m \in \mathcal{M}} \rangle \quad \Sigma \vdash_{(\vec{\alpha});\tau_{\mathbf{super}};\tau_{\mathbf{self}}} \mu v_m : \tau_m \qquad \text{forall } m \in \mathcal{M}}{\Sigma \vdash_{(\vec{\alpha});\tau_{\mathbf{super}};\tau_{\mathbf{self}}} \langle \mu {v_m}^{m \in \mathcal{M}} \rangle_{\mathtt{M}} : \langle m : {\tau_m}^{m \in \mathcal{M}} \rangle} \tag{94}$$

$$\frac{\Sigma \vdash E : \Gamma \qquad \Gamma,\ \vec{\alpha},\ \mathbf{super}\ :\ \tau_{\mathbf{super}},\ \mathbf{self}\ :\ \tau_{\mathbf{self}},\ x\ :\ \tau' \vdash e\ :\ \tau''}{\Sigma \vdash_{(\vec{\alpha});\tau_{\mathbf{super}};\tau_{\mathbf{self}}} (m : \tau' \to \tau'' = [E;\ \lambda x.e]) : \tau' \to \tau''} \tag{95}$$

$$\frac{\Sigma \vdash \lambda v : \tau \to \langle f : {\tau_f}^{f \in \mathcal{F}} \rangle_{\mathtt{F}} \qquad \Sigma \vdash_{();\langle\rangle;\langle l:\tau_l{}^{l \in \mathcal{F} \cup \mathcal{M}}\rangle} Mv : \langle m : {\tau_m}^{m \in \mathcal{M}} \rangle}{\Sigma \vdash \{\!|\, \lambda v;\ Mv \,|\!\} : \tau \to \{\!|\, l : {\tau_l}^{l \in \mathcal{F} \cup \mathcal{M}} \,|\!\}} \tag{96}$$

$$\frac{\Sigma \vdash ok}{\Sigma \vdash a : \Sigma(a)} \tag{97}$$

$$\frac{\Sigma \vdash E : \Gamma \qquad \Gamma,\ x\ :\ \tau' \vdash e\ :\ \tau''}{\Sigma \vdash [E;\ \lambda x.e] : \tau' \to \tau''} \tag{98}$$

$$\frac{\Sigma \vdash ok \qquad \Sigma \vdash ev_f : \tau'_f \qquad \epsilon \vdash \tau'_f <: \tau_f \quad \text{forall } f \in \mathcal{F}}{\Sigma \vdash \langle f = {ev_f}^{f \in \mathcal{F}} \rangle_{\mathtt{F}} : \langle f : {\tau_f}^{f \in \mathcal{F}} \rangle_{\mathtt{F}}} \tag{99}$$

$$\frac{\Sigma \vdash ok}{\Sigma \vdash () : Unit} \tag{100}$$

$$\frac{\Sigma \vdash fv : \langle f : \tau_f{}^{f \in \mathcal{F}} \rangle_{\mathsf{F}} \qquad \Sigma \vdash_{();\langle\rangle;\langle l:\tau_l{}^{l \in \mathcal{F} \cup \mathcal{M}}\rangle} Mv : \langle m : \tau_m{}^{m \in \mathcal{M}} \rangle}{\Sigma \vdash \langle fv;\ Mv \rangle : \langle l : \tau_l{}^{l \in \mathcal{F} \cup \mathcal{M}} \rangle} \tag{101}$$

E Type Soundness: Auxiliary lemmas

Theorem E.1 (Subject reduction) *If* $\Gamma \vdash_P P : \tau$ *and* $E,\ S \vdash P \longrightarrow ev \bullet E' \bullet S'$ *and* $\Sigma \vdash E : \Gamma$ *and* $\vdash S : \Sigma$*, then there exist context* $\Gamma' \preceq \Gamma$ *and store typing* $\Sigma' \preceq \Sigma$ *such that* $\Sigma' \vdash E' : \Gamma'$ *and* $\vdash S' : \Sigma'$ *and* $\Sigma' \vdash ev : \tau'$ *and* $\Gamma' \vdash \tau' <: \tau$*.*

Definition E.1 (Expression evaluation height) *We define the evaluation height of an expression* e *in environment* E *and store* S *to be* $h_{E,\,S}(e)$*, where* $h_{E,\,S}(e)$ *is defined in Figure 6.*

Lemma E.1 (Soundness of expression evaluation) *If* $\Gamma \vdash e : \tau$ *and* $\Sigma \vdash E : \Gamma$ *and* $\vdash S : \Sigma$ *and there exists an* n *such that* $h_{E,\,S}(e) = n$*, then there exist a store typing* $\Sigma' \preceq \Sigma$*, a store* S'*, an expression value* ev*, and a type* τ' *such that* $E,\ S \vdash e \longrightarrow ev \bullet S'$ *and* $\vdash S' : \Sigma'$ *and* $\Sigma' \vdash ev : \tau'$ *and* $\Gamma' \vdash \tau' <: \tau$*.*

The proof is by induction n. □

Definition E.2 (Program evaluation height) *We define the evaluation height of a program expression* P *in environment* E *and store* S *to be* $h^P_{E,\,S}(e)$*, where* $h^P_{E,\,S}(e)$ *is defined as follows:*

$$h_{E,\,S}(t = (\vec{\alpha})T; P) = 1 + \begin{cases} h^P_{E',\,S}(P) & \textit{if } E,\ S \vdash t = (\vec{\alpha})T \longrightarrow E' \bullet S \\ 1 & \textit{otherwise} \end{cases}$$

$$h_{E,\,S}(c = C; P) = 1 + \begin{cases} h^P_{E',\,S}(P) & \textit{if } E,\ S \vdash c = C \longrightarrow E' \bullet S \\ 1 & \textit{otherwise} \end{cases}$$

$$h_{E,\,S}(x = e; P) = 1 + h_{E,\,S}(e) + \begin{cases} h^P_{E',\,S'}(P) & \textit{if } E,\ S \vdash x = e \longrightarrow E' \bullet S' \\ 1 & \textit{otherwise} \end{cases}$$

$$h^P_{E,\,S}(e) = h_{E,\,S}(e)$$

$$
\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) \\
 & + & \left\{ \begin{array}{ll} h_{E,\,S_1}(e_2) & \text{if } E,\ S \vdash e_1 \longrightarrow [E_1;\ \lambda x.e] \bullet S_1 \\ 1 & \text{otherwise} \end{array} \right. \\
 & + & \left\{ \begin{array}{ll} h_{E_1 \pm \{x \mapsto ev_2\},\,S_2}(e) & \text{if } E,\ S \vdash e_1 \longrightarrow [E_1;\ \lambda x.e] \bullet S_1 \\ & \text{and } E,\ S_1 \vdash e_2 \longrightarrow ev_2 \bullet S_2 \\ 1 & \text{otherwise} \end{array} \right. \\
 & & \\
h_{E,\,S}(\mathbf{new}\ c\ e) & = & 1 \\
 & + & h_{E,\,S}(e) \\
 & + & \left\{ \begin{array}{ll} h_{E_F \pm \{x \mapsto ev\},\,S_1}(e_F) & \text{if } E,\ S \vdash c \longrightarrow \{\!|\, [E_F;\ \lambda x.e_F];\ Mv \,|\!\} \bullet S \\ & \text{and } E,\ S \vdash e \longrightarrow ev \bullet S_1 \\ 1 & \text{otherwise} \end{array} \right. \\
 & & \\
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) \\
 & + & \left\{ \begin{array}{ll} h_{E,\,S_1}(e_2) & \text{if } E,\ S \vdash e_1 \longrightarrow ev_1 \bullet S_1 \\ 1 & \text{otherwise} \end{array} \right. \\
 & & \\
h_{E,\,S}(\langle\rangle_{\mathsf{F}}) & = & 1 \\
h_{E,\,S_0}(\langle f_1 = e_1, \ldots, f_n = e_n \rangle_{\mathsf{F}}) & & \\
 & = & 1 \\
 & + & h_{E,\,S_0}(e_1) \\
 & \vdots & \\
 & + & \left\{ \begin{array}{ll} h_{E,\,S_{n-1}}(e_n) & \text{if } E,\ S_0 \vdash e_1 \longrightarrow ev_1 \bullet S_1 \\ & \vdots \\ & \text{and } E,\ S_{n-2} \vdash e_{n-1} \longrightarrow ev_{n-1} \bullet S_{n-1} \\ 1 & \text{otherwise} \end{array} \right. \\
 & & \\
h_{E,\,S}(e_1 \oplus e_2) & = & 1 \\
 & + & h_{E,\,S}(e_1) \\
 & + & \left\{ \begin{array}{ll} h_{E,\,S_1}(e_2) & \text{if } E,\ S \vdash e_1 \longrightarrow ev_1 \bullet S_1 \\ 1 & \text{otherwise} \end{array} \right. \\
 & & \\
h_{E,\,S}(()) & = & 1
\end{array}
$$

Figure 6: Definition of function for calculating the evaluation height of an expression.