Foundations for MOBY classes

Kathleen Fisher
AT&T Labs, Research

John Reppy
Bell Labs, Lucent Technologies

December 2, 1998

Abstract

MOBY is a statically typed, ML-like language that supports class-based object-oriented programming. Our design philosophy is to keep language features independent, but complementary. MOBY's support for class-based object-oriented programming is a prime example of this philosophy: classes provide support for object creation and inheritance, but we rely on the module system for information hiding and on object types for subtyping. The resulting design is both simple and expressive.

In this paper, we give semantics to the object-oriented features of MOBY in the style of Harper and Stone. We formalize the surface language in an *external language*, called MICROMOBY, and define a type-preserving translation from MICROMOBY to an *internal language*, called MOC. We designed MICROMOBY to study the class and object constructs in MOBY. The internal language MOC is a new extensible-object calculus designed to provide semantic foundations for MICROMOBY. We present both the static and dynamic semantics of MOC, and prove a subject reduction theorem. We also present the translation from MICROMOBY to MOC and prove that this translation preserves types. A consequence of this theorem and type-safety for MOC is that MICROMOBY is type-safe.

1 Introduction

MOBY is a statically typed, ML-like language that supports class-based object-oriented programming. In contrast to languages like C++ or JAVA, which have strong class mechanisms but weak module systems, MOBY provides a simple class construct and a rich, ML-style module system. By combining modules, classes, and objects, we can express in MOBY most of the class-based idioms supported by C++ and JAVA classes. Specifically, MOBY has three distinct mechanisms to cover the feature set of typical class-based languages:

- *Modules and module signatures.* MOBY provides an ML-style module system [Mac84, Ler94, MTHM97]. Such modules structure the name space by grouping definitions, including other modules. Module signatures serve as the types of modules. The *enrichment* relation between signatures is analogous to subtyping. By constraining a module with a signature, which is called *signature ascription*, we can control the external view of the definitions in the module. This mechanism provides both data and type abstraction.

- *Classes and class interfaces.* Classes in MOBY provide implementations of objects and support implementation inheritance. Because of these two roles, a class has two kinds of clients:

object clients, which use the class to create new objects, and *class clients*, which use it to derive new classes. Consequently, classes provide two views of their methods and fields: an *object view* and a *class view*. These views are captured in the *type* of the class, which we call its *interface*. Class interfaces are used in module signatures to specify the object and class views of classes visible outside the module.

- *Objects and object types.* Objects are run-time values that support dynamic dispatching. MOBY has no primitive form of object creation; instead one uses a class *maker* (a.k.a. constructor) to create objects. Object types describe the interface of an object, but do not specify the implementation. Subtyping on object types is structural.

A more complete description of MOBY's object-oriented features can be found in [FR98a][1].

In this paper, we define the formal semantics of the object-oriented portion of MOBY using the approach of Harper and Stone [HS97]. Namely, we craft an *external language* (EL) that formalizes in a direct fashion the aspects of the MOBY surface language that we wish to study. We then define an *internal language* (IL) rich enough to express all of the EL features, but simple enough to prove type safe via a subject reduction theorem. A formal, type-preserving translation from the EL to the IL establishes the semantics for EL programs and proves the EL type safe.

We adopt for our EL the language MICROMOBY, an idealized version of MOBY with only features related to classes and objects. Classes and class interfaces in MICROMOBY are quite similar to the corresponding features in MOBY, although they are written more concisely. The major non-syntactic simplification is that in MOBY a class definition contains one or more *maker* functions (a.k.a. constructors), whereas in MICROMOBY a class only has a single maker function that has the class's name. Just as in MOBY, MICROMOBY objects are created from classes and can be manipulated via method dispatch, field selection, and field update operations. Section 2 describes MICROMOBY in detail, including its formal syntax and typing rules. Section 5 discusses how the framework of this paper can be expanded to handle additional features of MOBY.

For our IL, we have developed a new extensible-object calculus, called MOC. This calculus has two kinds of object: extensible objects that model inheritance and object initialization, and non-extensible objects that model MICROMOBY's objects and provide rich subtyping. We call extensible objects *class values*[2] as they model classes. The **`new`** operator instantiates a class value to create an object value. To handle properly the semantics of private members, we have equipped our objects with *dictionaries* in the style of Riecke and Stone [RS99]. We type our objects with recursive object types, but do not have the more complicated *mytype* mechanism [Fis96, AC96]. In Section 3, we give the formal syntax, typing rules, and operational semantics of MOC and prove type soundness for it.

After defining our EL and IL, we give a compositional, type-directed translation from EL terms to IL terms. This formal translation takes the form of annotated MICROMOBY typing rules, where each such rule calculates the type of a MICROMOBY term and computes the corresponding IL term. We prove that the MICROMOBY type given to the EL term during translation may also be given to the resulting MOC term using the MOC typing rules. Such a result is possible as the types of MICROMOBY are a subset of those of MOC. Under the translation, a MICROMOBY class becomes

[1] This paper and other information on MOBY appears at `http://www.cs.bell-labs.com/~jhr/moby/`.

[2] Classes in MOC are first-class *values*, which is not the case in MICROMOBY.

a MOC function that takes the class's maker argument and returns a class value, which contains the methods and fields defined by the class. For derived classes, the translation produces a MOC function that first calls the MOC function for the parent class, passing the appropriate arguments. It then extends the resulting extensible object with the methods and fields of the deriving class. MICROMOBY's **new** operator is translated to a MOC term that invokes the relevant class's maker function with the appropriate argument and then uses MOC's **new** operator to create an object value from the resulting class value. We discuss the translation in detail in Section 4.

The contributions of this work are threefold: first, we have created a type-safe object calculus suitable for defining the semantics of a surface-level programming language; second, we have used this object calculus to give semantics to the object-oriented portions of MOBY via a formal translation; and third, we have shown these semantics to be type-safe by proving that the translation preserves types.

2 MICROMOBY

In this section, we present our external language MICROMOBY, which is designed to capture the significant aspects of MOBY's object and class mechanisms. A MICROMOBY program consists of a sequence of value and class bindings followed by an expression. Although MICROMOBY does not have a module mechanism, it does have *class interface ascription*, which is the aspect of signature ascription that relates to classes.[3] This mechanism supports private members and provides a weak form of package scope. The other major simplification that we make with respect to MOBY is that each MICROMOBY class defines a single maker function; the name of the class serves as the name of the maker.

As an example, the following code defines a `Point` class in MICROMOBY syntax:

```
class Point (x0 : Int) : {
  public meth getX : Unit -> Int
  public meth move : Int -> Unit
} = {
  field x : Int = x0
  public meth getX () : Int => self.x
  public meth move (dx : Int) : Unit => self.x := self.x+dx
}
```

This class defines an `x` field, which is initialized from maker argument `x0`, and two public methods, `getX` and `move`, that return the value stored in `x` and increment this amount by `dx`, respectively. The member specifications form a class interface; by ascribing the `Point` class with an interface that does not mention `x`, we make the `x` field private. We can define a new class of fast-points by inheriting from `Point` and overriding the `move` method:

[3] It is fairly straightforward to add one of the existing module system semantics (*e.g.*, Harper-Stone [HS97]) to our system, but this greatly increases the sizes of the formal systems. By including class interface ascription, we get the most interesting module/class interaction without the overhead of defining the semantics of modules.

```
class FastPoint (x0 : Int) : {
  public meth getX : Unit -> Int
  public meth move : Int -> Unit
} = {
  inherits Point(x0)
  public meth move (dx : Int) : Unit => super.move (2*dx)
}
```

Note that since `FastPoint` does not have access to the `x` field of the `Point` class, it must use `Point`'s `move` method to implement its own `move` method. Also note that `Point` and `Fast-Point` both have the same interface and implement the same object type.

2.1 Syntax

Figure 1 lists the lexically distinct classes of identifiers used in MICROMOBY, and Figure 2 presents the syntax of MICROMOBY terms. We assume that there exists some global set of predefined types (TYID) with a predefined subtyping structure. We assume that field and method labels are disjoint and use $\mathcal{F}$ and $\mathcal{M}$ to represent finite sets of these labels, respectively. When the distinction between fields and methods is unimportant, we use $\mathcal{L}$ to denote a set of labels.

2.2 Static semantics

The static semantics of MICROMOBY are given in a standard operational style. We start by defining the set of *semantic types* (TYPE), which we use τ to range over. The typing rules assign these types to expressions; they have the following syntax:

$$
\begin{array}{rcll}
\tau & ::= & \mathsf{T} & \text{primitive type} \\
 & | & \alpha & \text{type variable} \\
 & | & \tau \rightarrow \tau & \text{function type} \\
 & | & \tau \times \tau & \text{pair type} \\
 & | & \mathbf{obj}\,\alpha.\{\!|\, \mathsf{f} : \tau_{\mathsf{f}}{}^{\mathsf{f} \in \mathcal{F}};\ \mathsf{m} : \tau_{\mathsf{m}}{}^{\mathsf{m} \in \mathcal{M}} \,|\!\} & \text{object type}
\end{array}
$$

where $\mathsf{T} \in$ TYID is a global predefined type and $\alpha \in$ TYVAR is a type variable. Note that object types are recursive: when a type variable α occurs within the member types τ_{l}, where $\mathsf{l} \in \mathcal{F} \cup \mathcal{M}$, α refers to the containing type $\mathbf{obj}\,\alpha.\{\!|\, \mathsf{f} : \tau_{\mathsf{f}}{}^{\mathsf{f} \in \mathcal{F}};\ \mathsf{m} : \tau_{\mathsf{m}}{}^{\mathsf{m} \in \mathcal{M}} \,|\!\}$. When the field/method distinction is not important, we write $\mathbf{obj}\,\alpha.\{\!|\, \mathsf{l} : \tau_{\mathsf{l}}{}^{\mathsf{l} \in \mathcal{L}} \,|\!\}$ for an object type.

We define the standard well-formedness, equality, and subtyping judgements on types in Appendix A. These judgements require the following sets of equality and subtyping assumptions:

$$
\begin{array}{rcl}
A & \in & \mathrm{Fin}(\text{TYVAR}) \\
\mathcal{E} & \in & \text{EQTYASSUM} = (\text{TYVAR} \times \text{TYVAR}) \\
\mathcal{S} & \in & \text{SUBTYASSUM} = (\text{TYID} \cup \text{TYVAR}) \stackrel{\text{fin}}{\rightarrow} \text{TYPE}
\end{array}
$$

A subtyping environment $\mathcal{S}$ maps global predefined types and type variables to their declared supertypes. We assume there is an initial set of subtyping assumptions $\mathcal{S}_0$ on the types in TYID.

In addition to the types, we define the set of *class types* (CLASSTY) as terms of the following form:

t	$\in$	TYVARID	type variables
T	$\in$	TYID = {Unit, Int, ...}	type identifiers
C	$\in$	CLSID	class identifiers
f	$\in$	FIELDID	field labels
m	$\in$	METHID	method labels
l	$\in$	LABID = FIELDID $\cup$ METHID	object labels
c	$\in$	CONST	constants
x	$\in$	VALID	value identifiers

Figure 1: Classes of MICROMOBY identifiers

prog	::=	*exp*
	|	*dcl prog*
dcl	::=	**class** C **:** *interface* **=** *clsexp*
	|	**val** x **=** *exp*
clsexp	::=	C
	|	(x **:** *ty*) { *fld meth* }
	|	(x **:** *ty*) { **inherits** C(*exp*) *fld meth* }
fld	::=	ϵ
	|	*fld fld*
	|	**public field** f **:** *ty* **=** *exp*
	|	**field** f **:** *ty* **=** *exp*
meth	::=	ϵ
	|	*meth meth*
	|	**public meth** m **(**x **:** *ty***)** **:** *ty* **=>** *exp*
	|	**meth** m **(**x **:** *ty***)** **:** *ty* **=>** *exp*
exp	::=	c
	|	x
	|	**fn (**x **:** *ty***)** **:** *ty* **=>** *exp*
	|	*exp* **(***exp'***)**
	|	**new** C **(** *exp* **)**
	|	**self**
	|	**super**.m
	|	*exp*.l
	|	*exp*.f **:=** *exp*

ty	::=	T
	|	t
	|	*ty* **->** *ty*
	|	*ty* ***** *ty*
	|	**obj** t { *memb* }
memb	::=	ϵ
	|	*memb memb*
	|	**field** f **:** *ty*
	|	**meth** m **:** *ty*
interface	::=	(*ty*) { *fldspec methspec* }
fldspec	::=	ϵ
	|	*fldspec fldspec*
	|	**public field** f **:** *ty*
	|	**field** f **:** *ty*
methspec	::=	ϵ
	|	*methspec methspec*
	|	**public meth** m **:** *ty*
	|	**meth** m **:** *ty*

Figure 2: Syntax of MICROMOBY terms

$$\theta = \{\!|\, \mathsf{f} : \tau_{\mathsf{f}}{}^{\mathsf{f} \in \mathcal{F}} ; \mathsf{m} : \tau_{\mathsf{m}}{}^{\mathsf{m} \in \mathcal{M}} \,|\!\}$$

As with object types, we write $\{\!|\, \mathsf{l} : \tau_{\mathsf{l}}{}^{\mathsf{l} \in \mathcal{L}} \,|\!\}$ when the field/method distinction is not important. The definition of a well-formed class type can be found in Appendix A. Class types are used in the rules for type checking class definitions and to represent the *class views* of classes in the environment. Thus, we find it useful to define a number of operations on them.

Definition 1 Let $\theta = \{\!|\, \mathsf{f} : \tau_{\mathsf{f}}{}^{\mathsf{f} \in \mathcal{F}} ; \mathsf{m} : \tau_{\mathsf{m}}{}^{\mathsf{m} \in \mathcal{M}} \,|\!\}$ be a well-formed class type. Then we define the *fields*, *methods*, and *labels* of θ as follows: $\mathrm{Fields}(\theta) = \mathcal{F}$, $\mathrm{Methods}(\theta) = \mathcal{M}$, and $\mathrm{Labels}(\theta) = \mathcal{F} \cup \mathcal{M}$.

It is often convenient to view a class type θ as a finite map from labels to types. When $\mathsf{l} \in \mathrm{Labels}(\theta)$, we use the notation $\theta.\mathsf{l}$ to denote the type to which θ maps l. We use the notation $\theta \downarrow \mathcal{P}$ to denote the restriction of θ to the labels in $\mathcal{P}$.

Definition 2 Let $\theta = \{\!|\, \mathsf{f} : \tau_{\mathsf{f}}{}^{\mathsf{f} \in \mathcal{F}} ; \mathsf{m} : \tau_{\mathsf{m}}{}^{\mathsf{m} \in \mathcal{M}} \,|\!\}$ and $\theta' = \{\!|\, \mathsf{f} : \tau'_{\mathsf{f}}{}^{\mathsf{f} \in \mathcal{F}'} ; \mathsf{m} : \tau'_{\mathsf{m}}{}^{\mathsf{m} \in \mathcal{M}'} \,|\!\}$ be well-formed class types, and let $O_f = \mathrm{Fields}(\theta) \cap \mathrm{Fields}(\theta')$ and $O_m = \mathrm{Methods}(\theta) \cap \mathrm{Methods}(\theta')$ be the sets of overlapping fields and methods. If it is the case that

- for all $\mathsf{f} \in O_f$ we have $\{\} \vdash \tau_{\mathsf{f}} = \tau'_{\mathsf{f}}$ and
- for all $\mathsf{m} \in O_m$ we have $\mathcal{S}_0 \vdash \tau'_{\mathsf{m}} <: \tau_{\mathsf{m}}$,

then we define the *extension of* θ *by* θ' to be

$$\theta \oplus \theta' = \{\!|\, \mathsf{f} : \tau_{\mathsf{f}}{}^{\mathsf{f} \in (\mathcal{F} \setminus O_f)}, \mathsf{f} : \tau'_{\mathsf{f}}{}^{\mathsf{f} \in \mathcal{F}'} ; \mathsf{m} : \tau_{\mathsf{m}}{}^{\mathsf{m} \in (\mathcal{M} \setminus O_m)}, \mathsf{m} : \tau'_{\mathsf{m}}{}^{\mathsf{m} \in \mathcal{M}'} \,|\!\}$$

The typing rules use a number of different semantic objects:

$$\begin{array}{lcl}
\mathcal{T} & \in & \textsc{TyEnv} = \textsc{TyVarId} \xrightarrow{\text{fin}} \textsc{TyVar} \\
\mathcal{I} & \in & \textsc{Interface} = (\textsc{Type} \times \textsc{ClassTy} \times \mathrm{Fin}(\textsc{LabId})) \\
\mathcal{C} & \in & \textsc{ClsEnv} = \textsc{ClsId} \xrightarrow{\text{fin}} \textsc{Interface} \\
\mathcal{V} & \in & \textsc{ValEnv} = (\textsc{ValId} \cup \{\texttt{self}, \texttt{super}\}) \xrightarrow{\text{fin}} \textsc{Type} \\
\Gamma & \in & \textsc{Env} = (\textsc{ClsEnv} \times \textsc{ValEnv})
\end{array}$$

A *type environment* $\mathcal{T}$ is used to map bound syntactic type variables to their semantic counterparts when checking object type definitions. A *class interface* is a triple $(\tau, \theta, \mathcal{P})$, where τ is the type of the class maker argument, θ is the type of the fields and methods in the class, and $\mathcal{P}$ is the set of public field and method labels. MICROMOBY typing judgements are written with respect to an environment Γ, which is a pair of a class environment ($\mathcal{C}$) that maps class names to class interfaces and a value environment ($\mathcal{V}$) that maps variables to types. We use the notation ($\mathcal{C}$ of Γ) and ($\mathcal{V}$ of Γ) to extract the class and value environments from Γ, respectively.

2.3 Selected typing rules

To conclude the discussion of MICROMOBY, we examine the typing rules for key parts of the class mechanism. The complete static semantics is given as part of the translation to MOC in Appendix B. We have the following forms of typing judgements for MICROMOBY:

$\Gamma \vdash \mathit{prog} \triangleright \tau$	Program *prog* has type τ.
$\Gamma \vdash \mathit{dcl} \triangleright \Gamma'$	Declaration *dcl* yields environment Γ'.
$\Gamma \vdash \mathit{clsexp} \triangleright (\tau, \theta, \mathcal{P})$	Class expression *clsexp* has interface $(\tau, \theta, \mathcal{P})$.
$\Gamma \vdash \mathit{fld} \triangleright (\theta, \mathcal{P})$	Field definition *fld* has type $(\theta, \mathcal{P})$.
$\Gamma \vdash \mathit{meth} \triangleright (\theta, \mathcal{P})$	Method definition *meth* has type $(\theta, \mathcal{P})$.
$\Gamma \vdash \mathit{exp} \triangleright \tau$	Expression *exp* has type τ.

In the typing of field and method definitions, the class type θ records the names and types of the defined members, while $\mathcal{P}$ records which of those members are public.

The first rule that we consider is that for class declarations, which defines the semantics of class interface ascription:

$$\frac{\Gamma \vdash \mathit{clsexp} \triangleright (\tau', \theta', \mathcal{P}') \quad \vdash \mathit{interface} \triangleright (\tau, \theta, \mathcal{P}) \\ \mathcal{S}_0 \vdash \tau <: \tau' \quad \theta' <:_w \theta \quad \mathcal{P} \subseteq \mathcal{P}'}{\Gamma \vdash \texttt{class}\ \mathsf{C}\ \texttt{:}\ \mathit{interface}\ \texttt{=}\ \mathit{clsexp} \triangleright \Gamma \pm \{\mathsf{C} \mapsto (\tau, \theta, \mathcal{P})\}}$$

In the antecedents of this rule, we derive two distinct class interfaces: one for the implementation $(\tau', \theta', \mathcal{P}')$ and one for the declared interface $(\tau, \theta, \mathcal{P})$. We require that the declared argument type be a subtype of the maker argument type specified by the implementation and that the declared class type be a width supertype of the implementation class type. The other antecedent of this rule is that the public members in the interface must be a subset of the public members in the implementation. With these conditions met, we can infer an environment that extends Γ with a map from C to the declared class interface.

The most complex rule in the static semantics describes the typing for a class expression that inherits from some other class:

$$\frac{\{\} \vdash \mathit{ty} \triangleright \tau \quad (\mathcal{C}\ \mathrm{of}\ \Gamma)(\mathsf{B}) = (\tau_{\mathsf{B}}, \theta_{\mathsf{B}}, \mathcal{P}_{\mathsf{B}}) \quad \Gamma \vdash \mathit{exp} \triangleright \tau' \quad \mathcal{S}_0 \vdash \tau' <: \tau_{\mathsf{B}} \\ \Gamma \pm \{\mathsf{x} \mapsto \tau\} \vdash \mathit{fld} \triangleright (\theta_f, \mathcal{P}_f) \\ \Gamma \pm \{\texttt{super} \mapsto \mathbf{obj}\,\alpha.\theta_{\mathsf{B}}, \texttt{self} \mapsto \mathbf{obj}\,\alpha.\theta\} \vdash \mathit{meth} \triangleright (\theta_m, \mathcal{P}_m) \\ \theta = \theta_{\mathsf{B}} \oplus \theta_f \oplus \theta_m}{\Gamma \vdash (\mathsf{x}\ \texttt{:}\ \mathit{ty})\ \{\ \texttt{inherits}\ \mathsf{B}(\mathit{exp})\ \mathit{fld}\ \mathit{meth}\ \} \triangleright (\tau, \theta, \mathcal{P}_f \cup \mathcal{P}_m)}$$

This rule defines the resulting class type θ to be the type of the base class B (θ_{B}), extended with the types of the newly defined fields (θ_f) and methods (θ_m). When typechecking method bodies, we give **`super`** type $\mathbf{obj}\,\alpha.\theta_{\mathsf{B}}$, which is the object type that describes the class-view of B. We give **`self`** type $\mathbf{obj}\,\alpha.\theta$, the object type formed from θ, because it describes the class view of the newly defined class. Maker parameter x is added to the environment when checking fields but not methods because MOBY permits only fields to refer to maker parameters.

An important aspect of this rule is that the typing of the class expression depends only on the interface of its superclass, *i.e.*, on $(\tau_{\mathsf{B}}, \theta_{\mathsf{B}}, \mathcal{P}_{\mathsf{B}})$. This interface contains no information about the private members of B or about B's superclasses (if any). Consequently, deriving classes may be compiled independently of their base-class implementations. In other words, MOBY avoids the *fragile base-class problem*. Another important observation is that the object-view of the class B (specified by $\mathcal{P}_{\mathsf{B}}$) does not play any role in this rule: MOBY allows the deriving class complete flexibility in controlling the visibility of inherited fields and methods.[4] This flexibility increases the expressiveness of package scope.

The rules for method and field definitions are straightforward. We examine two of the method definition rules here; the rest are analogous and can be found in Appendix B. The rule for a public method declaration is similar to the standard rule for functions:

$$\frac{\Gamma \vdash \mathit{ty'} \triangleright \tau' \quad \Gamma \vdash \mathit{ty} \triangleright \tau \quad \Gamma \pm \{\mathsf{x} \mapsto \tau'\} \vdash \mathit{exp} \triangleright \tau'' \quad \mathcal{S}_0 \vdash \tau'' <: \tau}{\Gamma \vdash \texttt{public meth}\ \mathsf{m}\ \texttt{(}\mathsf{x}\ \texttt{:}\ \mathit{ty'}\texttt{)}\ \texttt{:}\ \mathit{ty}\ \texttt{=>}\ \mathit{exp} \triangleright (\{\!\!|\, \text{—};\ \mathsf{m} : \tau' \to \tau \,|\!\!\}, \{\mathsf{m}\})}$$

The rule for non-public methods is almost the same — the only difference is that the set of public labels in the conclusion is empty. The rule for a sequence of method definitions checks that the two definitions have disjoint label sets:

$$\frac{\Gamma \vdash \mathit{meth} \triangleright (\theta, \mathcal{P}) \quad \Gamma \vdash \mathit{meth'} \triangleright (\theta', \mathcal{P}') \quad \mathrm{dom}(\theta) \pitchfork \mathrm{dom}(\theta') \quad \theta'' = \theta \oplus \theta'}{\Gamma \vdash \mathit{meth}\ \mathit{meth'} \triangleright (\theta'', \mathcal{P} \cup \mathcal{P}')}$$

Note that since θ and θ' have disjoint labels, the extension does not involve overriding.

Finally, we consider two of the rules for expressions. The rule for object creation shows how the set of public labels is used to restrict the result to the object view of the class:

$$\frac{(\mathcal{C}\ \mathrm{of}\ \Gamma)(\mathsf{C}) = (\tau_{\mathsf{C}}, \theta_{\mathsf{C}}, \mathcal{P}_{\mathsf{C}}) \quad \Gamma \vdash \mathit{exp} \triangleright \tau \quad \mathcal{S}_0 \vdash \tau <: \tau_{\mathsf{C}}}{\Gamma \vdash \texttt{new}\ \mathsf{C}\ \texttt{(}\mathit{exp}\texttt{)} \triangleright \mathbf{obj}\,\alpha.\{\!\!|\, \mathsf{l} : \theta_{\mathsf{C}}.\mathsf{l}^{\mathsf{l} \in \mathrm{Labels}(\theta) \cap \mathcal{P}_{\mathsf{C}}} \,|\!\!\}}$$

The rule for object member selection is as follows:

$$\frac{\Gamma \vdash \mathit{exp} \triangleright \mathbf{obj}\,\alpha.\theta \quad \mathsf{l} \in \mathrm{Labels}(\theta)}{\Gamma \vdash \mathit{exp}\,\texttt{.}\,\mathsf{l} \triangleright \theta.\mathsf{l}[\alpha \mapsto \mathbf{obj}\,\alpha.\theta]}$$

Note that we substitute the type $\mathbf{obj}\,\alpha.\theta$ for α in type $\theta.\mathsf{l}$; this substitution has the effect of unrolling the recursion in the object type.

3 MOC

MOC, short for MOBY object calculus, is an extensible-object calculus designed to serve as the internal language for MICROMOBY. MOC has both extensible objects, which we call class values, and non-extensible objects. Classes support adding and overriding fields and methods, but not method dispatch or field selection, while objects are fixed but allow method dispatch, field selection, and imperative update of fields. The MOC **new** operator creates an object from a class value by

[4]Except, of course, for private members (*i.e.*, those hidden by interface ascription).

Programs

p	$::=$	e	expression
	$\mid$	$d\ p$	sequencing

Declarations

d	$::=$	$\texttt{val}\ \mathsf{x}\texttt{=}e$	value binding

Expressions

e	$::=$	c	constant
	$\mid$	x	variable
	$\mid$	$\texttt{fn}\,(\mathsf{x}\,\texttt{:}\,\tau')\,:\,\tau\ \texttt{=>}\ e$	function
	$\mid$	$e(e')$	application
	$\mid$	$(e\ \texttt{<:}\ \tau)$	type restriction
	$\mid$	$\langle\!\vert\ \vert\!\rangle$	null class value
	$\mid$	$e\ \texttt{+=}\ \langle\!\vert \mathsf{f} = e \vert\!\rangle$	field extension/override
	$\mid$	$e'\ \texttt{+=}\ \langle\!\vert \mathsf{m} = \mu_{\mathsf{m}}{}^{\mathsf{m}\in\mathcal{M}} \vert\!\rangle$	method extension/override
	$\mid$	$(e \downarrow \mathcal{P})$	member hiding
	$\mid$	$\texttt{new}\ e$	object creation
	$\mid$	$e\,\texttt{.}\,\mathsf{l}$	field/method selection
	$\mid$	$e\,\texttt{.}\,\mathsf{f}\ \texttt{:=}\ e'$	field update
	$\mid$	$\texttt{self}$	self
	$\mid$	$\texttt{super}\,\texttt{.}\,\mathsf{m}$	super-class method

Methods

μ	$::=$	$\texttt{meth}\,(\mathsf{x}\,\texttt{:}\,\tau')\,\texttt{:}\,\tau\ \texttt{=>}\ e$	method definition

Figure 3: The syntax of MOC

allocating it in the store and initializing its members from the values in the class. Adopting the technique of Riecke and Stone [RS99], we annotate each MOC class with a dictionary to prevent name clashes with private names [Fis96, BF98]. This approach requires us to provide direct support for both **self** and **super**.[5] In this section, we present the syntax of MOC, describe its operational semantics, discuss its type system, and state subject reduction and progress theorems for it.

3.1 Syntax

The syntax of MOC is given in Figure 3, where τ ranges over MOC types, which are described in Section 3.3. A MOC program is a sequence of declarations followed by an expression. Expressions include constants, variables, functions, and function applications, as well as various forms related to classes and objects. Functions are annotated with their argument and return types. The term $(e\ \texttt{<:}\ \tau)$ denotes an *upward* coercion to type τ. Static checks insure that e's type is a subtype of τ, but the expression has no run-time effect.

The next four expression forms involve classes. The expression $\langle\!\vert\ \vert\!\rangle$ denotes the empty class value, from which other classes are created. There are two forms for creating a class from an existing one: the first extends or overrides a field, while the second extends and/or overrides a collection of methods. The expression $e'\ \texttt{+=}\ \langle\!\vert \mathsf{f} = e \vert\!\rangle$ either adds or overrides the field f of class e' with the

[5] We had thought to handle **super** during translation, since all references to methods through **super** can be resolved at method-installation time. But, we realized that **super** has the same problem with private names that **self** has and that consequently we needed a primitive notion of **super** in our calculus.

value of e. If e' already had an f field, then this operation is extension, otherwise it is overriding. The expression form for augmenting a class with additional methods is more complicated, since it allows multiple methods to be installed simultaneously.[6] The expression $(e \downarrow \mathcal{P})$ denotes the class just like e, except that the members of e not mentioned in the label set $\mathcal{P}$ are *hidden*. Such members are *private* in C++ terminology. Existing methods can continue to refer to them, but new methods cannot. Once a label is hidden, it may be reused without overriding the hidden definition. We use Riecke-Stone style *dictionaries* to capture this behavior in our dynamic semantics [RS99].

The remaining expressions involve objects. The first of these, **`new`** e, creates an object from the class value denoted by e. The next two expressions denote member selection and field update, respectively. The last two expression forms may occur only inside method definitions. When the special variable **`self`** occurs in a method, it refers to the object containing the method. The expression **`super.`**m refers to the m method of the super class.

Finally, method bodies are written using the syntax **`meth`**$(\mathsf{x}\texttt{:}\tau')\texttt{:}\tau$ **`=>`** e. As with functions, the argument and return types are part of the method definition.

3.2 Dynamic semantics of MOC

To define the dynamic semantics of MOC, we enrich its syntax with computational terms. We use this enriched syntax and an *object store* to describe the MOC semantics. First, however, we need to define some additional semantic objects:

$$
\begin{array}{rcll}
i, j, k & \in & \textsc{SlotId} & \text{slot IDs} \\
\mathcal{I}, \mathcal{J}, \mathcal{K} & \in & \mathrm{Fin}(\textsc{SlotId}) & \text{slot ID sets} \\
\phi & \in & \textsc{LabId} \stackrel{\text{fin}}{\to} \textsc{SlotId} & \text{dictionary} \\
\rho & \in & \textsc{Addr} & \text{object store address} \\
\{\!| i = v_i{}^{i \in \mathcal{I}};\ j = \mu_j{}^{j \in \mathcal{J}} |\!\} & \in & \textsc{ObjVal} & \text{object value} \\
\sigma & \in & \textsc{Store} = \textsc{Addr} \stackrel{\text{fin}}{\to} \textsc{ObjVal} & \text{object store}
\end{array}
$$

The representation of class and object values are essentially records indexed by *slot IDs* ($\textsc{SlotId}$). Intuitively, one can think of slot IDs as offsets into the representation of an object's data and method suites. We use *dictionaries* to map the symbolic labels that denote class and object members to their assigned slots. We say that a dictionary ϕ is a *slot assignment* for a label set $\mathcal{L}$ if $\mathrm{dom}(\phi) = \mathcal{L}$ and ϕ is one-to-one. In our semantics, object values live in the store and are refered to by addresses in the term language.

Using these definitions, we extend the MOC expressions with three computational forms:

$$
\begin{array}{rcll}
e & ::= & \rho & \text{object address} \\
& | & \{\!| i = v_i{}^{i \in \mathcal{I}};\ j = \mu_j{}^{j \in \mathcal{J}} |\!\}_\phi & \text{class value} \\
& | & (e @ \phi) & \text{object view} \\
& | & \cdots &
\end{array}
$$

where the v_i are *values*, defined by the following sub-grammar of expressions:

[6]This simultaneous extension/override is necessary to support the semantics of **`super`** properly.

$$
\begin{array}{rcl}
v & ::= & \mathtt{c} \\
& | & \mathbf{fn}\,(\mathtt{x} : \tau') : \tau \;\mathtt{=>}\; e \\
& | & \rho \\
& | & \{\!| i = {v_i}^{i \in \mathcal{I}};\ j = {\mu_j}^{j \in \mathcal{J}} |\!\}_\phi \\
& | & (v@\phi)
\end{array}
$$

As noted above, the components of a class value are indexed by slot IDs. We distinguish the field slots (indexed by $\mathcal{I}$) from the method slots (indexed by $\mathcal{J}$) and require that $\mathcal{I}$ and $\mathcal{J}$ are disjoint ($\mathcal{I} \pitchfork \mathcal{J}$). Class values are annotated with a dictionary (ϕ) that maps the visible labels to their slots. By removing a label from the domain of a dictionary, we make it private.

Unlike class values, objects do not have dictionaries built into their representation. Instead, we use the computational term $(e@\phi)$ to associate a dictionary ϕ with the object denoted by e. This separation of object value and dictionary is necessary because a single object value can have different dictionaries associated with it in different contexts.

Following the approach of Harper and Stone [HS97], we define the dynamic semantics of MOC as a relation ($\hookrightarrow$) between *configurations*. A configuration is a triple $\langle \sigma, E, e \rangle$, where σ is a store, E is an evaluation context stack, and e an expression representing the term currently being evaluated. Informally, an evaluation context stack E is a stack of frames, where a frame is a term with a single hole in it. We write the hole as $[\cdot]$ and write $E \circ F$ for the stack with frame F on top. The formal definition of evaluation stacks and frames is given in Appendix D. We write

$$
\langle \sigma, E, e \rangle \quad \hookrightarrow \quad \langle \sigma', E', e' \rangle
$$

to indicate that the configuration $\langle \sigma, E, e \rangle$ evaluates in one step to configuration $\langle \sigma', E', e' \rangle$. The evaluation relation is defined by the reduction rules given in Appendix D.

In the remainder of this section, we discuss selected evaluation rules. The evaluation rules come in two flavors: rules to find the next redux, which have the effect of specifying the order of evaluation, and rules to perform computation. We start with the reduction rules related to function evaluation to illustrate the form of the rules:

$$
\begin{array}{rcl}
\langle \sigma, E, e_1(e_2) \rangle & \hookrightarrow & \langle \sigma, E \circ ([\cdot](e_2)), e_1 \rangle \\
\langle \sigma, E \circ ([\cdot](e)), v \rangle & \hookrightarrow & \langle \sigma, E \circ (v([\cdot])), e \rangle \\
\langle \sigma, E \circ (\mathbf{fn}\,(\mathtt{x} : \tau) : \tau' \;\mathtt{=>}\; e)([\cdot]), v \rangle & \hookrightarrow & \langle \sigma, E, e[\mathtt{x} \mapsto v] \rangle \\
\langle \sigma, E \circ (\mathtt{c}([\cdot])), v \rangle & \hookrightarrow & \langle \sigma, E, \delta(\mathtt{c}, v) \rangle
\end{array}
$$

The first two rules specify the order of evaluation in functions: first evaluate the function and then the argument. The first rule can be read as saying that with any store σ and with any evaluation context stack E, if the expression to evaluate is a function application, then evaluation starts with the function expression e_1. The rule indicates this choice by pushing the context $([\cdot](e_2))$ onto the stack and making e_1 be the current expression to evaluate. The second of these rules says that once the function position of an application has been reduced to a value, we evaluate the argument position. The third and fourth rules specify the reduction for application. In the case where the function is a lambda term, we apply β-reduction, and in the case where the function is a function constant $\mathtt{c}$, we use the auxiliary semantic function δ to specify the result.

The reduction rule corresponding to method installation is the most complex of the system:

$$
\begin{array}{rcl}
\langle \sigma, E \circ ([\cdot] \; \mathtt{+=} \; \langle\!|\, \mathsf{m} = \mu_m{}^{m \in \mathcal{M}} \,|\!\rangle), v \rangle & \hookrightarrow & \langle \sigma, E, \{\!|\, i = v_i{}^{i \in \mathcal{I}};\, j = \mu_j{}^{j \in \mathcal{J} \setminus \mathcal{J}'}, k = \mu'_k{}^{k \in \mathcal{J}' \cup \mathcal{K}} \,|\!\}_{\phi''} \rangle \\
& & \text{where} \quad v = \{\!|\, i = v_i{}^{i \in \mathcal{I}};\, j = \mu_j{}^{j \in \mathcal{J}} \,|\!\}_{\phi} \\
& & \qquad \mathcal{J}' = \{\phi(\mathsf{m}) \mid \mathsf{m} \in \mathrm{dom}(\phi) \cap \mathcal{M}\} \\
& & \qquad \phi' \text{ is a slot assignment for } \mathcal{M} \setminus \mathrm{dom}(\phi), \\
& & \qquad \text{with } \mathrm{rng}(\phi') = \mathcal{K} \text{ and } \mathcal{K} \pitchfork \mathcal{I} \cup \mathcal{J} \\
& & \qquad \phi'' = \phi \pm \phi' \\
& & \qquad \mu'_{\phi''(\mathsf{m})} = [\![\mu_{\mathsf{m}}]\!]_{v,\phi''} \text{ for } \mathsf{m} \in \mathcal{M}
\end{array}
$$

The left-hand configuration specifies that we are about to add methods $\langle\!|\, \mathsf{m} = \mu_{\mathsf{m}}{}^{\mathsf{m} \in \mathcal{M}} \,|\!\rangle$ to class value $v = \{\!|\, i = v_i{}^{i \in \mathcal{I}};\, j = \mu_j{}^{j \in \mathcal{J}} \,|\!\}_{\phi}$. The class value $\{\!|\, i = v_i{}^{i \in \mathcal{I}};\, j = \mu_j{}^{j \in \mathcal{J} \setminus \mathcal{J}'}, k = \mu'_k{}^{k \in \mathcal{J}' \cup \mathcal{K}} \,|\!\}_{\phi''}$ results. The fields of the new class value are the same as v's. The methods $j = \mu_j{}^{j \in \mathcal{J} \setminus \mathcal{J}'}$ are those that are inherited unchanged from the old class value. To see this fact, note that $\mathcal{J}'$ defines the indices of methods in the class value v that are overridden by a method in $\mathcal{M}$. Index set $\mathcal{K}$ defines the indices of the new methods. It is taken to be disjoint from the slots previously assigned to fields $\mathcal{I}$ and methods $\mathcal{J}$. Note that set $\mathcal{M} \setminus \mathrm{dom}(\phi)$ names the new methods. We construct the new dictionary ϕ'' by merging the old dictionary ϕ and the slot assignment ϕ'. By construction, the domains of ϕ and ϕ' are disjoint. The final piece of this evaluation rule specifies how to translate method bodies before they are installed: $\mu'_{\phi''(\mathsf{m})} = [\![\mu_{\mathsf{m}}]\!]_{v,\phi''}$. The translation function is defined in Appendix D; its purpose is twofold. First, occurrences of **`self`** within each method body are annotated with the dictionary for the new class ϕ''. This dictionary mediates future references to object members through the **`self`** variable. Second, message sends through **`super`** are inlined by replacing expressions of the form **`super.`**m with the appropriate method body from the old class value v. This process implements the static resolution of **`super`** method dispatch.

The next evaluation rule describes object instantiation:

$$
\begin{array}{rcl}
\langle \sigma, E \circ (\mathbf{new}\, [\cdot]), \{\!|\, i = v_i{}^{i \in \mathcal{I}};\, j = \mu_j{}^{j \in \mathcal{J}} \,|\!\}_{\phi} \rangle & \hookrightarrow & \langle \sigma \pm \{\rho \mapsto \{\!|\, i = v_i{}^{i \in \mathcal{I}};\, j = \mu_j{}^{j \in \mathcal{J}} \,|\!\}\}, E, (\rho @ \phi) \rangle \\
& & \text{where } \rho \notin \mathrm{dom}(\sigma)
\end{array}
$$

To instantiate an object from class value $\{\!|\, i = v_i{}^{i \in \mathcal{I}};\, j = \mu_j{}^{j \in \mathcal{J}} \,|\!\}_{\phi}$, we augment the store σ with a mapping from ρ to the object value $\{\!|\, i = v_i{}^{i \in \mathcal{I}};\, j = \mu_j{}^{j \in \mathcal{J}} \,|\!\}$. We pop the **`new`** evaluation context from the stack and return the value $(\rho @ \phi)$ to be further evaluated as directed by E. By annotating the object address ρ with the dictionary ϕ, we insure that external access to this object will be mediated by the dictionary in force at the time of object instantiation (*i.e.*, ϕ).

The final evaluation rule we consider describes method selection:

$$
\begin{array}{rcl}
\langle \sigma, E \circ ([\cdot]\,.\,\mathsf{m}), (\rho @ \phi) \rangle & \hookrightarrow & \langle \sigma, E, \mathbf{fn}\,(\mathsf{x} : \tau') : \tau \Rightarrow e[\mathbf{self} \mapsto \rho] \rangle \\
& & \text{where} \quad \sigma(\rho) = \{\!|\, i = v_i{}^{i \in \mathcal{I}};\, j = \mu_j{}^{j \in \mathcal{J}} \,|\!\} \\
& & \qquad \mu_{\phi(\mathsf{m})} = \mathbf{meth}\,(\mathsf{x} : \tau') : \tau \Rightarrow e
\end{array}
$$

In this reduction rule, we evaluate dispatching method m of the object $(\rho @ \phi)$ by looking up address ρ in store σ and then selecting the method $\mu_{\phi(\mathsf{m})}$ from the resulting object value. The result of the method dispatch is a function value that is formed from the body of $\mu_{\phi(\mathsf{m})}$ by substituting ρ for **`self`**. Note that since $\mu_{\phi(\mathsf{m})}$ is an installed method, all occurrences of **`self`** within e are annotated with the dictionary that was in force at the time that $\mu_{\phi(\mathsf{m})}$ was installed. This property is necessary to get private names to work correctly.

3.3 Static semantics of MOC

Types in MOC are the MICROMOBY types extended with class types θ and *object value types*:

$$
\begin{array}{rcll}
\tau & ::= & \theta & \text{class type} \\
 & | & \{\!\!|\, i : \tau_i{}^{i \in \mathcal{I}};\ j : \tau_j{}^{j \in \mathcal{J}} \,|\!\!\} & \text{object value type} \\
 & | & \cdots &
\end{array}
$$

Note that the type of an object value uses slot IDs for its labels.

The MOC typing system shares well-formedness judgements with MICROMOBY. These judgements appear in Appendix A. The MOC typing judgements are written using value environments ($\mathcal{V}$) and are given in Appendix C. The judgement forms are standard:

$\mathcal{V} \vdash p \triangleright \tau$	Program p has type τ.
$\mathcal{V} \vdash d \triangleright \mathcal{V}'$	Declaration d yields type assumptions $\mathcal{V}'$.
$\mathcal{V} \vdash e \triangleright \tau$	Expression e has type τ.

Here we discuss the rule for method extension, as it is the most complex in the system.

$$
\frac{\begin{array}{c}\mathcal{V} \vdash e' \triangleright \theta' \quad \theta = \theta' \oplus \{\!\!|\, \mathsf{m} : \tau_{\mathsf{m}}{}^{\mathsf{m} \in \mathcal{M}} \,|\!\!\} \\ \mathcal{V} \pm \{\mathbf{self} \mapsto \mathbf{obj}\,\alpha.\theta, \mathbf{super} \mapsto \mathbf{obj}\,\alpha.\theta'\} \vdash \mu_{\mathsf{m}} \triangleright \tau_{\mathsf{m}}\ (\forall \mathsf{m} \in \mathcal{M})\end{array}}{\mathcal{V} \vdash e' \mathrel{\mathbf{+=}} \langle\!| \mathsf{m} = \mu_{\mathsf{m}}{}^{\mathsf{m} \in \mathcal{M}} |\!\rangle \triangleright \theta}
$$

This rule infers that adding methods $\langle\!| \mathsf{m} = \mu_{\mathsf{m}}{}^{\mathsf{m} \in \mathcal{M}} |\!\rangle$ to class e' produces a class with type θ. To reach this conclusion, we must first show that e' has type θ'. We then compute the new class type θ by extending θ' with $\{\!\!|\, \mathsf{m} : \tau_{\mathsf{m}}{}^{\mathsf{m} \in \mathcal{M}} \,|\!\!\}$ via the $\oplus$ operator. Next, for each method m in $\mathcal{M}$ we type check the method body μ_{m} under an environment augmented with typing assumptions about **`self`** and **`super`**. In particular, we assume that **`self`** has type $\mathbf{obj}\,\alpha.\theta$, the object type corresponding to class type θ. Every object instantiated from (an extension of) a class with type θ can be given this type. Similarly, we assign **`super`** type $\mathbf{obj}\,\alpha.\theta'$ as it represents the view of the object defined by the superclass e'.

3.4 Type soundness of MOC

To prove type soundness via a subject reduction theorem, we must provide types and typing rules for the computational terms we added to express MOC evaluation. To type object addresses, we define a store type, which assigns types to addresses:

$$\Sigma \quad \in \quad \textsc{StoreType} = \textsc{Addr} \xrightarrow{\text{fin}} \textsc{Type}$$

Next, we augment the typing judgements for programs, declarations, and expressions from the previous section with a store typing Σ:

$\Sigma, \mathcal{V} \vdash p \triangleright \tau$	Program p has type τ under environment $\mathcal{V}$ and store type Σ.
$\Sigma, \mathcal{V} \vdash d \triangleright \mathcal{V}'$	Declaration d yields type assumptions $\mathcal{V}'$ under $\mathcal{V}$ and store type Σ.
$\Sigma, \mathcal{V} \vdash e \triangleright \tau$	Expression e has type τ under environment $\mathcal{V}$ and store type Σ.

The typing rules in Appendix C are written in this form. We briefly discuss some of the more interesting typing rules for computational terms. The complete collection of rules appear in Appendix C. The rule for typing an address ρ is straightforward:

$$\frac{\vdash \Sigma \triangleright \mathbf{Ok} \quad \vdash \mathcal{V} \triangleright \mathbf{Ok} \quad \rho \in \mathrm{dom}(\Sigma)}{\Sigma, \mathcal{V} \vdash \rho \triangleright \Sigma(\rho)}$$

The following typing rule is used to infer types for object values:

$$\frac{\begin{array}{c}\Sigma, \mathcal{V} \vdash v_i \triangleright \tau_i' \ (\forall i \in \mathcal{I}) \quad \mathcal{S}_0 \vdash \tau_i' <: \tau_i \ (\forall i \in \mathcal{I}) \\ \Sigma, \{\mathbf{\texttt{self}} \mapsto \{\!| \, i : {\tau_i}^{i \in \mathcal{I}};\ j : {\tau_j}^{j \in \mathcal{J}} \, |\!\}\} \vdash \mu_j \triangleright \tau_j \ (\forall j \in \mathcal{J})\end{array}}{\Sigma \vdash \{\!| i = {v_i}^{i \in \mathcal{I}};\ j = {\mu_j}^{j \in \mathcal{J}} |\!\} \triangleright \{\!| \, i : {\tau_i}^{i \in \mathcal{I}};\ j : {\tau_j}^{j \in \mathcal{J}} \, |\!\}}$$

To show that object value $\{\!| i = {v_i}^{i \in \mathcal{I}};\ j = {\mu_j}^{j \in \mathcal{J}} |\!\}$ has type $\{\!| \, i : {\tau_i}^{i \in \mathcal{I}};\ j : {\tau_j}^{j \in \mathcal{J}} \, |\!\}$, we first show that for each slot i, the value stored in that slot (v_i) has a type (τ_i') that is a subtype of the type **`self`** assumes for that slot (τ_i). Next, we show that for each method slot j, the method stored in that slot μ_j has the type **`self`** assumed it to have (τ_j). The methods are typed under the assumption that **`self`** has the type of the object value. Note that each of these methods has been installed in an object and that consequently any occurrences of **`super`** in the original methods have been replaced with inlined code.

The typing rule for the view operator ($e@\phi$) specifies how to convert an object value e, which indexes its members via slots, to an object view, which uses labels for indexing:

$$\frac{\begin{array}{c}\Sigma, \mathcal{V} \vdash \mathrm{e} \triangleright \{\!| \, i : {\tau_i}^{i \in \mathcal{I}};\ j : {\tau_j}^{j \in \mathcal{J}} \, |\!\} \\ \mathcal{L} = \mathrm{dom}(\phi) \quad \phi(\mathcal{L}) \subseteq \mathcal{I} \cup \mathcal{J} \\ \{\} \vdash \tau_{\phi(\mathsf{l})} = \tau_{\mathsf{l}}[\alpha \mapsto \mathbf{obj}\, \alpha.\{\!| \, \mathsf{l} : {\tau_{\mathsf{l}}}^{\mathsf{l} \in \mathcal{L}} \, |\!\}] \ (\forall \mathsf{l} \in \mathcal{L})\end{array}}{\Sigma, \mathcal{V} \vdash (e@\phi) \triangleright \mathbf{obj}\, \alpha.\{\!| \, \mathsf{l} : {\tau_{\mathsf{l}}}^{\mathsf{l} \in \mathcal{L}} \, |\!\}}$$

Object view ($e@\phi$) has type $\mathbf{obj}\, \alpha.\{\!| \, \mathsf{l} : {\tau_{\mathsf{l}}}^{\mathsf{l} \in \mathcal{L}} \, |\!\}$ if we first show that e is an object value with type $\{\!| \, i : {\tau_i}^{i \in \mathcal{I}};\ j : {\tau_j}^{j \in \mathcal{J}} \, |\!\}$. From the domain of dictionary ϕ, we extract the set of labels $\mathcal{L}$ provided by the view. We then check that the dictionary maps the labels to a subset of the supplied slots. Finally, we insure that for each label l, the type of the l member in the object view type (τ_{l}) is equal to the type of the corresponding slot ($\tau_{\phi(\mathsf{l})}$). Because the object view type is recursive, we first substitute the full type for type variable α in τ_{l}.

Definition 3 (Store Typing) We say that a store σ is has store typing Σ, written $\Sigma \models \sigma$, if for every $\rho \in \mathrm{dom}(\sigma)$ we have $\Sigma \vdash \sigma(\rho) \triangleright \Sigma(\rho)$.

We check each of the stored values with respect to the type of the entire store in this definition because the store may be recursive.

Definition 4 (Configuration Typing) We say that a configuration $\langle \sigma, E, e \rangle$ yields type τ under store typing Σ, which we write as $\Sigma \vdash \langle \sigma, E, e \rangle \triangleright \tau$, if the store is well-typed and the remaining computation $E[e]$ has type τ:

$$\frac{\Sigma \models \sigma \quad \Sigma, \{\} \vdash E[e] \triangleright \tau}{\Sigma \vdash \langle \sigma, E, e \rangle \triangleright \tau}$$

Definition 5 A configuration of the form $\langle \sigma, [\cdot], v \rangle$ is said to be *terminal*.

An examination of the rules in Appendix D reveals that a terminal configuration is irreducible. With these definitions, we can state the subject reduction and progress theorems for MOC.

Theorem 1 (Subject Reduction) If $\Sigma \vdash \langle \sigma, E, e \rangle \triangleright \tau$ and $\langle \sigma, E, e \rangle \hookrightarrow \langle \sigma', E', e' \rangle$, then there exists a store typing Σ' that extends Σ and a type τ' such that $\Sigma' \vdash \langle \sigma', E', e' \rangle \triangleright \tau'$ where $\mathcal{S}_0 \vdash \tau' <: \tau$.

The proof of this theorem proceeds by a case analysis of the form of e.

Theorem 2 (Progress) If $\Sigma \vdash \langle \sigma, E, e \rangle \triangleright \tau$, then either $\langle \sigma, E, e \rangle$ is terminal or there exists a configuration $\langle \sigma', E', e' \rangle$ such that $\langle \sigma, E, e \rangle \hookrightarrow \langle \sigma', E', e' \rangle$.

The proof of this theorem proceeds by a case analysis on the form of the configuration $\langle \sigma, E, e \rangle$.

Definition 6 A MOC program p is *well-typed* if we may derive the judgement $\{\}, \{\} \vdash p \triangleright \tau$ for some type τ. We also say in this case that p is *well-typed with type* τ.

Definition 7 A well-typed MOC program p is *type sound* if the configuration $\langle \{\}, [\cdot], p \rangle$ evaluates infinitely or else reduces to a terminal configuration $\langle \sigma', [\cdot], v \rangle$ with the property that there exists a store type Σ and a type τ' such that we may derive the judgements $\Sigma \vdash \langle \sigma', [\cdot], v \rangle \triangleright \tau'$ and $\mathcal{S}_0 \vdash \tau' <: \tau$, where τ is the type of p.

Theorem 3 (Type Soundness for MOC) If MOC program p is well-typed, then p is type sound.

This result follows from Theorems 1 and 2.

4 Translating MICROMOBY to MOC

Now that we have shown that MOC is type sound, we can define a type-preserving translation from MICROMOBY to MOC. We give the translation by augmenting the typing rule for each MICROMOBY syntactic form with the translation for that form. Consequently, the translation is described via a collection of typing/translation judgements:

$\Gamma \vdash prog \triangleright \tau \Rightarrow \overline{prog}$	*prog* has type τ and translates to $\overline{prog}$.
$\Gamma \vdash dcl \triangleright \Gamma' \Rightarrow \overline{dcl}$	*dcl* yields environment Γ' and translates to $\overline{dcl}$.
$\Gamma \vdash clsexp \triangleright (\tau, \theta, \mathcal{P}) \Rightarrow \overline{exp}$	*clsexp* has type $(\tau, \theta, \mathcal{P})$ and translates to $\overline{exp}$.
$\Gamma \vdash fld \triangleright (\theta, \mathcal{P}) \Rightarrow \overline{fld}[\cdot]$	*fld* has type $(\theta, \mathcal{P})$ and translates to $\overline{fld}[\cdot]$.
$\Gamma \vdash meth \triangleright (\theta, \mathcal{P}) \Rightarrow \overline{meth}$	*meth* has type $(\theta, \mathcal{P})$ and translates to $\overline{meth}$.
$\Gamma \vdash exp \triangleright \tau \Rightarrow \overline{exp}$	*exp* has type τ and translates to $\overline{exp}$.

In all but the *fld* translation judgement, the form after the double arrow is a MOC term. In the *fld* case, it is is a MOC context; we discuss the reason for this difference when we discuss *clsexp* translation. By convention, we write the translation of a MICROMOBY term *term* as $\overline{term}$. The full collection of rules for inferring these judgements appears in Appendix B. We briefly discuss some of the more interesting cases here.

Each MICROMOBY class declaration is translated to a MOC function declaration:

$$\frac{\begin{array}{c}\Gamma \vdash clsexp \triangleright (\tau', \theta', \mathcal{P}') \Rightarrow \overline{exp} \quad \vdash interface \triangleright (\tau, \theta, \mathcal{P}) \\ \mathcal{S}_0 \vdash \tau <: \tau' \quad \theta' <:_w \theta \quad \mathcal{P} \subseteq \mathcal{P}'\end{array}}{\begin{array}{l}\Gamma \vdash \mathbf{class}\ \mathsf{C}\ \mathbf{:}\ interface\ \mathbf{=}\ clsexp \triangleright \Gamma \pm \{\mathsf{C} \mapsto (\tau, \theta, \mathcal{P})\} \\ \qquad \Rightarrow \mathbf{val}\ \overline{\mathsf{C}}\ \mathbf{=}\ \mathbf{fn}\,(\mathsf{x}\,\mathbf{:}\,\tau) : \theta\ \mathbf{=>}\ ((\overline{exp})(\mathsf{x}) \downarrow \mathrm{Labels}(\theta))\end{array}}$$

We write $\overline{\mathsf{C}}$ to denote a fresh value identifier that is generated from the class name C. To translate a class declaration, we first translate the body of the class, *clsexp*, to produce the MOC expression $\overline{exp}$. This expression corresponds to the maker function for the class expression; it takes an argument of type τ' and returns a MOC class expression with type θ'. The typing judgement for the class interface gives us $(\tau, \theta, \mathcal{P})$, which is the exported interface of the class. Using these pieces, we construct a MOC function that wraps the function $\overline{exp}$ to produce a maker function with the type required by the declaration. In particular, the new function takes an argument of type τ and returns a class value with type θ. The body of the new function applies $\overline{exp}$ to x and restricts the resulting class value to the set of member labels exposed by θ.

There are two rules for translating class expressions, one for base classes and one for subclasses. We present the subclass rule, as it is more interesting:

$$\frac{\begin{array}{c}\{\} \vdash ty \triangleright \tau \quad (\mathcal{C}\ \mathrm{of}\ \Gamma)(\mathsf{B}) = (\tau_{\mathsf{B}}, \theta_{\mathsf{B}}, \mathcal{P}_{\mathsf{B}}) \\ \Gamma \vdash exp \triangleright \tau' \Rightarrow \overline{exp} \quad \mathcal{S}_0 \vdash \tau' <: \tau_{\mathsf{B}} \\ \Gamma \pm \{\mathsf{x} \mapsto \tau\} \vdash fld \triangleright (\theta_f, \mathcal{P}_f) \Rightarrow \overline{fld}[\cdot] \\ \Gamma \pm \{\mathbf{super} \mapsto \mathbf{obj}\,\alpha.\theta_{\mathsf{B}}, \mathbf{self} \mapsto \mathbf{obj}\,\alpha.\theta\} \vdash meth \triangleright (\theta_m, \mathcal{P}_m) \Rightarrow \overline{meth} \\ \theta = \theta_{\mathsf{B}} \oplus \theta_f \oplus \theta_m\end{array}}{\begin{array}{l}\Gamma \vdash (\mathsf{x}\ \mathbf{:}\ ty)\ \{\ \mathbf{inherits}\ \mathsf{B}(exp)\ fld\ meth\ \} \\ \qquad \triangleright (\tau, \theta, \mathcal{P}_f \cup \mathcal{P}_m) \Rightarrow \mathbf{fn}\,(\mathsf{x}\ \mathbf{:}\ \tau)\ \mathbf{:}\ \theta\ \mathbf{=>}\ \overline{fld}[\overline{\mathsf{B}}(\overline{exp})]\ \mathbf{+=}\ \overline{meth}\end{array}}$$

The intuition behind this rule is as follows. The variable $\overline{\mathsf{B}}$ is the name of the MOC function for the superclass B. We apply this function to $\overline{exp}$, which is the result of translating the argument to B's maker function. We then extend the resulting class value with the translations of the fields and methods defined in the class expression. The notation for the field extension requires more explanation. Recall that MOC supports aggregate method extension, but only unary field extension.

Consequently, field translation cannot produce an aggregation of MOC fields to add to the base class. Instead, the translation produces $\overline{fld}[\cdot]$, a MOC context of the form $[\cdot]$ **+=** $\langle\!|\, \mathsf{f}_1 = \overline{exp}_1 \,|\!\rangle$**+=** $\ldots$ **+=** $\langle\!|\, \mathsf{f}_n = \overline{exp}_n \,|\!\rangle$. Replacing the hole in this context with $\overline{\mathsf{B}}(\overline{exp})$ has the effect of extending the MOC class value for B with the fields from the new class expression. By extending this value with $\overline{meth}$, we produce a MOC class value corresponding to the MICROMOBY class expression. Finally, we return as the result of the translation a function from maker parameter x to the constructed class value.

As an example of this translation, the MICROMOBY `Point` class from Section 2 translates to the following MOC declaration:

$$
\begin{array}{l}
\textbf{val}\ \texttt{Point} = \\
\quad \textbf{fn}\,(\texttt{z}:\texttt{Int}):\{\!|\, \text{—};\ \texttt{getX}:\texttt{Unit}\to\texttt{Int},\ \texttt{move}:\texttt{Int}\to\texttt{Unit} \,|\!\} \ \textbf{=>}\ (\\
\qquad (\textbf{fn}\,(\texttt{x0}:\texttt{Int}):\{\!|\, \texttt{x}:\texttt{Int};\ \texttt{getX}:\texttt{Unit}\to\texttt{Int},\ \texttt{move}:\texttt{Int}\to\texttt{Unit} \,|\!\} \ \textbf{=>} \\
\qquad\quad \langle\!|\,|\!\rangle \ \ \textbf{+=}\ \ \langle\!|\, \texttt{x}=\texttt{x0} \,|\!\rangle \\
\qquad\quad \ \ \textbf{+=}\ \ \langle\!|\, \texttt{getX}=\textbf{meth}\ \ldots,\ \texttt{move}=\textbf{meth}\ \ldots \,|\!\rangle)(z) \downarrow \{\texttt{getX},\texttt{move}\})
\end{array}
$$

To show that our translation preserves types, we first define a translation from MICROMOBY environments to MOC value environments.

Definition 8 (Environment Translation) If Γ is a MICROMOBY environment, we define its translation $\overline{\Gamma}$ to be $(\mathcal{V}\text{ of }\Gamma) \pm \{\overline{\mathsf{C}} \mapsto (\tau \to \theta) \mid \mathsf{C} \in \text{dom}(\mathcal{C}\text{ of }\Gamma) \text{ and } (\mathcal{C}\text{ of }\Gamma)(\mathsf{C}) = (\tau, \theta, \mathcal{P})\}$.

Lemma 1 (Expression Type Preservation) If we can derive the MICROMOBY translation judgement $\Gamma \vdash exp \rhd \tau \Rightarrow \overline{exp}$, then we can derive the MOC judgement $\overline{\Gamma} \vdash \overline{exp} \rhd \tau$.

The proof is by induction on the derivation of $\Gamma \vdash exp \rhd \tau \Rightarrow \overline{exp}$.

Theorem 4 (Program Type Preservation) If we can derive the judgement $\{\} \vdash prog \rhd \tau \Rightarrow \overline{prog}$, then MOC program $\overline{prog}$ is well-typed with type τ.

The proof follows from Lemma 1 and the definition of environment translation.

Definition 9 A MICROMOBY program *prog* is type-sound if we may derive the translation judgement $\{\} \vdash prog \rhd \tau \Rightarrow \overline{prog}$ and MOC program $\overline{prog}$ is type-sound.

Theorem 5 (Type Soundness for MICROMOBY) If we can derive the MICROMOBY judgement $\{\} \vdash prog \rhd \tau$, then the MICROMOBY program *prog* is type-sound.

This fact follows from Theorems 3 and 4.

5 Extensions

The EL that we use in this paper is necessarily restricted, omitting many of MOBY's object-oriented features. In this section, we discuss how the results of this paper might be extended to handle more of MOBY.

- *Modules.* The most important feature missing from MICROMOBY is the module system. A number of approaches exist for specifying the semantics of ML-style modules [MTHM97, Ler96, HL94, HS97]. The key technical issue that shapes these semantics is accounting for type identity and abstraction. In an unpublished working paper [FR98b], we have explored the the *stamp-based* approach used in the definition of SML [MTHM97]. While this technique works well for specifying EL typing rules, it does not provide a type-theoretic accounting and is not well suited for an IL. Currently, we are investigating an extension of MOC with modules in the style of Harper, Lillibridge, and Stone [HL94, HS97].

- *Final and abstract methods.* Like JAVA, MOBY allows classes to annotate methods as either *final* (cannot be overridden by subclasses) or *abstract* (must be overridden by subclasses). We can handle this feature entirely in the MICROMOBY typechecking rules by enriching the domain of class interfaces (INTERFACE) to track the status of methods.

- *Multiple makers.* In MICROMOBY, a class definition defines a single maker function for creating objects, whereas in MOBY, a class definition may include multiple maker functions. Adding multiple makers to our formal system requires changing both the definition of class interfaces and the translation rules. Class interfaces must be augmented with a *maker environment* that maps maker IDs to their argument types. This environment replaces the first component of the interfaces in this paper. Furthermore, the translation rules must generate a function for each maker (instead of a single function for the whole class). Finally, the EL static semantics must ensure that all fields are initialized and that derived classes always invoke a superclass maker.

- *Initially clauses.* To establish *class-level invariants*, which are properties of the collection of objects instantiated from a class, a MOBY class may include an *initially clause*. Once an object has been created and initialized by a maker function, the initially clauses of the implementing classes are invoked, starting with the clause from the most basic class. These expressions may refer to the newly created object (using **`self`**) and may invoke its methods. We capture this semantics in the translation from the EL to IL. We add a non-public initialization method to each class with the reserved name "`initially`." For a derived class with an initially clause *exp*, we define the `initially` method to be:

  $$\textbf{meth}\ \texttt{()}\ :\ \texttt{Unit}\ \textbf{=>}\ \{\ \textbf{super}\texttt{.initially();}\ \overline{exp}\ \}$$

  For base classes, there is no invocation of the superclass's method. Finally, we change the translation of **`new`** to invoke the `initially` method after object construction.

It is worth noting that, with the exception of modules, MOC is expressive enough to support all of these features without change. Therefore, we only have to modify the proof of Theorem 4 to reestablish type soundness.

6 Related Work

The work related to this paper roughly falls into two categories: previous work on formalizing programming language semantics and previous work on object calculi.

6.1 Language Formalization

Work on language semantics spans the range from small calculi designed to explore specific language mechanisms to complete formal descriptions of real-world languages. Perhaps the best example of the latter is the *Definition of Standard ML* [MTHM97], which gives both the static and dynamic semantics of SML in an operational style. Harper and Stone have put the semantics on a more rigorous foundation in their type-theoretic accounting of the semantics of SML [HS97]. Drossopoulou and Eisenbach have developed a semantics for a significant fragment of JAVA and used this to prove type safety [DE97]. A more modest example of language formalization is the work of Rémy and Vouillon, who have formalized a small language that models many of the features of O'CAML [RV98]. Our work is most similar to Rémy and Vouillon's in its scope (their language is roughly equivalent to MICROMOBY in size), but is more closely related to Harper and Stone's in its technical style, since we both use a two-level approach (*i.e.*, EL to IL translation).

Harper and Stone formalize the semantics of SML by defining an EL that is a stripped-down version of the SML surface language and a type directed translation into a higher-order typed lambda calculus. The main difference between our work and that of Harper and Stone stems from the fact that we are formalizing an object-oriented language and thus have adopted an object calculus for our IL. In addition, their formalism includes a module system; we are actively working on extending MOC with a similar module system. Their work has a much broader scope as well, as it covers all of SML's features. The fact that Harper and Stone have been able to use the two-level approach to specify SML gives us hope that MOBY will be amenable to similar formalization.

In general, we are quite pleased with the two-level approach. The one-level approach has the advantage of requiring fewer formal systems than the two-level, but it ties the formal language very closely to the surface language. Consequently, surface language changes require redesigning the formal language and redoing intricate subject reduction theorems. With the two-level approach, changes can be accommodated via the translation. Only the proof that the translation preserves types needs modification. Another potential advantage of the two-level system is that the IL can serve as a "back-end" to a number of different surface languages. Such internal languages can then be studied independently of the originating surface language.

6.2 Object Calculi

In the past ten years, there has been significant work designing object calculi and exploring their formal properties [AC96, Fis96, Liq98, BF98, RS99]. Although we have drawn on much of this work in the design of MOC, its intended role as an IL for MOBY precluded us from simply using one of the earlier calculi. In particular, we believe that modeling classes properly requires a calculus with support for extension, which eliminates the various calculi of Abadi and Cardelli [AC96]. We also consider depth subtyping on object types to be an important flexibility and thus cannot use the Riecke-Stone calculus [RS99]. Their calculus has a single form of object which supports extension and message sending, but only width subtyping.

As a result of these choices, the MOC calculus fits most closely in the family of object calculi with two forms of object: an extensible form to support inheritance and a fixed form with rich object subtyping [Fis96, Liq98, BF98]. MOC differs from the work in Fisher's thesis [Fis96] and that described in Liquori's survey paper [Liq98] in having an imperative operational semantics, which is crucial for modeling a surface language with imperative fields. Because of the imperative semantics, we felt the additional complexity of the *mytype* mechanisms of [Fis96, Liq98] were not justified.

MOC is most closely related to the calculus of Bono and Fisher [BF98]. Both calculi have two forms of object, an imperative semantics, and no *mytype* mechanism, but there are several significant differences between them. One difference is that the Bono-Fisher calculus does not handle private names properly, whereas MOC does via Riecke-Stone style dictionaries [RS99]. Another difference is that the Bono-Fisher calculus uses subsumption to convert implicitly from extensible objects to fixed objects, whereas MOC has the **`new`** operator. While making extensible objects subtypes of fixed objects captures an interesting insight, specifically marking where object instantiation occurs is more appropriate for an IL. A third difference is that MOC provides support for **`super`**, which requires aggregate method extension. Finally, the Bono-Fisher calculus has a type system based on row polymorphism instead of subtyping, and it does not provide recursive types.

7 Conclusions

In this paper, we define the semantics of the object-oriented portions of the MOBY programming language. This definition involves three steps:

- We formalize the relevant portions of MOBY in MICROMOBY.
- We define an object calculus MOC with an operational semantics.
- We specify how to translate from MICROMOBY to MOC terms.

By establishing type soundness for MOC and showing that the translation from MICROMOBY to MOC preserves types, we prove that the object-oriented fragment of MOBY is type safe.

In the future, we plan to continue our formalization of MOBY. Our current focus is on adding a type-theoretic accounting for modules in the style of Harper, Lillibridge, and Stone [HL94, HS97]. This effort requires extending both MICROMOBY and MOC. Once so equipped, we believe MOC will be rich enough to express all the features of MOBY's class, object, and module mechanisms. Aside from modules, the features we do not currently handle can be dealt with by adding the necessary forms to MICROMOBY and then extending the translation to map these additional forms into existing MOC primitives. Hence we envision MICROMOBY will grow closer to MOBY over time, while MOC (extended with modules) remains constant. We believe the two-level approach will allow us to scale our formalism to richer and richer external languages. With each extension, we will need to redo the translation and prove that it preserves types, but we will not need to touch the delicate subject reduction proof for the underlying calculus.

Acknowledgements

Work on the foundations and design of ML2000 provides the basis for our work on MOBY, and discussions with the group about the first version of MOBY were helpful. David MacQueen and Jon Riecke have also been involved in the MOBY design effort, and have provided useful feedback.

References

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

[BF98] Bono, V. and K. Fisher. An imperative, first-order calculus with object extension. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, July 1998, pp. 462–497.

[DE97] Drossopoulou, S. and S. Eisenbach. Java type-safe — probably. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, July 1997, pp. 389–418.

[Fis96] Fisher, K. *Type Systems for Object-oriented Programming Langauges*. Ph.D. dissertation, Stanford University, August 1996.

[FR98a] Fisher, K. and J. Reppy. The design of a class mechanism for MOBY. *Submitted for publication*, October 1998.

[FR98b] Fisher, K. and J. Reppy. MOBY objects and classes. Available at `http://www.cs.bell-labs.com/~jhr/moby/`, 1998.

[HL94] Harper, R. and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, January 1994, pp. 123–137.

[HS97] Harper, R. and C. Stone. An interpretation of Standard ML in type theory. *Technical Report CMU-CS-97-147*, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 1997.

[Ler94] Leroy, X. Manifest types, modules, and separate compilation. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, January 1994, pp. 109–122.

[Ler96] Leroy, X. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, **6**(5), September 1996, pp. 1–32.

[Liq98] Liquori, L. On object extension. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, July 1998, pp. 499–522.

[Mac84] MacQueen, D. Modules for Standard ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, August 1984, pp. 198–207.

[MTHM97] Milner, R., M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, 1997.

[RS99] Riecke, J. G. and C. Stone. Privacy via subsumption. *Theory and Practice of Object Systems*, 1999. To appear.

[RV98] Rémy, D. and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, **4**(1), 1998, pp. 27–50.

A Judgements on types

Well-formed types

$$\boxed{A \vdash \tau \triangleright \mathbf{Ok}}$$

$$\frac{}{A \vdash \mathsf{T} \triangleright \mathbf{Ok}} \quad (1) \qquad \frac{\alpha \in A}{A \vdash \alpha \triangleright \mathbf{Ok}} \quad (2)$$

$$\frac{A \vdash \tau \triangleright \mathbf{Ok} \quad A \vdash \tau' \triangleright \mathbf{Ok}}{A \vdash \tau \to \tau' \triangleright \mathbf{Ok}} \quad (3) \qquad \frac{\alpha \notin A \quad A \cup \{\alpha\} \vdash \tau_{\mathsf{l}} \triangleright \mathbf{Ok}\ (\forall \mathsf{l} \in \mathcal{L})}{A \vdash \mathbf{obj}\,\alpha.\{\!|\, \mathsf{l} : \tau_{\mathsf{l}}{}^{\mathsf{l} \in \mathcal{L}} \,|\!\} \triangleright \mathbf{Ok}} \quad (4)$$

Well-formed class type

$$\boxed{A \vdash \theta \triangleright \mathbf{Ok}}$$

$$\frac{A \vdash \theta.\mathsf{l} \triangleright \mathbf{Ok}\ (\forall \mathsf{l} \in \mathrm{Labels}(\theta))}{A \vdash \theta \triangleright \mathbf{Ok}} \quad (5)$$

Well-formed class interfaces

$$\boxed{\vdash (\tau, \theta, \mathcal{P}) \triangleright \mathbf{Ok}}$$

$$\frac{\{\} \vdash \tau \triangleright \mathbf{Ok} \quad \{\} \vdash \theta \triangleright \mathbf{Ok} \quad \mathcal{P} \subseteq \mathrm{Labels}(\theta)}{\vdash (\tau, \theta, \mathcal{P}) \triangleright \mathbf{Ok}} \quad (6)$$

Well-formed environments

$$\boxed{\vdash \textit{environment} \triangleright \mathbf{Ok}}$$

$$\frac{\{\} \vdash \mathcal{S}(t) \triangleright \mathbf{Ok}\ (\forall t \in \mathrm{dom}(\mathcal{S}))}{\vdash \mathcal{S} \triangleright \mathbf{Ok}} \quad (7) \qquad \frac{\mathcal{T}(\mathsf{t}) \neq \mathcal{T}(\mathsf{t'})\ (\forall \mathsf{t}, \mathsf{t'} \in \mathrm{dom}(\mathcal{T}))}{\vdash \mathcal{T} \triangleright \mathbf{Ok}} \quad (8)$$

$$\frac{\{\} \vdash \mathcal{V}(\mathsf{x}) \triangleright \mathbf{Ok}\ (\forall \mathsf{x} \in \mathrm{dom}(\mathcal{V}))}{\vdash \mathcal{V} \triangleright \mathbf{Ok}} \quad (9) \qquad \frac{\vdash \mathcal{C}(\mathsf{C}) \triangleright \mathbf{Ok}\ (\forall \mathsf{C} \in \mathrm{dom}(\mathcal{C}))}{\vdash \mathcal{C} \triangleright \mathbf{Ok}} \quad (10)$$

$$\frac{\vdash \mathcal{V} \text{ of } \Gamma \triangleright \mathbf{Ok} \quad \vdash \mathcal{C} \text{ of } \Gamma \triangleright \mathbf{Ok}}{\vdash \Gamma \triangleright \mathbf{Ok}} \quad (11) \qquad \frac{\{\} \vdash \Sigma(\rho) \triangleright \mathbf{Ok}\ (\forall \rho \in \mathrm{dom}(\Sigma))}{\vdash \Sigma \triangleright \mathbf{Ok}} \quad (12)$$

Type equality

$$\boxed{\mathcal{E} \vdash \tau = \tau'}$$

$$\frac{}{\mathcal{E} \vdash \tau = \tau} \quad (13) \qquad \frac{\mathcal{E} \vdash \tau' = \tau}{\mathcal{E} \vdash \tau = \tau'} \quad (14)$$

$$\frac{\mathcal{E} \vdash \tau = \tau' \quad \mathcal{E} \vdash \tau' = \tau''}{\mathcal{E} \vdash \tau = \tau''} \quad (15) \qquad \frac{(\alpha, \alpha') \in \mathcal{E}}{\mathcal{E} \vdash \alpha = \alpha'} \quad (16)$$

$$\frac{\mathcal{E} \vdash \tau_1 = \tau_2 \quad \mathcal{E} \vdash \tau_1' = \tau_2'}{\mathcal{E} \vdash \tau_1 \to \tau_1' = \tau_2 \to \tau_2'} \quad (17) \qquad \frac{\mathcal{E} \vdash \tau_1 = \tau_2 \quad \mathcal{E} \vdash \tau_1' = \tau_2'}{\mathcal{E} \vdash \tau_1 \times \tau_1' = \tau_2 \times \tau_2'} \quad (18)$$

$$\frac{\begin{array}{c}\alpha, \alpha' \notin \mathrm{Vars}(\mathcal{E}) \quad \mathcal{E}' = \mathcal{E} \cup \{(\alpha = \alpha')\} \\ \mathcal{E}' \vdash \tau_{\mathsf{l}} = \tau_{\mathsf{l}}' \ (\forall \mathsf{l} \in \mathcal{L}) \quad \mathcal{L} = \mathcal{L}'\end{array}}{\mathcal{E} \vdash \mathbf{obj}\, \alpha. \{\!| \mathsf{l} : \tau_{\mathsf{l}}{}^{\mathsf{l} \in \mathcal{L}} |\!\} = \mathbf{obj}\, \alpha'. \{\!| \mathsf{l} : \tau'_{\mathsf{l}}{}^{\mathsf{l} \in \mathcal{L}'} |\!\}} \quad (19)$$

$$\frac{\mathrm{Vars}(\mathcal{E}) \vdash \mathbf{obj}\, \alpha. \{\!| \mathsf{l} : \tau_{\mathsf{l}}{}^{\mathsf{l} \in \mathcal{L}} |\!\} \triangleright \mathbf{Ok}}{\mathcal{E} \vdash \mathbf{obj}\, \alpha. \{\!| \mathsf{l} : \tau_{\mathsf{l}}{}^{\mathsf{l} \in \mathcal{L}} |\!\} = \mathbf{obj}\, \alpha. \{\!| \mathsf{l} : \tau_{\mathsf{l}}[\alpha \mapsto \mathbf{obj}\, \alpha. \{\!| \mathsf{l} : \tau_{\mathsf{l}}{}^{\mathsf{l} \in \mathcal{L}} |\!\}]^{\mathsf{l} \in \mathcal{L}} |\!\}} \quad (20)$$

Subtyping

$$\boxed{\mathcal{S} \vdash \tau <: \tau'}$$

$$\frac{\vdash \mathcal{S} \triangleright \mathbf{Ok} \quad \{\} \vdash \tau = \tau'}{\mathcal{S} \vdash \tau <: \tau'} \quad (21) \qquad \frac{\mathcal{S} \vdash \tau <: \tau' \quad \mathcal{S} \vdash \tau' <: \tau''}{\mathcal{S} \vdash \tau <: \tau''} \quad (22)$$

$$\frac{\vdash \mathcal{S} \triangleright \mathbf{Ok} \quad \mathsf{T} \in \mathrm{dom}(\mathcal{S})}{\mathcal{S} \vdash \mathsf{T} <: \mathcal{S}(\mathsf{T})} \quad (23) \qquad \frac{\alpha \in \mathrm{dom}(\mathcal{S})}{\mathcal{S} \vdash \alpha <: \mathcal{S}(t)} \quad (24)$$

$$\frac{\mathcal{S} \vdash \tau_2 <: \tau_1 \quad \mathcal{S} \vdash \tau_1' <: \tau_2'}{\mathcal{S} \vdash \tau_1 \to \tau_1' <: \tau_2 \to \tau_2'} \quad (25) \qquad \frac{\mathcal{S} \vdash \tau_1 <: \tau_2 \quad \mathcal{S} \vdash \tau_1' <: \tau_2'}{\mathcal{S} \vdash \tau_1 \times \tau_1' <: \tau_2 \times \tau_2'} \quad (26)$$

$$\frac{\begin{array}{c}\mathcal{F}' \subseteq \mathcal{F} \quad \mathcal{M}' \subseteq \mathcal{M} \\ \alpha, \alpha' \notin \mathrm{Vars}(\mathcal{S}) \quad \mathcal{S}' = \mathcal{S} \pm \{\alpha \mapsto \alpha'\} \\ \{\} \vdash \tau_{\mathsf{f}}[\alpha \mapsto \mathbf{obj}\, \alpha. \{\!| \mathsf{f} : \tau_{\mathsf{f}}{}^{\mathsf{f} \in \mathcal{F}} ; \mathsf{m} : \tau_{\mathsf{m}}{}^{\mathsf{m} \in \mathcal{M}} |\!\}] = \tau_{\mathsf{f}}'[\alpha' \mapsto \mathbf{obj}\, \alpha'. \{\!| \mathsf{f} : \tau'_{\mathsf{f}}{}^{\mathsf{f} \in \mathcal{F}'} ; \mathsf{m} : \tau'_{\mathsf{m}}{}^{\mathsf{m} \in \mathcal{M}'} |\!\}] \ (\forall \mathsf{f} \in \mathcal{F}') \\ \mathcal{S}' \vdash \tau_{\mathsf{m}} <: \tau_{\mathsf{m}}' \ (\forall \mathsf{m} \in \mathcal{M}')\end{array}}{\mathcal{S} \vdash \mathbf{obj}\, \alpha. \{\!| \mathsf{f} : \tau_{\mathsf{f}}{}^{\mathsf{f} \in \mathcal{F}} ; \mathsf{m} : \tau_{\mathsf{m}}{}^{\mathsf{m} \in \mathcal{M}} |\!\} <: \mathbf{obj}\, \alpha'. \{\!| \mathsf{f} : \tau'_{\mathsf{f}}{}^{\mathsf{f} \in \mathcal{F}'} ; \mathsf{m} : \tau'_{\mathsf{m}}{}^{\mathsf{m} \in \mathcal{M}'} |\!\}} \quad (27)$$

Class-type width subtyping

$$\boxed{\theta <:_w \theta'}$$

$$\frac{\{\} \vdash \{\!|\, \mathsf{l} : \tau_{\mathsf{l}}{}^{\mathsf{l}\in\mathcal{L}} \,|\!\} \triangleright \mathbf{Ok} \quad \mathcal{L}' \subseteq \mathcal{L} \qquad \{\} \vdash \tau_{\mathsf{l}} = \tau_{\mathsf{l}}' \ (\forall \mathsf{l} \in \mathcal{L}')}{\{\!|\, \mathsf{l} : \tau_{\mathsf{l}}{}^{\mathsf{l}\in\mathcal{L}} \,|\!\} <:_w \{\!|\, \mathsf{l} : \tau'_{\mathsf{l}}{}^{\mathsf{l}\in\mathcal{L}'} \,|\!\}} \tag{28}$$

B Translating MICROMOBY to MOC

The rules in this appendix specify how to convert from MICROMOBY to MOC at both the type and value levels. The rules are organized according to the grammar in Figure 2. By erasing the $\Rightarrow \overline{term}$ portion of each judgement, we obtain the typing rules for MICROMOBY.

Programs

$$\boxed{\Gamma \vdash prog \triangleright \tau \Rightarrow \overline{prog}}$$

$$\frac{\Gamma \vdash dcl \triangleright \Gamma' \Rightarrow \overline{dcl} \quad \Gamma' \vdash prog \triangleright \tau \Rightarrow \overline{prog}}{\Gamma \vdash dcl\ prog \triangleright \tau \Rightarrow \overline{dcl}\ \overline{prog}} \tag{29}$$

Declarations

$$\boxed{\Gamma \vdash dcl \triangleright \Gamma' \Rightarrow \overline{dcl}}$$

$$\frac{\Gamma \vdash clsexp \triangleright (\tau', \theta', \mathcal{P}') \Rightarrow \overline{exp} \quad \vdash interface \triangleright (\tau, \theta, \mathcal{P}) \qquad \mathcal{S}_0 \vdash \tau <: \tau' \quad \theta' <:_w \theta \quad \mathcal{P} \subseteq \mathcal{P}'}{\Gamma \vdash \mathtt{class}\ \mathsf{C} : interface = clsexp \triangleright \Gamma \pm \{\mathsf{C} \mapsto (\tau, \theta, \mathcal{P})\} \Rightarrow \mathtt{val}\ \overline{\mathsf{C}} = \mathtt{fn}\,(\mathsf{x} : \tau) : \theta \mathtt{=>} ((\overline{exp})(\mathsf{x}) \downarrow \mathrm{Labels}(\theta))} \tag{30}$$

$$\frac{\Gamma \vdash exp \triangleright \tau \Rightarrow \overline{exp}}{\Gamma \vdash \mathtt{val}\ \mathsf{x} = exp \triangleright \Gamma \pm \{\mathsf{x} \mapsto \tau\} \Rightarrow \mathtt{val}\ \mathsf{x} = \overline{exp}} \tag{31}$$

Class expressions

$$\boxed{\Gamma \vdash clsexp \triangleright (\tau, \theta, \mathcal{P}) \Rightarrow \overline{exp}}$$

$$\frac{\vdash \Gamma \triangleright \mathbf{Ok} \quad \mathsf{C} \in \mathrm{dom}(\mathcal{C}\ \mathrm{of}\ \Gamma)}{\Gamma \vdash \mathsf{C} \triangleright (\mathcal{C}\ \mathrm{of}\ \Gamma)(\mathsf{C}) \Rightarrow \overline{\mathsf{C}}} \tag{32}$$

$$\frac{\begin{array}{c}\{\} \vdash \mathit{ty} \triangleright \tau \\ \Gamma \pm \{\mathsf{x} \mapsto \tau\} \vdash \mathit{fld} \triangleright (\theta_f, \mathcal{P}_f) \Rightarrow \overline{\mathit{fld}}[\cdot] \\ \Gamma \pm \{\texttt{self} \mapsto \mathbf{obj}\,\alpha.\theta\} \vdash \mathit{meth} \triangleright (\theta_m, \mathcal{P}_m) \Rightarrow \overline{\mathit{meth}} \\ \theta = \theta_f \oplus \theta_m\end{array}}{\begin{array}{l}\Gamma \vdash (\mathsf{x} \texttt{ : } \mathit{ty})\ \{\ \mathit{fld}\ \mathit{meth}\ \}\ \triangleright (\tau, \theta, \mathcal{P}_f \cup \mathcal{P}_m) \\ \qquad \Rightarrow \texttt{fn}\ (\mathsf{x} \texttt{ : } \tau) \texttt{ : } \theta \texttt{ => } \overline{\mathit{fld}}[\langle\!|\ |\!\rangle] \texttt{ += } \overline{\mathit{meth}}\end{array}} \tag{33}$$

$$\frac{\begin{array}{c}\{\} \vdash \mathit{ty} \triangleright \tau \quad (\mathcal{C} \text{ of } \Gamma)(\mathsf{B}) = (\tau_{\mathsf{B}}, \theta_{\mathsf{B}}, \mathcal{P}_{\mathsf{B}}) \\ \Gamma \vdash \mathit{exp} \triangleright \tau' \Rightarrow \overline{\mathit{exp}} \quad \mathcal{S}_0 \vdash \tau' <: \tau_{\mathsf{B}} \\ \Gamma \pm \{\mathsf{x} \mapsto \tau\} \vdash \mathit{fld} \triangleright (\theta_f, \mathcal{P}_f) \Rightarrow \overline{\mathit{fld}}[\cdot] \\ \Gamma \pm \{\texttt{super} \mapsto \mathbf{obj}\,\alpha.\theta_{\mathsf{B}}, \texttt{self} \mapsto \mathbf{obj}\,\alpha.\theta\} \vdash \mathit{meth} \triangleright (\theta_m, \mathcal{P}_m) \Rightarrow \overline{\mathit{meth}} \\ \theta = \theta_{\mathsf{B}} \oplus \theta_f \oplus \theta_m\end{array}}{\begin{array}{l}\Gamma \vdash (\mathsf{x} \texttt{ : } \mathit{ty})\ \{\ \texttt{inherits}\ \mathsf{B}(\mathit{exp})\ \mathit{fld}\ \mathit{meth}\ \} \\ \qquad \triangleright (\tau, \theta, \mathcal{P}_f \cup \mathcal{P}_m) \Rightarrow \texttt{fn}\ (\mathsf{x} \texttt{ : } \tau) \texttt{ : } \theta \texttt{ => } \overline{\mathit{fld}}[\overline{\mathsf{B}}(\overline{\mathit{exp}})] \texttt{ += } \overline{\mathit{meth}}\end{array}} \tag{34}$$

Field declarations

$$\boxed{\Gamma \vdash \mathit{fld} \triangleright (\theta, \mathcal{P}) \Rightarrow \overline{\mathit{fld}}[\cdot]}$$

$$\frac{\vdash \Gamma \triangleright \mathbf{Ok}}{\Gamma \vdash \epsilon \triangleright (\{\!|\,\text{—}\,|\!\}, \{\}) \Rightarrow [\cdot]} \tag{35}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{fld} \triangleright (\theta, \mathcal{P}) \Rightarrow \overline{\mathit{fld}}[\cdot] \\ \Gamma \vdash \mathit{fld'} \triangleright (\theta', \mathcal{P}') \Rightarrow \overline{\mathit{fld'}}[\cdot] \\ \mathrm{dom}(\theta) \pitchfork \mathrm{dom}(\theta') \quad \theta'' = \theta \oplus \theta'\end{array}}{\begin{array}{l}\Gamma \vdash \mathit{fld}\ \mathit{fld'} \triangleright (\theta'', \mathcal{P} \cup \mathcal{P}') \\ \qquad \Rightarrow \overline{\mathit{fld'}}[\overline{\mathit{fld}}[\cdot]]\end{array}} \tag{36}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{exp} \triangleright \tau' \Rightarrow \overline{\mathit{exp}} \quad \Gamma \vdash \mathit{ty} \triangleright \tau \\ \mathcal{S}_0 \vdash \tau' <: \tau\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{public field}\ \mathsf{f} \texttt{ : } \mathit{ty} \texttt{ = } \mathit{exp} \\ \qquad \triangleright (\{\!|\,\mathsf{f} : \tau;\ \text{—}\,|\!\}, \{\mathsf{f}\}) \Rightarrow [\cdot] \texttt{ += } \langle\!|\mathsf{f} \texttt{ = } \overline{\mathit{exp}}|\!\rangle\end{array}} \tag{37}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{exp} \triangleright \tau' \Rightarrow \overline{\mathit{exp}} \quad \Gamma \vdash \mathit{ty} \triangleright \tau \\ \mathcal{S}_0 \vdash \tau' <: \tau\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{field}\ \mathsf{f} \texttt{ : } \mathit{ty} \texttt{ = } \mathit{exp} \\ \qquad \triangleright (\{\!|\,\mathsf{f} : \tau;\ \text{—}\,|\!\}, \{\}) \Rightarrow [\cdot] \texttt{ += } \langle\!|\mathsf{f} \texttt{ = } \overline{\mathit{exp}}|\!\rangle\end{array}} \tag{38}$$

Method declarations

$$\boxed{\Gamma \vdash \mathit{meth} \triangleright (\theta, \mathcal{P}) \Rightarrow \overline{\mathit{meth}}}$$

$$\frac{\vdash \Gamma \triangleright \mathbf{Ok}}{\Gamma \vdash \epsilon \triangleright (\{\!|\,\text{—}\,|\!\}, \{\}) \Rightarrow \langle\!|\ |\!\rangle} \tag{39}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{meth} \triangleright (\theta, \mathcal{P}) \Rightarrow \langle\!|\mathsf{m} \texttt{ = } \mu_{\mathsf{m}}{}^{\mathsf{m} \in \mathcal{M}}|\!\rangle \\ \Gamma \vdash \mathit{meth'} \triangleright (\theta', \mathcal{P}') \Rightarrow \langle\!|\mathsf{m} \texttt{ = } \mu_{\mathsf{m}}{}^{\mathsf{m} \in \mathcal{M}'}|\!\rangle \\ \mathrm{dom}(\theta) \pitchfork \mathrm{dom}(\theta') \quad \theta'' = \theta \oplus \theta'\end{array}}{\begin{array}{l}\Gamma \vdash \mathit{meth}\ \mathit{meth'} \triangleright (\theta'', \mathcal{P} \cup \mathcal{P}') \\ \qquad \Rightarrow \langle\!|\mathsf{m} \texttt{ = } \mu_{\mathsf{m}}{}^{\mathsf{m} \in (\mathcal{M} \cup \mathcal{M}')}|\!\rangle\end{array}} \tag{40}$$

$$\frac{\Gamma \vdash ty' \triangleright \tau' \quad \Gamma \vdash ty \triangleright \tau \quad \Gamma \pm \{\mathsf{x} \mapsto \tau'\} \vdash exp \triangleright \tau'' \Rightarrow \overline{exp} \quad \mathcal{S}_0 \vdash \tau'' <: \tau}{\Gamma \vdash \mathbf{public\ meth}\ \mathsf{m}\ \mathtt{(}\mathsf{x}\ \mathtt{:}\ ty'\mathtt{)}\ \mathtt{:}\ ty\ \mathtt{=>}\ exp \triangleright (\{\!|\, \text{—}; \mathsf{m} : \tau' \to \tau \,|\!\}, \{\mathsf{m}\}) \Rightarrow \langle\!|\, \mathsf{m} = \mathbf{meth}\ \mathtt{(}\mathsf{x}\ \mathtt{:}\ \tau'\mathtt{)}\ \mathtt{:}\ \tau\ \mathtt{=>}\ \overline{exp} \,|\!\rangle} \tag{41}$$

$$\frac{\{\} \vdash ty' \triangleright \tau' \quad \{\} \vdash ty \triangleright \tau \quad \Gamma \pm \{\mathsf{x} \mapsto \tau'\} \vdash exp \triangleright \tau'' \Rightarrow \overline{exp} \quad \mathcal{S}_0 \vdash \tau'' <: \tau}{\Gamma \vdash \mathbf{meth}\ \mathsf{m}\ \mathtt{(}\mathsf{x}\ \mathtt{:}\ ty'\mathtt{)}\ \mathtt{:}\ ty\ \mathtt{=>}\ exp \triangleright (\{\!|\, \text{—}; \mathsf{m} : \tau' \to \tau \,|\!\}, \{\}) \Rightarrow \langle\!|\, \mathsf{m} = \mathbf{meth}\ \mathtt{(}\mathsf{x}\ \mathtt{:}\ \tau'\mathtt{)}\ \mathtt{:}\ \tau\ \mathtt{=>}\ \overline{exp} \,|\!\rangle} \tag{42}$$

Expressions

$\boxed{\Gamma \vdash exp \triangleright \tau \Rightarrow \overline{exp}}$

$$\frac{\vdash \Gamma \triangleright \mathbf{Ok} \quad (\mathcal{V}\ \text{of}\ \Gamma)(\mathsf{x}) = \tau}{\Gamma \vdash \mathsf{x} \triangleright \tau \Rightarrow \mathsf{x}} \tag{43}$$

$$\frac{\vdash \Gamma \triangleright \mathbf{Ok} \quad \text{TypeOf}(\mathsf{c}) = \tau}{\mathcal{V} \vdash \mathsf{c} \triangleright \tau \Rightarrow \mathsf{c}} \tag{44}$$

$$\frac{\Gamma \vdash ty' \triangleright \tau' \quad \Gamma \vdash ty \triangleright \tau \quad \Gamma \pm \{\mathsf{x} \mapsto \tau'\} \vdash exp \triangleright \tau'' \Rightarrow \overline{exp} \quad \mathcal{S}_0 \vdash \tau'' <: \tau}{\Gamma \vdash \mathbf{fn}\ \mathtt{(}\mathsf{x}\ \mathtt{:}\ ty'\mathtt{)}\ \mathtt{:}\ ty\ \mathtt{=>}\ exp \triangleright \tau' \to \tau \Rightarrow \mathbf{fn}\ (\mathsf{x}\ \mathtt{:}\ \tau')\ \mathtt{:}\ \tau\ \mathtt{=>}\ \overline{exp}} \tag{45}$$

$$\frac{\Gamma \vdash exp \triangleright \tau' \to \tau \Rightarrow \overline{exp} \quad \Gamma \vdash exp' \triangleright \tau'' \Rightarrow \overline{exp'} \quad \mathcal{S}_0 \vdash \tau'' <: \tau'}{\Gamma \vdash exp\ \mathtt{(}exp'\mathtt{)} \triangleright \tau \Rightarrow \overline{exp}(\overline{exp'})} \tag{46}$$

$$\frac{(\mathcal{C}\ \text{of}\ \Gamma)(\mathsf{C}) = (\tau_{\mathsf{C}}, \theta_{\mathsf{C}}, \mathcal{P}_{\mathsf{C}}) \quad \Gamma \vdash exp \triangleright \tau \Rightarrow \overline{exp} \quad \mathcal{S}_0 \vdash \tau <: \tau_{\mathsf{C}} \quad \tau' = \mathbf{obj}\,\alpha.\{\!|\, \mathsf{l} : \theta_{\mathsf{C}}.\mathsf{l}^{\mathsf{l} \in \text{Labels}(\theta) \cap \mathcal{P}_{\mathsf{C}}} \,|\!\}}{\Gamma \vdash \mathbf{new}\ \mathsf{C}\ \mathtt{(}\ exp\ \mathtt{)} \triangleright \tau' \Rightarrow (\mathbf{new}\ \overline{\mathsf{C}}\ (\ \overline{exp}\ )\ \mathtt{<:}\ \tau'\ )} \tag{47}$$

$$\frac{\vdash \Gamma \triangleright \mathbf{Ok} \quad (\mathcal{V}\ \text{of}\ \Gamma)(\mathbf{self}) = \tau}{\Gamma \vdash \mathbf{self} \triangleright \tau \Rightarrow \mathbf{self}} \tag{48}$$

$$\frac{\vdash \Gamma \triangleright \mathbf{Ok} \quad (\mathcal{V}\ \text{of}\ \Gamma)(\mathbf{super}) = \mathbf{obj}\,\alpha.\theta \quad \mathsf{m} \in \text{Methods}(\theta)}{\Gamma \vdash \mathbf{super.}\mathsf{m} \triangleright \theta.\mathsf{m}[\alpha \mapsto \mathbf{obj}\,\alpha.\theta] \Rightarrow \mathbf{super.}\mathsf{m}} \tag{49}$$

$$\frac{\Gamma \vdash exp \triangleright \mathbf{obj}\,\alpha.\theta \Rightarrow \overline{exp} \quad \mathsf{l} \in \text{Labels}(\theta)}{\Gamma \vdash exp\mathtt{.}\mathsf{l} \triangleright \theta.\mathsf{l}[\alpha \mapsto \mathbf{obj}\,\alpha.\theta] \Rightarrow \overline{exp}\mathtt{.}\mathsf{l}} \tag{50}$$

$$\frac{\Gamma \vdash exp \triangleright \mathbf{obj}\,\alpha.\theta \Rightarrow \overline{exp} \quad \mathsf{f} \in \text{Fields}(\theta) \quad \Gamma \vdash exp' \triangleright \tau' \Rightarrow \overline{exp'} \quad \mathcal{S}_0 \vdash \tau' <: \theta.\mathsf{f}[\alpha \mapsto \mathbf{obj}\,\alpha.\theta]}{\Gamma \vdash exp\mathtt{.}\mathsf{f}\ \mathtt{:=}\ exp' \triangleright \mathtt{Unit} \Rightarrow \overline{exp}\mathtt{.}\mathsf{f}\ \mathtt{:=}\ \overline{exp'}} \tag{51}$$

Types

$\boxed{\mathcal{T} \vdash ty \triangleright \tau}$

$$\frac{\vdash \mathcal{T} \triangleright \mathbf{Ok}}{\mathcal{T} \vdash \mathsf{T} \triangleright \mathsf{T}} \tag{52}$$

$$\frac{\vdash \mathcal{T} \triangleright \mathbf{Ok} \quad \mathsf{t} \in \text{dom}(\mathcal{T})}{\mathcal{T} \vdash \mathsf{t} \triangleright \mathcal{T}(\mathsf{t})} \tag{53}$$

$$\frac{\mathcal{T} \vdash \mathit{ty} \rhd \tau \quad \mathcal{T} \vdash \mathit{ty'} \rhd \tau'}{\mathcal{T} \vdash \mathit{ty}\ \texttt{->}\ \mathit{ty'} \rhd \tau \to \tau'} \tag{54}$$

$$\frac{\mathcal{T} \vdash \mathit{ty} \rhd \tau \quad \mathcal{T} \vdash \mathit{ty'} \rhd \tau'}{\mathcal{T} \vdash \mathit{ty}\ \texttt{*}\ \mathit{ty'} \rhd \tau \times \tau'} \tag{55}$$

$$\frac{\mathcal{T} \pm \{\mathsf{t} \mapsto \alpha\} \vdash \mathit{memb} \rhd \theta \quad \alpha \notin \mathrm{rng}(\mathcal{T})}{\mathcal{T} \vdash \texttt{obj}\ \mathsf{t}\ \{\ \mathit{memb}\ \} \rhd \mathbf{obj}\,\alpha.\theta} \tag{56}$$

Object-type members

$\boxed{\mathcal{T} \vdash \mathit{memb} \rhd \theta}$

$$\frac{\vdash \mathcal{T} \rhd \mathbf{Ok}}{\mathcal{T} \vdash \epsilon \rhd \{\!|\,-\,|\!\}} \tag{57}$$

$$\frac{\begin{array}{c}\mathcal{T} \vdash \mathit{memb} \rhd \theta \quad \mathcal{T} \vdash \mathit{memb'} \rhd \theta' \\ \mathrm{dom}(\theta) \pitchfork \mathrm{dom}(\theta') \quad \theta'' = \theta \oplus \theta'\end{array}}{\mathcal{T} \vdash \mathit{memb}\ \mathit{memb'} \rhd \theta''} \tag{58}$$

$$\frac{\mathcal{T} \vdash \mathit{ty} \rhd \tau}{\mathcal{T} \vdash \texttt{field}\ \mathsf{f}\ \texttt{:}\ \mathit{ty} \rhd \{\!|\ \mathsf{f} : \tau\ |\!\}} \tag{59}$$

$$\frac{\mathcal{T} \vdash \mathit{ty} \rhd \tau}{\mathcal{T} \vdash \texttt{meth}\ \mathsf{m}\ \texttt{:}\ \mathit{ty} \rhd \{\!|\ \mathsf{m} : \tau\ |\!\}} \tag{60}$$

Interfaces

$\boxed{\vdash \mathit{interface} \rhd (\tau, \theta, \mathcal{P})}$

$$\frac{\{\} \vdash \mathit{ty} \rhd \tau \quad \{\} \vdash \mathit{fldspec} \rhd (\theta_f, \mathcal{P}_f) \quad \{\} \vdash \mathit{methspec} \rhd (\theta_m, \mathcal{P}_m) \quad \theta = \theta_f \oplus \theta_m}{\vdash (\mathit{ty})\ \{\ \mathit{fld}\ \mathit{meth}\ \} \rhd (\tau, \theta, \mathcal{P}_f \cup \mathcal{P}_m)} \tag{61}$$

Field specifications

$\boxed{\vdash \mathit{fldspec} \rhd (\theta, \mathcal{P})}$

$$\frac{}{\vdash \epsilon \rhd (\{\!|\,-\,|\!\}, \{\})} \tag{62}$$

$$\frac{\begin{array}{c}\vdash \mathit{fldspec} \rhd (\theta, \mathcal{P}) \\ \vdash \mathit{fldspec'} \rhd (\theta', \mathcal{P}') \\ \mathrm{dom}(\theta) \pitchfork \mathrm{dom}(\theta') \quad \theta'' = \theta \oplus \theta'\end{array}}{\vdash \mathit{fldspec}\ \mathit{fldspec'} \rhd (\theta'', \mathcal{P} \cup \mathcal{P}')} \tag{63}$$

$$\frac{\{\} \vdash \mathit{ty} \rhd \tau}{\begin{array}{l}\vdash \texttt{public}\ \texttt{field}\ \mathsf{f}\ \texttt{:}\ \mathit{ty} \\ \qquad\qquad \rhd (\{\!|\ \mathsf{f} : \tau\ |\!\}, \{\mathsf{f}\})\end{array}} \tag{64}$$

$$\frac{\{\} \vdash \mathit{ty} \rhd \tau}{\vdash \texttt{field}\ \mathsf{f}\ \texttt{:}\ \mathit{ty} \rhd (\{\!|\ \mathsf{f} : \tau\ |\!\}, \{\})} \tag{65}$$

Method specifications

$\boxed{\vdash \textit{methspec} \triangleright (\theta, \mathcal{P})}$

$$\frac{}{\vdash \epsilon \triangleright (\{\!\!|\, — \,|\!\!\}, \{\})} \quad (66)$$

$$\frac{\vdash \textit{methspec} \triangleright (\theta, \mathcal{P}) \quad \vdash \textit{methspec'} \triangleright (\theta', \mathcal{P}') \quad \mathrm{dom}(\theta) \pitchfork \mathrm{dom}(\theta') \quad \theta'' = \theta \oplus \theta'}{\vdash \textit{methspec methspec'} \triangleright (\theta'', \mathcal{P} \cup \mathcal{P}')} \quad (67)$$

$$\frac{\{\} \vdash \textit{ty} \triangleright \tau}{\vdash \texttt{\textbf{public}}\,\texttt{\textbf{meth}}\ \mathsf{m}\ \texttt{:}\ \textit{ty} \triangleright (\{\!\!|\, \mathsf{m} : \tau \,|\!\!\}, \{\mathsf{m}\})} \quad (68)$$

$$\frac{\{\} \vdash \textit{ty} \triangleright \tau}{\vdash \texttt{\textbf{meth}}\ \mathsf{m}\ \texttt{:}\ \textit{ty} \triangleright (\{\!\!|\, \mathsf{m} : \tau \,|\!\!\}, \{\})} \quad (69)$$

C Static semantics of MOC

The judgements in this appendix define the type system for MOC extended with computational forms. By deleting the rules related to computational forms and removing the store typings Σ from the remaining rules, we obtain the type system for static MOC programs. The rules are organized according to the grammar in Figure 3.

Programs

$\boxed{\Sigma, \mathcal{V} \vdash p \triangleright \tau}$

$$\frac{\Sigma, \mathcal{V} \vdash d \triangleright \mathcal{V}' \quad \Sigma, \mathcal{V}' \vdash p \triangleright \tau}{\Sigma, \mathcal{V} \vdash d\ p \triangleright \tau} \quad (70)$$

Declarations

$\boxed{\Sigma, \mathcal{V} \vdash d \triangleright \mathcal{V}'}$

$$\frac{\Sigma, \mathcal{V} \vdash e \triangleright \tau}{\Sigma, \mathcal{V} \vdash \texttt{\textbf{val}}\ \mathsf{x}\ \texttt{=}\ e \triangleright \mathcal{V} \pm \{\mathsf{x} \mapsto \tau\}} \quad (71)$$

Expressions

$\boxed{\Sigma, \mathcal{V} \vdash e \triangleright \tau}$

$$\frac{\vdash \Sigma \triangleright \mathbf{Ok} \quad \vdash \mathcal{V} \triangleright \mathbf{Ok} \quad \mathrm{TypeOf}(\mathsf{c}) = \tau}{\Sigma, \mathcal{V} \vdash \mathsf{c} \triangleright \tau} \quad (72)$$

$$\frac{\vdash \Sigma \triangleright \mathbf{Ok} \quad \vdash \mathcal{V} \triangleright \mathbf{Ok} \quad \mathsf{x} \in \mathrm{dom}(\mathcal{V})}{\Sigma, \mathcal{V} \vdash \mathsf{x} \triangleright \mathcal{V}(\mathsf{x})} \quad (73)$$

$$\frac{\mathcal{V} \pm \{\mathsf{x} \mapsto \tau'\} \vdash e \triangleright \tau'' \quad \mathcal{S}_0 \vdash \tau'' <: \tau}{\Sigma, \mathcal{V} \vdash \texttt{\textbf{fn}}\ (\mathsf{x} : \tau') : \tau\ \texttt{=>}\ e \triangleright \tau' \to \tau} \quad (74)$$

$$\frac{\Sigma, \mathcal{V} \vdash e \triangleright \tau'' \to \tau \quad \Sigma, \mathcal{V} \vdash e' \triangleright \tau' \quad \mathcal{S}_0 \vdash \tau' <: \tau''}{\Sigma, \mathcal{V} \vdash e(e') \triangleright \tau} \quad (75)$$

$$\frac{\Sigma, \mathcal{V} \vdash e \triangleright \tau' \quad \mathcal{S}_0 \vdash \tau' <: \tau}{\Sigma, \mathcal{V} \vdash (e \texttt{<:}\, \tau) \triangleright \tau} \quad (76) \qquad \frac{\vdash \Sigma \triangleright \mathbf{Ok} \quad \vdash \mathcal{V} \triangleright \mathbf{Ok}}{\Sigma, \mathcal{V} \vdash \langle\!|\,|\!\rangle \triangleright \{\!|\,—\,|\!\}} \quad (77)$$

$$\frac{\begin{array}{c}\Sigma, \mathcal{V} \vdash e' \triangleright \theta' \quad \Sigma, \mathcal{V} \vdash e \triangleright \tau \\ \{\} \vdash \tau <: \tau' \quad \theta = \theta' \oplus \{\!|\, \mathsf{f} : \tau' \,|\!\}\end{array}}{\Sigma, \mathcal{V} \vdash e' \;\mathbf{+=}\; \langle\!| \mathsf{f} = e |\!\rangle \triangleright \theta} \quad (78) \qquad \frac{\Sigma, \mathcal{V} \vdash \mathrm{e} \triangleright \theta \quad \mathcal{P} \subseteq \mathrm{Labels}(\theta)}{\Sigma, \mathcal{V} \vdash (e \downarrow \mathcal{P}) \triangleright \theta \downarrow \mathcal{P}} \quad (79)$$

$$\frac{\begin{array}{c}\Sigma, \mathcal{V} \vdash e' \triangleright \theta' \quad \theta = \theta' \oplus \{\!|\, \mathsf{m} : \tau_{\mathsf{m}}{}^{\mathsf{m} \in \mathcal{M}} \,|\!\} \\ \Sigma, \mathcal{V} \pm \{\texttt{self} \mapsto \mathbf{obj}\,\alpha.\theta, \texttt{super} \mapsto \mathbf{obj}\,\alpha.\theta'\} \vdash \mu_{\mathsf{m}} \triangleright \tau_{\mathsf{m}} \ (\forall \mathsf{m} \in \mathcal{M})\end{array}}{\Sigma, \mathcal{V} \vdash e' \;\mathbf{+=}\; \langle\!| \mathsf{m} = \mu_{\mathsf{m}}{}^{\mathsf{m} \in \mathcal{M}} |\!\rangle \triangleright \theta} \quad (80)$$

$$\frac{\Sigma, \mathcal{V} \vdash e \triangleright \theta}{\Sigma, \mathcal{V} \vdash \texttt{new}\, e \triangleright \mathbf{obj}\,\alpha.\theta} \quad (81) \qquad \frac{\Sigma, \mathcal{V} \vdash e \triangleright \mathbf{obj}\,\alpha.\theta \quad \mathsf{l} \in \mathrm{Labels}(\theta)}{\Sigma, \mathcal{V} \vdash e\,\texttt{.}\,\mathsf{l} \triangleright \theta.\mathsf{l}[\alpha \mapsto \mathbf{obj}\,\alpha.\theta]} \quad (82)$$

$$\frac{\begin{array}{c}\Sigma, \mathcal{V} \vdash e \triangleright \mathbf{obj}\,\alpha.\theta \quad \mathsf{f} \in \mathrm{Fields}(\theta) \\ \Sigma, \mathcal{V} \vdash e' \triangleright \tau' \quad \mathcal{S}_0 \vdash \tau' <: \theta.\mathsf{f}[\alpha \mapsto \mathbf{obj}\,\alpha.\theta]\end{array}}{\Sigma, \mathcal{V} \vdash e\,\texttt{.}\,\mathsf{f} \;\texttt{:=}\; e' \triangleright \texttt{Unit}} \quad (83) \qquad \frac{\begin{array}{c}\vdash \Sigma \triangleright \mathbf{Ok} \quad \vdash \mathcal{V} \triangleright \mathbf{Ok} \\ \mathcal{V}(\texttt{super}) = \mathbf{obj}\,\alpha.\theta \quad \mathsf{m} \in \mathrm{Methods}(\theta)\end{array}}{\Sigma, \mathcal{V} \vdash \texttt{super.}\mathsf{m} \triangleright \theta.\mathsf{m}[\alpha \mapsto \mathbf{obj}\,\alpha.\theta]} \quad (84)$$

We have the following additional rules for typing the computational terms:

$$\frac{\vdash \Sigma \triangleright \mathbf{Ok} \quad \vdash \mathcal{V} \triangleright \mathbf{Ok} \quad \rho \in \mathrm{dom}(\Sigma)}{\Sigma, \mathcal{V} \vdash \rho \triangleright \Sigma(\rho)} \quad (85)$$

$$\frac{\begin{array}{c}\Sigma, \mathcal{V} \vdash \mathrm{e} \triangleright \{\!|\, i : \tau_i{}^{i \in \mathcal{I}};\ j : \tau_j{}^{j \in \mathcal{J}} \,|\!\} \\ \mathcal{L} = \mathrm{dom}(\phi) \quad \phi(\mathcal{L}) \subseteq \mathcal{I} \cup \mathcal{J} \\ \{\} \vdash \tau_{\phi(\mathsf{l})} = \tau_{\mathsf{l}}[\alpha \mapsto \mathbf{obj}\,\alpha.\{\!|\, \mathsf{l} : \tau_{\mathsf{l}}{}^{\mathsf{l} \in \mathcal{L}} \,|\!\}] \ (\forall \mathsf{l} \in \mathcal{L})\end{array}}{\Sigma, \mathcal{V} \vdash (e @ \phi) \triangleright \mathbf{obj}\,\alpha.\{\!|\, \mathsf{l} : \tau_{\mathsf{l}}{}^{\mathsf{l} \in \mathcal{L}} \,|\!\}} \quad (86)$$

$$\frac{\begin{array}{c}\mathcal{F} = \mathrm{dom}(\phi) \cap \textsc{FieldId} \quad \mathcal{M} = \mathrm{dom}(\phi) \cap \textsc{MethId} \\ \Sigma, \mathcal{V} \vdash v_i \triangleright \tau_i' \ (\forall i \in \mathcal{I}) \quad \mathcal{S}_0 \vdash \tau_i' <: \tau_i \ (\forall i \in \mathcal{I}) \\ \Sigma, \{\texttt{self} \mapsto \{\!|\, i : \tau_i{}^{i \in \mathcal{I}};\ j : \tau_j{}^{j \in \mathcal{J}} \,|\!\}\} \vdash \mu_j \triangleright \tau_j \ (\forall j \in \mathcal{J})\end{array}}{\Sigma, \mathcal{V} \vdash \{\!| i = v_i{}^{i \in \mathcal{I}};\ j = \mu_j{}^{j \in \mathcal{J}} |\!\}_\phi \triangleright \{\!|\, \mathsf{f} : \tau_{\phi(\mathsf{f})}{}^{\mathsf{f} \in \mathcal{F}};\ \mathsf{m} : \tau_{\phi(\mathsf{m})}{}^{\mathsf{m} \in \mathcal{M}} \,|\!\}} \quad (87)$$

Methods

$\boxed{\Sigma, \mathcal{V} \vdash \mu \triangleright \tau}$

$$\frac{\Sigma, \mathcal{V} \pm \{\mathsf{x} \mapsto \tau'\} \vdash e \triangleright \tau'' \quad \mathcal{S}_0 \vdash \tau'' <: \tau}{\Sigma, \mathcal{V} \vdash \texttt{meth}\,(\mathsf{x}\ \texttt{:}\ \tau')\ \texttt{:}\ \tau\ \texttt{=>}\ e \triangleright \tau' \to \tau} \quad (88)$$

Object values $\boxed{\Sigma \vdash \{\!|i = {v_i}^{i\in\mathcal{I}};\ j = {\mu_j}^{j\in\mathcal{J}}|\!\} \triangleright \tau}$

$$\frac{\begin{array}{c}\Sigma, \mathcal{V} \vdash v_i \triangleright \tau_i' \ (\forall i \in \mathcal{I}) \quad \mathcal{S}_0 \vdash \tau_i' <: \tau_i \ (\forall i \in \mathcal{I}) \\ \Sigma, \{\mathtt{self} \mapsto \{\!|\, i : {\tau_i}^{i\in\mathcal{I}};\ j : {\tau_j}^{j\in\mathcal{J}}\,|\!\}\} \vdash \mu_j \triangleright \tau_j \ (\forall j \in \mathcal{J})\end{array}}{\Sigma \vdash \{\!|i = {v_i}^{i\in\mathcal{I}};\ j = {\mu_j}^{j\in\mathcal{J}}|\!\} \triangleright \{\!|\, i : {\tau_i}^{i\in\mathcal{I}};\ j : {\tau_j}^{j\in\mathcal{J}}\,|\!\}} \tag{89}$$

Store typing $\boxed{\Sigma \models \sigma}$

$$\frac{\Sigma \vdash \sigma(\rho) \triangleright \Sigma(\rho) \quad (\forall \rho \in \mathrm{dom}(\sigma))}{\Sigma \models \sigma} \tag{90}$$

Configuration typing $\boxed{\Sigma \vdash \langle \sigma, E, e\rangle \triangleright \tau}$

$$\frac{\Sigma \models \sigma \quad \Sigma, \{\} \vdash E[e] \triangleright \tau}{\Sigma \vdash \langle \sigma, E, e\rangle \triangleright \tau} \tag{91}$$

D Dynamic semantics of MOC

Definition 10 A *stack of evaluation contexts* (E) has the following structure:

$$\begin{array}{lll}
E & ::= & [\cdot] \quad | \quad E \circ F \\
F & ::= & \mathtt{val}\ \mathsf{x}\mathtt{=}[\cdot]\ p \\
 & | & [\cdot](e) \quad | \quad v([\cdot]) \\
 & | & ([\cdot]\ \mathtt{<:}\ \tau) \\
 & | & [\cdot]\ \mathtt{+=}\ \langle\!|\mathsf{f}\,\mathtt{=}\,e|\!\rangle \quad | \quad v\ \mathtt{+=}\ \langle\!|\mathsf{f}\,\mathtt{=}\,[\cdot]|\!\rangle \\
 & | & [\cdot]\ \mathtt{+=}\ \langle\!|\mathsf{m} = {\mu_{\mathsf{m}}}^{\mathsf{m}\in\mathcal{M}}|\!\rangle \\
 & | & ([\cdot] \downarrow \mathcal{P}) \\
 & | & \mathtt{new}\ [\cdot] \\
 & | & [\cdot]\,\mathtt{.}\,\mathsf{l} \\
 & | & [\cdot]\,\mathtt{.}\,\mathsf{f}\ \mathtt{:=}\ e' \quad | \quad v\,\mathtt{.}\,\mathsf{f}\ \mathtt{:=}\ [\cdot]
\end{array}$$

where F is called a *frame*. We require that the term formed by composing the frames in a stack be a syntactically valid MOC program.

Definition 11 We define the *installation* of a method into a class value, written $[\![\mu]\!]_{v,\phi}$, as follows:

$$
\begin{array}{rcl}
[\![\mathbf{meth}\,(\mathsf{x}\,\mathbf{:}\,\tau')\,\mathbf{:}\,\tau\;\mathbf{=>}\;e]\!]_{v,\phi} & = & \mathbf{meth}\,(\mathsf{x}\,\mathbf{:}\,\tau')\,\mathbf{:}\,\tau\;\mathbf{=>}\;[\![e]\!]_{v,\phi} \\
[\![\mathsf{c}]\!]_{v,\phi} & = & \mathsf{c} \\
[\![\mathsf{x}]\!]_{v,\phi} & = & \mathsf{x} \\
[\![\mathbf{fn}\,(\mathsf{x}\,\mathbf{:}\,\tau')\,:\,\tau\;\mathbf{=>}\;e]\!]_{v,\phi} & = & \mathbf{fn}\,(\mathsf{x}\,\mathbf{:}\,\tau')\,:\,\tau\;\mathbf{=>}\;[\![e]\!]_{v,\phi} \\
[\![e(e')]\!]_{v,\phi} & = & [\![e]\!]_{v,\phi}([\![e']\!]_{v,\phi}) \\
[\![(e\;\mathbf{<:}\;\tau)]\!]_{v,\phi} & = & ([\![e]\!]_{v,\phi}\;\mathbf{<:}\;\tau) \\
[\![\langle\!|\;|\!\rangle]\!]_{v,\phi} & = & \langle\!|\;|\!\rangle \\
[\![e\;\mathbf{+=}\;\langle\!|\mathsf{f}\,\mathbf{=}\,e'|\!\rangle]\!]_{v,\phi} & = & [\![e]\!]_{v,\phi}\;\mathbf{+=}\;\langle\!|\mathsf{f}\,\mathbf{=}\,[\![e']\!]_{v,\phi}|\!\rangle \\
[\![e\;\mathbf{+=}\;\langle\!|\mathsf{m} = \mu_{\mathsf{m}}{}^{\mathsf{m}\in\mathcal{M}}|\!\rangle]\!]_{v,\phi} & = & [\![e]\!]_{v,\phi}\;\mathbf{+=}\;\langle\!|\mathsf{m} = \mu_{\mathsf{m}}{}^{\mathsf{m}\in\mathcal{M}}|\!\rangle \\
[\![(e \downarrow \mathcal{P})]\!]_{v,\phi} & = & ([\![e]\!]_{v,\phi} \downarrow \mathcal{P}) \\
[\![\mathbf{new}\;e]\!]_{v,\phi} & = & \mathbf{new}\;[\![e]\!]_{v,\phi} \\
[\![e\,\mathbf{.}\,\mathsf{l}]\!]_{v,\phi} & = & [\![e]\!]_{v,\phi}\,\mathbf{.}\,\mathsf{l} \\
[\![e\,\mathbf{.}\,\mathsf{f}\;\mathbf{:=}\,e']\!]_{v,\phi} & = & [\![e]\!]_{v,\phi}\,\mathbf{.}\,\mathsf{f}\;\mathbf{:=}\,[\![e']\!]_{v,\phi} \\
[\![\mathbf{self}]\!]_{v,\phi} & = & (\mathbf{self}@\phi) \\
[\![\mathbf{super}\,\mathbf{.}\,\mathsf{m}]\!]_{\{i=v_i{}^{i\in\mathcal{I}};\,j=\mu_j{}^{j\in\mathcal{J}}\}_{\phi},\phi'} & = & \mathbf{fn}\,(\mathsf{x}\,\mathbf{:}\,\tau')\,:\,\tau\;\mathbf{=>}\;e \quad \text{where } \mu_{\phi(\mathsf{m})} = \mathbf{meth}\,(\mathsf{x}\,\mathbf{:}\,\tau')\,\mathbf{:}\,\tau\;\mathbf{=>}\;e
\end{array}
$$

The dynamic semantics of MOC is defined in terms of two sets of reduction rules. The first set contains *structural* rules that specify how the next redex is chosen. The second set specify how the chosen redux is reduced.

D.1 Structural Rules

$$\langle\sigma, E, e_1(e_2)\rangle \quad\hookrightarrow\quad \langle\sigma, E \circ ([\cdot](e_2)), e_1\rangle \tag{92}$$

$$\langle\sigma, E \circ ([\cdot](e)), v\rangle \quad\hookrightarrow\quad \langle\sigma, E \circ (v([\cdot])), e\rangle \tag{93}$$

$$\langle\sigma, E, e_1\;\mathbf{+=}\;\langle\!|\mathsf{f}\,\mathbf{=}\,e_2|\!\rangle\rangle \quad\hookrightarrow\quad \langle\sigma, E \circ ([\cdot]\;\mathbf{+=}\;\langle\!|\mathsf{f}\,\mathbf{=}\,e_2|\!\rangle), e_1\rangle \tag{94}$$

$$\langle\sigma, E \circ ([\cdot]\;\mathbf{+=}\;\langle\!|\mathsf{f}\,\mathbf{=}\,e|\!\rangle), v\rangle \quad\hookrightarrow\quad \langle\sigma, E \circ (v\;\mathbf{+=}\;\langle\!|\mathsf{f}\,\mathbf{=}\,[\cdot]|\!\rangle), e\rangle \tag{95}$$

$$\langle\sigma, E, e\;\mathbf{+=}\;\langle\!|\mathsf{m}\,\mathbf{=}\,\mu_m{}^{m\in\mathcal{M}}|\!\rangle\rangle \quad\hookrightarrow\quad \langle\sigma, E \circ ([\cdot]\;\mathbf{+=}\;\langle\!|\mathsf{m}\,\mathbf{=}\,\mu_m{}^{m\in\mathcal{M}}|\!\rangle, e\rangle \tag{96}$$

$$\langle\sigma, E, e\,\mathbf{.}\,\mathsf{l}\rangle \quad\hookrightarrow\quad \langle\sigma, E \circ ([\cdot]\,\mathbf{.}\,\mathsf{l}), e\rangle \tag{97}$$

$$\langle\sigma, E, e_1\,\mathbf{.}\,\mathsf{f}\;\mathbf{:=}\,e_2\rangle \quad\hookrightarrow\quad \langle\sigma, E \circ ([\cdot]\,\mathbf{.}\,\mathsf{f}\;\mathbf{:=}\,e_2), e_1\rangle \tag{98}$$

$$\langle\sigma, E \circ ([\cdot]\,\mathbf{.}\,\mathsf{f}\;\mathbf{:=}\,e), v\rangle \quad\hookrightarrow\quad \langle\sigma, E \circ (v\,\mathbf{.}\,\mathsf{f}\;\mathbf{:=}\,[\cdot]), e\rangle \tag{99}$$

$$\langle\sigma, E, \mathbf{new}\;e\rangle \quad\hookrightarrow\quad \langle\sigma, E \circ (\mathbf{new}\;[\cdot]), e\rangle \tag{100}$$

$$\langle\sigma, E, (e \downarrow \mathcal{P})\rangle \quad\hookrightarrow\quad \langle\sigma, E \circ ([\cdot] \downarrow \mathcal{P}), e\rangle \tag{101}$$

$$\langle\sigma, E, (e\;\mathbf{<:}\;\tau)\rangle \quad\hookrightarrow\quad \langle\sigma, E \circ ([\cdot]\;\mathbf{<:}\;\tau), e\rangle \tag{102}$$

D.2 Evaluation Rules

$$\langle \sigma, \mathtt{val}\ \mathsf{x} = [\cdot]\ e, v\rangle \quad \hookrightarrow \quad \langle \sigma, [\cdot], e[\mathsf{x} \mapsto v]\rangle \tag{103}$$

$$\langle \sigma, \mathtt{val}\ \mathsf{x} = [\cdot]\ \mathtt{val}\ \mathsf{y} = e\ p, v\rangle \quad \hookrightarrow \quad \langle \sigma, \mathtt{val}\ \mathsf{y} = [\cdot]\ p[\mathsf{x} \mapsto v], e[\mathsf{x} \mapsto v]\rangle \qquad \text{where } \mathsf{x} \neq \mathsf{y} \tag{104}$$

$$\langle \sigma, E \circ (\mathtt{fn}\,(\mathsf{x} \mathtt{:} \tau) : \tau' \mathtt{=>} e)([\cdot]), v\rangle \quad \hookrightarrow \quad \langle \sigma, E, e[\mathsf{x} \mapsto v]\rangle \tag{105}$$

$$\langle \sigma, E \circ (\mathsf{c}([\cdot])), v\rangle \quad \hookrightarrow \quad \langle \sigma, E, \delta(\mathsf{c}, v)\rangle \tag{106}$$

$$\langle \sigma, E, \langle\!|\ |\!\rangle\rangle \quad \hookrightarrow \quad \langle \sigma, E, \{\!|{-}|\!\}_{\{\}}\rangle \tag{107}$$

$$\begin{aligned}
&\langle \sigma, E \circ (\{\!| i = v_i{}^{i \in \mathcal{I}};\ j = \mu_j{}^{j \in \mathcal{J}} |\!\}_\phi \ \mathtt{+=}\ \langle\!| \mathsf{f} = [\cdot] |\!\rangle), v\rangle \\
&\qquad \hookrightarrow \quad \begin{cases} \langle \sigma, E, \{\!| i = v_i{}^{i \in \mathcal{I} \setminus \{\phi(\mathsf{f})\}}, \phi(\mathsf{f}) = v;\ j = \mu_j{}^{j \in \mathcal{J}} |\!\}_\phi\rangle & \mathsf{f} \in \mathit{dom}(\phi) \\ \langle \sigma, E, \{\!| i = v_i{}^{i \in \mathcal{I}}, k = v;\ j = \mu_j{}^{j \in \mathcal{J}} |\!\}_{\phi \pm \{\mathsf{f} \mapsto k\}}\rangle & \mathsf{f} \notin \mathit{dom}(\phi)\text{, and } k \notin \mathcal{I} \cup \mathcal{J} \end{cases}
\end{aligned} \tag{108}$$

$$\begin{aligned}
&\langle \sigma, E \circ ([\cdot]\ \mathtt{+=}\ \langle\!| \mathsf{m} = \mu_m{}^{m \in \mathcal{M}} |\!\rangle), v\rangle \quad \hookrightarrow \quad \langle \sigma, E, \{\!| i = v_i{}^{i \in \mathcal{I}};\ j = \mu_j{}^{j \in \mathcal{J} \setminus \mathcal{J}'}, k = \mu'_k{}^{k \in \mathcal{J}' \cup \mathcal{K}} |\!\}_{\phi''}\rangle \\
&\qquad \text{where} \quad v = \{\!| i = v_i{}^{i \in \mathcal{I}};\ j = \mu_j{}^{j \in \mathcal{J}} |\!\}_\phi \\
&\qquad\qquad \mathcal{J}' = \{\phi(\mathsf{m}) \mid \mathsf{m} \in \mathrm{dom}(\phi) \cap \mathcal{M}\} \\
&\qquad\qquad \phi' \text{ is a slot assignment for } \mathcal{M} \setminus \mathrm{dom}(\phi), \\
&\qquad\qquad \text{with } \mathrm{rng}(\phi') = \mathcal{K} \text{ and } \mathcal{K} \pitchfork \mathcal{I} \cup \mathcal{J} \\
&\qquad\qquad \phi'' = \phi \pm \phi' \\
&\qquad\qquad \mu'_{\phi''(\mathsf{m})} = [\![\mu_\mathsf{m}]\!]_{v, \phi''} \text{ for } \mathsf{m} \in \mathcal{M}
\end{aligned} \tag{109}$$

$$\begin{aligned}
&\langle \sigma, E \circ ([\cdot]\,\mathtt{.}\,\mathsf{f}), (\rho @ \phi)\rangle \quad \hookrightarrow \quad \langle \sigma, E, v_{\phi(\mathsf{f})}\rangle \\
&\qquad \text{where } \sigma(\rho) = \{\!| i = v_i{}^{i \in \mathcal{I}};\ j = \mu_j{}^{j \in \mathcal{J}} |\!\}
\end{aligned} \tag{110}$$

$$\begin{aligned}
&\langle \sigma, E \circ ([\cdot]\,\mathtt{.}\,\mathsf{m}), (\rho @ \phi)\rangle \quad \hookrightarrow \quad \langle \sigma, E, \mathtt{fn}\,(\mathsf{x} \mathtt{:} \tau') : \tau\ \mathtt{=>}\ e[\mathtt{self} \mapsto \rho]\rangle \\
&\qquad \text{where} \quad \sigma(\rho) = \{\!| i = v_i{}^{i \in \mathcal{I}};\ j = \mu_j{}^{j \in \mathcal{J}} |\!\} \\
&\qquad\qquad \mu_{\phi(\mathsf{m})} = \mathtt{meth}\,(\mathsf{x} \mathtt{:} \tau') \mathtt{:} \tau\ \mathtt{=>}\ e
\end{aligned} \tag{111}$$

$$\begin{aligned}
&\langle \sigma, E \circ ((\rho @ \phi)\,\mathtt{.}\,\mathsf{f} \ \mathtt{:=}\ [\cdot]), v\rangle \quad \hookrightarrow \quad \langle \sigma \pm \{\rho \mapsto \{\!| i = v_i{}^{i \in \mathcal{I} \setminus \{\phi(\mathsf{f})\}}, \phi(\mathsf{f}) = v;\ j = \mu_j{}^{j \in \mathcal{J}} |\!\}\}, E, ()\rangle \\
&\qquad \text{where } \sigma(\rho) = \{\!| i = v_i{}^{i \in \mathcal{I}};\ j = \mu_j{}^{j \in \mathcal{J}} |\!\}
\end{aligned} \tag{112}$$

$$\begin{aligned}
&\langle \sigma, E \circ (\mathtt{new}\,[\cdot]), \{\!| i = v_i{}^{i \in \mathcal{I}};\ j = \mu_j{}^{j \in \mathcal{J}} |\!\}_\phi\rangle \quad \hookrightarrow \quad \langle \sigma \pm \{\rho \mapsto \{\!| i = v_i{}^{i \in \mathcal{I}};\ j = \mu_j{}^{j \in \mathcal{J}} |\!\}\}, E, (\rho @ \phi)\rangle \\
&\qquad \text{where } \rho \notin \mathrm{dom}(\sigma)
\end{aligned} \tag{113}$$

$$\langle \sigma, E \circ ([\cdot] \downarrow \mathcal{P}), \{\!| i = v_i{}^{i \in \mathcal{I}};\ j = \mu_j{}^{j \in \mathcal{J}} |\!\}_\phi\rangle \quad \hookrightarrow \quad \langle \sigma, E, \{\!| i = v_i{}^{i \in \mathcal{I}};\ j = \mu_j{}^{j \in \mathcal{J}} |\!\}_{\phi \downarrow \mathcal{P}}\rangle \tag{114}$$

$$\langle \sigma, E \circ ([\cdot]\ \mathtt{<:}\ \tau), v\rangle \quad \hookrightarrow \quad \langle \sigma, E, v\rangle \tag{115}$$