Inheritance-based subtyping

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

John Reppy
Bell Labs, Lucent Technologies
jhr@research.bell-labs.com

December 9, 1999

1 Introduction

There is a great divide between the study of the foundations of object-oriented languages and the practice of mainstream object-oriented languages like JAVA[AG98] and C++[Str97]. One of the most striking examples of this divide is the rôle that class inheritance plays in defining subtyping relations. In most foundational descriptions of OO languages, and in the language designs that these studies have informed, inheritance does not define any subtyping relation, whereas in languages like JAVA and C++, inheritance defines a subtyping hierarchy. What is interesting about this distinction is that there are certain idioms, such as friend functions and binary methods, that are natural to express in an inheritance-based subtyping framework, but which require substantial complication to handle in a structural subtyping framework.

In this paper, we explore why inheritance-based subtyping relations are useful and present a formal accounting of a small language that supports such subtyping relations. We begin by examining the common object-oriented idiom of *friend functions* and exploring how one might implement this idiom in MOBY [FR99a], which is a language with only structural subtyping. This example illustrates the deficiency of relying solely on structural subtyping in the language design. We then describe an extension to MOBY in Section 3 that adds class types and inheritance-based subtyping to MOBY. We show how this extension supports a number of common idioms, such as friend functions, binary methods, and object cloning. We then present XMOC in Section 4, which is an object calculus that supports both structural and inheritance-based subtyping, as well as privacy. XMOC provides a model of the type system of extended MOBY, and we prove subject-reduction for its type system to validate the design of Extended MOBY. In Section 5 we describe related work and we conclude in Section 6.

2 The problem with friends

Both C++ and JAVA have mechanisms that allow some classes and functions to have greater access privileges to a class's members than others. In C++, a class grants this access by declaring that certain other classes and functions are *friends*. In JAVA, members that are not annotated as **`public`**, **`protected`**, or **`private`** are visible to other classes in the same package, but not to those outside the package. In this section, we examine how to support this idiom in MOBY, a language with structural subtyping and

```
module BagM : {
  type Rep <: Bag
  objtype Bag { meth add : Int -> Unit }
  val union : (Rep, Rep) -> Unit
  val mkBag : Unit -> Rep
} {
  class Bag {
    public field items : var List(Int)
    public meth add (x : Int) -> Unit { self.items := x :: self.items }
    public maker mk () { field items = Nil }
  }
  objtype Rep = typeof(Bag)
  objtype Bag { meth add : Int -> Unit }
  fun union (s1 : Rep, s2 : Rep) -> Unit { List.app s1.add s2.items }
  fun mkBag () -> Rep = new mk()
}
```

Figure 1: Bags and friends using type abstraction

flexible control over class-member visibility [FR99a] (we include a brief description of MOBY in Appendix A). This study demonstrates that while it is possible to encode the friends idiom in a language with only structural type relations, the resulting encoding is not very appealing.

2.1 Friends via partial type abstraction

A standard way to program friends is to use partially abstract types [PT93, KLM94]. For example, Figure 1 gives the MOBY code for an implementation of a `Bag` class that has a `union` function as a friend. In this example, we have ascribed the `BagM` module with a signature that makes the `Rep` type partially abstract to the module's clients. Outside the module, if we have an object of type `Rep`, we can use both the `union` function and the `add` method (since `Rep` is a subtype of `Bag`), but we cannot access the `items` field. Inside the module, the `Rep` type allows access to all of the members of the `Bag` class;[1] the implementation of the `union` function exploits this access.

Unfortunately, this approach only works for *final* classes. If we want to extend the `Bag` class, we must reveal the class in the signature of the `BagM` module (as is done in Figure 2). In this version, an object created using the `mk` maker cannot be used as an argument to the `union` function, because it will not have the `Rep` type. This limitation also applies to objects created from subclasses of `Bag`.

2.2 Friends via representation methods

To support both friends and class extension for the same class requires a public mechanism for mapping from an object to its abstract representation type. With such a mechanism, we can recover the representation type required by the friend functions. For example, suppose we extend our `Bag` class to include a method that returns the number of items in the bag. We call this new class `CBag` (for counting bag), and we want to use the `union` function on objects created from the `CBag` class. Figure 3 presents this new implementation. Notice that we have added a public method `bagRep` to the interface of the `Bag`

[1]The MOBY notation **`typeof`**`(C)` is shorthand for the object type that consists of the public members of class *C*.

```
module BagM : {
  type Rep <: typeof(Bag)
  class Bag {
    public meth add : Int -> Unit
    public maker mk of Unit
  }
  val union : (Rep, Rep) -> Unit
  val mkBag : Unit -> Rep
} {
  ...
}
```

Figure 2: Revealing the `Bag` class

class, which returns **`self`** at the representation type (`Rep`). To apply the `union` function to two bags `b1` and `b2`, we write "`Bag.union (b1.bagRep(), b2.bagRep())`." This expression works even when `b1` and/or `b2` are counting bags. Also note that the `items` field is **`public`** inside the `BagM` module, but is not part of `Bag`'s interface outside the module. This is an example of why objects created from subclasses of `Bag` are not subtypes of `Rep`.

Although this example does not include friends for the `CBag` class, we have included the representation method in its interface, which illustrates the main weakness of this approach. Namely, for each level in the class hierarchy, we must add representation types and methods. These methods pollute the method namespace and, in effect, partially encode the class hierarchy in the object types. Furthermore, it suffers from the source-code version of the *fragile base-class* problem: if we refactor the class hierarchy to add a new intermediate class, we have to add a new representation method, which changes the types of the objects created below that point in the hierarchy. While this encoding approach appears to be adequate for most of the examples that require a strong connection between the implementation and types, it is awkward and unpleasant.

3 Extended MOBY

In the previous section, we showed how we can use abstract representation types and representation methods to tie object types to specific classes. From the programmer's perspective, a more natural approach is to make the classes themselves serve the rôle of types when this connection is needed. In this section, we present an extension of MOBY [FR99a] that supports such *class types* and *inheritance-based subtyping*. Intuitively, an object has a class type **#**`C` if the object was instantiated from `C` or one of its descendants. Inheritance-based subtyping is a form of by-name subtyping that follows the inheritance hierarchy. We illustrate this extension using several examples.

3.1 Adding inheritance-based subtyping

Inheritance-based subtyping requires four additions to MOBY's type system, as well as a couple of changes to the existing rules:

- For any class `C`, we define **#**`C` to be its *class type*, which can be used as a type in any context that

```
module BagM : {
  type Rep <: typeof(Bag)
  class Bag : {
    public meth add : Int -> Unit
    public meth bagRep : Unit -> Rep
    public maker mkBag of Unit
  }
  val union : (Rep, Rep) -> Unit
} {
  class Bag {
    public field items : var List(Int)
    public meth add (x : Int) -> Unit { self.items := x :: self.items }
    public meth bagRep () -> Rep { self }
    public maker mkBag () { field items = Nil }
  }
  objtype Rep = typeof(Bag)
  fun union (s1 : Rep, s2 : Rep) -> Unit {
    List.app s1.add s2.items
  }
}

module CBagM : {
  type Rep <: typeof(CBag)
  class CBag : {
    public meth add : Int -> Unit
    public meth bagRep : Unit -> BagM.Rep
    public meth size : Unit -> Int
    public meth cbagRep : Unit -> Rep
    public maker mkCBag of Unit
  }
} {
  class CBag {
    inherits BagM.Bag
    public field nItems : var Int
    public meth add (x : Int) -> Unit {
      self.nItems := self.nItems+1;
      super.add(x)
    }
    public meth size () -> Int { self.nItems }
    public meth cbagRep () -> Rep { self }
    public maker mkCBag () { super mkBag(); field nItems = 0 }
  }
  objtype Rep = typeof(CBag)
}
```

Figure 3: Bags and friends using representation methods

```
class B {
  public meth m1 () -> Int { ... }
  public meth m2 ...
  ...
}
class C {
  inherits B : { public meth m2  ... }
  public meth m1 () -> Bool { ... }
  maker mkC of Unit { ... }
  ...
}
```

Figure 4: Example of reusing a private method name

is in `C`'s scope. Note that the meaning of a class type depends on its context. Inside a method body, the class type of the host class allows access to all members, whereas outside the class, only the public members can be accessed.

- We extend class interfaces to allow an optional **`inherits`** clause. If in a given context, a class `C` has an interface that includes an "**`inherits`** `B`" clause, then we view **#**`C` as a subtype of **#**`B`. Omitting the **`inherits`** clause from `C`'s interface causes the relationship between `B` and `C` to be hidden.
- We say that **#**`C` is a subtype of **`typeof`**`(C)` (this relation corresponds to Fisher's observation that implementation types are subtypes of interface types [Fis96]).
- The existing typing judgements for method and field selection require the argument to have an object type. We add new judgements for the case where the argument has a class type. We add new rules, instead adding of subtyping to the existing rules, to avoid a technical problem that is described in Section 3.2.
- When typing the methods of a class `C`, we give **`self`** the type **#**`C` (likewise, if `B` is `C`'s superclass, then **`super`** has the type **#**`B`).
- When typing a **`new`** expression, we assign the corresponding class type to the result.

3.2 Inheritance-based subtyping vs. privacy

There is a potential problem in the Extended MOBY type system that has to do with the interaction of inheritance-based subtyping and MOBY's support for privacy. Because MOBY allows signature ascription to hide object members (*e.g.*, the `items` field in Figure 2), **#**`C` can be a subtype of **#**`B` even when **`typeof`**`(C)` is not a subtype of **`typeof`**`(B)`. The problem arises in the case where class `C` has defined a method that has the same name as one of `B`'s private methods. Consider the code fragment in Figure 4, for example.[2] Given these definitions, how do we typecheck the expression: "`(`**`new`**` mkC()).m1()`?" If we allow subtyping on the left-hand side of the method selection, then there are two incompatible ways

[2] This example uses a *class interface* annotation on the class `B`; this syntactic form avoids the need to wrap `B` in a module and signature to hide the `m2` meth.

```
module BagM : {
  class Bag : {
    public meth add : Int -> Unit
    public maker mkBag of Unit
  }
  val union : (#Bag, #Bag) -> Unit
} {
  class Bag {
    public field items : var List(Int)
    public meth add (x : Int) -> Unit { self.items := x :: self.items }
    public maker mkBag () { field items = Nil }
  }
  fun union (s1 : #Bag, s2 : #Bag) -> Unit {
    List.app s1.add s2.items
  }
}

module CBagM : {
  class CBag : {
    inherits BagM.Bag
    public meth size : Unit -> Int
    public maker mkCBag of Unit
  }
} {
  class CBag {
    inherits BagM.Bag
    public field nItems : var Int
    public meth add (x : Int) -> Unit {
      self.nItems := self.nItems+1;
      super.add(x)
    }
    public meth size () -> Int { self.nItems }
    public maker mkCBag () { super mkBag(); field nItems = 0 }
  }
}
```

Figure 5: Bags with friends in Extended MOBY

to typecheck this expression. To avoid this ambiguity, we have different rules for the case where the left-hand side has a class type vs. an object type.[3]

3.3 Friends revisited

We can now revisit our bag class example using the inheritance-based subtyping features of Extended MOBY. In this new implementation (see Figure 5), we use the class type `#Bag` instead of the `Rep` type, which allows us to simplify the code by both eliminating the `Rep` type and the representation method. Note that the interface for the `CBag` class includes an **`inherits`** clause that specifies that it is a subclass of `Bag`. This relation allows the `union` function to be used on values that have the `#CBag` type.

[3]Note that the MOBY typing rules do not include a *subsumption* rule.

```
class B : {
  public meth getX : Unit -> Int
  public meth clone : Unit -> #B
  public maker mkB of Int
  maker copyB of #B
} {
  public meth getX () -> Int { self.pvtX }
  public meth clone () -> #B { new copyB(self) }
  public maker mkB (x : Int) { field pvtX = x }

  field pvtX : Int
  maker copyB (orig : #B) { field pvtX = orig.pvtX }
}

class C {
  inherits B
  public meth clone () -> #C { new copyC(self) }
  public maker mkC (y : Int) { super mkB(y) }
  maker copyC (orig : #C) { super copyB(orig) }
}
```

Figure 6: Cloning with privacy in Extended MOBY

3.4 Binary methods

Binary methods are methods that take another object of the same class as an argument [BCC+96]. There are a number of different flavors of binary methods, depending on how objects from subclasses are treated. Using class types, we can implement binary methods that require access to the private fields of their argument objects. For example, the `union` function in the previous example can be implemented as a binary method as follows:

```
class Bag {
  field items : var List(Int)
  public meth add (x : Int) -> Unit { self.items := x :: self.items }
  public meth union (s : #Bag) -> Unit { List.app self.add s.items }
  public maker mkBag () { field items = Nil }
}
```

3.5 Object cloning

Another case where inheritance-based subtyping is useful is in the typing of *copy constructors*, which can be used to implement a user-defined object cloning mechanism.[4] Figure 6 gives an example of cloning in Extended MOBY. Class `B` has a private field (`pvtX`), which makes object types insufficient to type check `C`'s use of the `copyB` maker function. The problem arises because the object type associated with **`self`** in type-checking `C` does not have a `pvtX` field (because that field is private to `B`), but the `copyB` maker function requires one. Thus, we need the inheritance-based subtyping relationship to allow the `copyC` maker to pass **`self`**, typed with **`#`**`C`, as a parameter to the `copyB` maker. Because we

[4] Note that in MOBY, constructors are called *makers*.

```
signature HAS_SHOW {
  type InitB
  class B : {
    meth show : Unit -> String
    maker mk of InitB
  }
}
module PrintMix (M : HAS_SHOW)
{
  class Pr {
    inherits M.B
    public meth print () -> Unit { ConsoleIO.print(self.show()) }
    maker mk (x : InitB) { super mk(x) }
  }
}

class A {
  public meth show () -> String { "Hi" }
  public meth anotherMeth () -> Unit { ... }
  maker mk () { }
}

module P = PrintMix({type InitB = Unit; class B = A})

class PrA {
  inherits P.Pr
  public meth anotherMeth () -> Unit { (self : #A).anotherMeth() }
}
```

Figure 7: Encoding mixins in Extended MOBY

know that `C` inherits from `B`, this application typechecks. We also exploit this subtyping relation when we override the `clone` method.

3.6 Encoding mixins

MOBY does not support any form of multiple inheritance, but with the combination of parameterized modules and class types, it is possible to encode mixins [BC90, FKF98]. In this encoding, a mixin is implemented as a class parameterized over its base class using a parameterized module. The class interface of the base class contains only those components that are necessary for the mixin. After applying the mixin to a particular base class, we create a new class that inherits from the mixed base class and uses the class types to reconstitute the methods of the base class that were hidden as a result of the module application. Without class types, it would not be possible to make the original class's methods visible again. For example, Figure 7 gives the encoding of a mixin class that adds a `print` method to a class that has a `show` method. After applying `PrintMix` to class `A`, we define a class `PrA` that reconstitutes `A`'s `anotherMeth` method. Notice that we need to use an explicit type constraint to convert the type of **`self`** from **#**`PrA` to **#**`A`, since we do not have subtyping at method dispatch.

While this encoding is cumbersome, it illustrates the power of class types. Also, it might serve as the definition of a derived form that supported mixins directly.

$$
\begin{array}{rcl}
p & ::= & d;\ p \\
 & | & e \\
d & ::= & \texttt{class}\ C\ (x : \tau)\,\{\!|\ \texttt{inherits}\ b;\ m = \mu_m{}^{m \in \mathcal{M}}\ |\!\} \\
 & | & \texttt{class}\ C : \sigma = C' \\
\sigma & ::= & (\tau)\,\{\!|\ \texttt{inherits}\ B;\ m : \tau_m{}^{m \in \mathcal{M}}\ |\!\} \\
\tau & ::= & \alpha \\
 & | & \tau \to \tau' \\
 & | & \texttt{obj}\,\alpha\,\texttt{.}\,\{\!|\ m : \tau_m{}^{m \in \mathcal{M}}\ |\!\} \\
 & | & \texttt{\#}C
\end{array}
\qquad
\begin{array}{rcl}
b & ::= & \mathbf{None} \\
 & | & C(e) \\
\mu & ::= & (x : \tau) \Rightarrow e \\
e & ::= & x \\
 & | & \texttt{fn}(x : \tau) \Rightarrow e \\
 & | & e(e') \\
 & | & \texttt{new}\ C(e) \\
 & | & \texttt{self} \\
 & | & e\,\texttt{.}\,m \\
 & | & e\,@\,C
\end{array}
$$

Figure 8: Syntax of XMOC terms

3.7 Efficiency of method dispatch

Although it is not our main motivation, it is worth noting that method dispatch and field selection from an object with a class type can be implemented easily as a constant time operation. When the dispatched method is final in the class type, the compiler can eliminate the dispatch altogether and call the method directly. In contrast, when an object has an object type, the compiler knows nothing about the layout of the object, making access more expensive. Even when the exact layout of the class is not known because of abstraction (*e.g.*, the mixin encoding from the previous section), we can implement dispatch for objects that have a class type with two memory references and an indirect jump [FRR99].

4 XMOC

We have developed a functional object calculus, called XMOC, that models the type system of Extended MOBY and validates its design. XMOC supports both traditional structural subtyping and inheritance-based subtyping. In this section, we discuss the intuitions behind XMOC and state subject reduction results; space considerations preclude a more detailed presentation. The full system is given in Appendices B and C.

4.1 Syntax

The term syntax of XMOC is given in Figure 8. An XMOC program consists of a sequence of class declarations terminated by an expression. Class declarations come in two forms. In the first, a class C can be declared to *inherit* from a parent class b (when b is $\mathbf{None}$, we say that C is a *base-class*). The class is parameterized by x; whenever an object is constructed from C, x is bound to the supplied initial value. In the second form of class declaration, a class C can be derived from an existing class C' by *class-interface ascription*, which produces a class that inherits its implementation from C', but has the more restrictive class interface σ. A class interface gives the type of the class parameter, the name of the nearest revealed ancestor class (or $\mathbf{None}$), and a typed list of available methods. Types include type variables, function types, recursive object types, and class types.

In a class declaration, we denote the base class either by the special symbol $\mathbf{None}$ or by the name of the base class applied to an argument. Each method (μ) takes a single parameter and has an expression for its body. The syntax of expressions (e) includes variables, functions, function application, new object creation, the special variable **`self`** (only allowed inside method bodies), and method dispatch. The last expression form ($e \,@\, C$) is an *object-view coercion*. Unlike Extended MOBY, XMOC does not map the inheritance relation directly to the subtyping relation; instead we rely on object-view coercions to explicitly coerce the type of an expression from a class to one of its superclasses. This approach avoids the problem discussed in Section 3.2 without requiring two typing judgements for method dispatch. It is possible to automatically insert these coercions into the XMOC representation of a program as part of typechecking (such a translation is similar to the type-directed representation wrapping that has been done for polymorphic languages [Ler92]).

4.2 Dynamic Semantics

Evaluation of an XMOC program occurs in two phases. The first phase is defined by the *class linking* relation, written $\mathcal{K}, p \rightarrowtail \mathcal{K}', p'$, which takes a *dynamic class environment* $\mathcal{K}$ and links the left-most class definition in p to produce $\mathcal{K}'$. Class linking terminates with a residual expression once all of the class declarations have been linked. The second phase evaluates the residual expression to a value (assuming termination). This phase is defined by the expression evaluation relation, which we write as $\mathcal{K} \vdash e \hookrightarrow e'$. Defining the semantics of linking and evaluation requires extending the term syntax with run-time forms.

Correctly handling class-interface ascription provides the greatest challenge in defining the semantics for XMOC. Using this mechanism, a public method m in B can be made private in a subclass C, and subsequently m can be reused to name an unrelated method in some descendant class of C (recall the example in Figure 4). Methods inherited from B must invoke the original m method when they send the m message to **`self`**, while methods defined in D must get the new version. One solution to this problem is to use Riecke-Stone dictionaries in the semantics [RS98, FR99b]. Dictionaries provide the α-conversion needed to avoid capture by mapping method names to *slots*. For XMOC, we use a related technique, which we call *views*. When we process a class C, we tag each method newly defined in C with the label C, using the notation $C :: \mu$. Inherited and overridden methods retain their existing labels. Furthermore, we replace each occurrence of **`self`** in C's new and overridden methods with the object view $\mathit{self} \,@\, C$. Rule 4 in Appendix B describes this annotation formally. At runtime, we represent each object as a pair of a raw object (denoted by meta-variable obj) and a view (denoted by a class name). The raw object contains the list of annotated methods implemented by the object. The view represents the visibility context in which the message send occurs; those methods in scope in class C are available. With this information, we check two conditions when we lookup method m in runtime object $\langle obj\,,\ C\rangle$: first, that m is in the list of methods provided by obj, and second, that C descends from the class annotating the m method. If these two conditions are met, we return the associated method body; otherwise, we search in the portion of the object inherited from its base class. Rules 1 and 2 in Appendix B formally specify method lookup.

4.3 Static semantics

The XMOC typing judgements are written with respect to a static environment Γ, which consists of a set of bound type variables ($\mathcal{A}$), a subtype assumption map ($\mathcal{S}$), a class environment ($\mathcal{C}$), and a variable environment ($\mathcal{V}$). The definition of these environments and the complete set of XMOC typing judgements are given in Appendix C. Here we briefly discuss some of the more important rules.

As mentioned earlier, each XMOC class name doubles as an object type. We associate such a type with an object whenever we instantiate an object from a class, according to the typing rule

$$\frac{(\mathcal{C}\,\mathrm{of}\,\Gamma)(C) = (\tau)\,\{\!\!\{\, \mathtt{inherits}\; B;\; m : {\tau_m}^{m\in\mathcal{M}} \,\}\!\!\} \quad \Gamma \vdash e \triangleright \tau' \quad \Gamma \vdash \tau' <: \tau}{\Gamma \vdash \mathtt{new}\; C(e) \triangleright \mathtt{\#}C}$$

which looks up class C in Γ, infers a type τ' for the constructor argument e, and insures that this type is a subtype of the type of the class parameter τ.

In contexts that allow subtyping, we can treat a class type as an object type according to the following subtyping judgement:

$$\frac{\Gamma \vdash \mathbf{Ok} \quad (\mathcal{C}\,\mathrm{of}\,\Gamma)(C) = (\tau)\,\{\!\!\{\, \mathtt{inherits}\; B;\; m : {\tau_m}^{m\in\mathcal{M}} \,\}\!\!\}}{\Gamma \vdash \mathtt{\#}C <: \mathtt{obj}\,\alpha\,\mathtt{.}\,\{\!\!\{\, m : {\tau_m}^{m\in\mathcal{M}} \,\}\!\!\}}$$

This rule corresponds to the property that `#C` is a subtype of **`typeof(C)`** in Extended MOBY. Note that α cannot occur free in the types τ_m, but the object type $\mathtt{obj}\,\alpha\,\mathtt{.}\,\{\!\!\{\, m : {\tau_m}^{m\in\mathcal{M}} \,\}\!\!\}$ is subject to recursive winding and unwinding.

Unlike Extended MOBY, we do not treat a class type $\mathtt{\#}C$ as being a subtype of its superclass type. Instead we use an object view constraint, which is typed as follows:

$$\frac{\Gamma \vdash e \triangleright \mathtt{\#}C' \quad \Gamma \vdash C' \lessdot C}{\Gamma \vdash e\,@\,C \triangleright \mathtt{\#}C}$$

Because we do not treat inheritance directly as subtyping in XMOC, we only need one rule for typing method dispatch.

$$\frac{\Gamma \vdash e \triangleright \tau \quad \Gamma \vdash \tau <: \mathtt{obj}\,\alpha\,\mathtt{.}\,\{\!\!\{\, m : {\tau_m}^{m\in\mathcal{M}} \,\}\!\!\} \quad m \in \mathcal{M}}{\Gamma \vdash e\,\mathtt{.}\,m \triangleright \tau_m[\alpha \mapsto \mathtt{obj}\,\alpha\,\mathtt{.}\,\{\!\!\{\, m : {\tau_m}^{m\in\mathcal{M}} \,\}\!\!\}]}$$

4.4 Subject reduction

We have proven subject reduction theorems for XMOC. The first states that the linking relation produces a dynamic class environment that is consistent with the static environment defined by the program, and that linking does not change the type of the program.

Theorem 1 If $(\{\}, \{\}, \{\}, \{\}) \vdash p \triangleright \Gamma, \tau$ and $\{\}, p \rightarrowtail^* \mathcal{K}, e$, then $\Gamma \vdash \mathcal{K}$ and $\Gamma, \mathcal{K} \vdash e \triangleright \tau$.

The second theorem states that a well-typed expression evaluations to an expression with a compatible type.

Theorem 2 If $\Gamma \vdash \mathcal{K}$ and $\Gamma, \mathcal{K} \vdash e \triangleright \tau$ and $\mathcal{K} \vdash e \hookrightarrow^* e'$, then $\Gamma, \mathcal{K} \vdash e' \triangleright \tau'$ with $\Gamma \vdash \tau' <: \tau$.

Definition 1 We say that a program p *yields* the value w and $\mathcal{K}$, if there exists an expression e such that $\{\}, p \rightarrowtail^* \mathcal{K}, e$ and $\mathcal{K} \vdash e \hookrightarrow^* w$.

Given this definition, we can characterize the complete evaluation of a program.

Theorem 3 If $(\{\}, \{\}, \{\}, \{\}) \vdash p \triangleright \Gamma, \tau$ and p yields w and $\mathcal{K}$, then $\Gamma, \mathcal{K} \vdash w \triangleright \tau'$ with $\Gamma \vdash \tau' <: \tau$.

5 Related work

Our class types are motivated by the rôle that classes play in languages like C++ and JAVA. The main difference between Extended MOBY and the class types provided by these other languages is in the way that abstraction is supported. Extended MOBY allows partial hiding of inherited components using signature ascription, which means that **typeof**(C) may not be a subtype of **typeof**(B) even when C is known to inherit from B (see Section 3.2). A related mechanism is C++'s *private inheritance*, which allows a subclass to inherit from a base class while hiding the inherited members and concealing the subtyping relationship. Extended MOBY is more flexible, since it allows hiding on a per-member basis. Extended MOBY also allows the class hierarchy to be hidden by omitting the **inherits** clause in class interfaces. In C++ and JAVA the full class hierarchy is manifest in the class types (except for classes related using C++'s private inheritance). Another point of difference is that Extended MOBY supports structural subtyping on object types; JAVA has object types (called interfaces), but subtyping is *by-name*. C++ does not have an independent notion of object type.

Fisher's Ph.D. dissertation [Fis96] is the earliest formalization of class types that we are aware of. In her work, each class is tagged with a row variable using a form of bounded existential row. In our work, we adopt classes as a primitive notion and use the names of such classes in a fashion analogous to Fisher's row variables. A weakness of the earlier work is its treatment of private names; it provides no way to hide a method and then later add an unrelated method with the same name.

Our use of class names to label methods in an object value in XMOC (see Appendix B) is similar to the use of *role tags* on methods in Ghelli and Palmerini's calculus for modeling objects with roles [GP99]. Likewise, our pairing of an object's state with the class name that defines the current view of the object is similar to their representation of object values. The main difference between XMOC and their calculus is in the surface language features being modeled.

More recently, Igarashi *et al.* have described *Featherweight Java*, which is an object calculus designed to model the core features of JAVA's type system [IPW99]. Like our calculus, Featherweight Java has a notion of subtyping based on class inheritance. Our calculus is richer, however, in a number of ways. Our calculus models private members and narrowing of class interfaces. We also have a notion of structural subtyping and we relate the implementation and structural subtyping notions.

The notion of type identity based on implementation was present in the original definition of *Standard ML* in the form of *structure sharing* [MTH90]. The benefits of structure sharing were fairly limited and it was dropped in the 1997 revision of SML [MTHM97].

6 Conclusion

This paper presents an extension to MOBY that supports classes as types. We have illustrated the utility of this extension with a number of examples. We have also developed a formal model of this extension and have proven subject reduction for it. We are continuing to work on improving our formal treatment of class types and implementation-based inheritance.[5] One minor issue is that XMOC requires that

[5] Since this paper was written, we have developed a more elegant treatment of XMOC based on Riecke-Stone dictionaries and have proven type soundness. A paper describing this revised system is available on the MOBY web page (`http://www.cs.bell-labs.com/~jhr/moby`).

class names be unique in a program; this restriction can be avoided by introducing some mechanism, such as *stamps*, to distinguish top-level names (*e.g.*, see Leroy's approach to module system semantics [Ler96]). We would also like to generalize the rule that relates class types with object types (rule 32 in Appendix C) to allow positive occurrences of $\#C$ to be replaced by the object type's bound type variable. While we believe that this generalization is sound, we have not yet proven it.

References

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

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

[BCC+96] Bruce, K., L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. Pierce. On binary methods. *TAPOS*, **1**(3), 1996, pp. 221–242.

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

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

[FR99a] Fisher, K. and J. Reppy. The design of a class mechanism for Moby. In *PLDI'99*, May 1999, pp. 37–49.

[FR99b] Fisher, K. and J. Reppy. Foundations for MOBY classes. *Technical Memorandum*, Bell Labs, Lucent Technologies, Murray Hill, NJ, February 1999.

[FRR99] Fisher, K., J. Reppy, and J. G. Riecke. A calculus for compiling and linking classes. Submitted for publication, 1999. Available from `http://www.cs.bell-labs.com/~jhr/moby`.

[GP99] Ghelli, G. and D. Palmerini. Foundations for extensible objects with roles. In *FOOL6*, January 1999.

[IPW99] Igarashi, A., B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA'99*, November 1999, pp. 132–146.

[KLM94] Katiyar, D., D. Luckham, and J. Mitchell. A type system for prototyping languages. In *POPL'94*, January 1994, pp. 138–161.

[Ler92] Leroy, X. Unboxed objects and polymorphic typing. In *POPL'92*, January 1992, pp. 177–188.

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

[MTH90] Milner, R., M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, MA, 1990.

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

[PT93] Pierce, B. C. and D. N. Turner. Statically typed friendly functions via partially abstract types. *Technical Report ECS-LFCS-93-256*, University of Edinburgh, LFCS, April 1993. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

[RS98] Riecke, J. G. and C. Stone. Privacy via subsumption. In *FOOL5*, January 1998. A longer version will appear in TAPOS.

[Str97] Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 3rd edition, 1997.

A A brief introduction to MOBY

This appendix provides a brief introduction to some of MOBY's features to help the reader understand the examples in the paper.

MOBY programs are organized into a collection of *modules*, which have *signatures*. A module's signature controls the visibility of its components. Signatures are the primary mechanism for data and type abstraction in MOBY. To support object-oriented programming, MOBY provides *classes* and *object types*. The following example illustrates these features:

```
module M : {
  class Hi : {
    public meth hello : Unit -> Unit
    public maker mk of String
  }
  val hi : typeof(Hi)
} {
  fun pr (s : String) -> Unit { ConsoleIO.print s }
  class Hi {
    field msg : String
    public meth hello () -> Unit { pr "hello "; pr (self.msg); pr "\n" }
    public maker mk (s : String) { field msg = s }
  }
  val hi : typeof(Hi) = new mk "world"
}
```

This code defines a module `M` that is constrained by a signature with two specifications: the class `Hi` and the value `hi`. The interface of the `Hi` class specifies that it has two public components: a method `hello` and a maker `mk` ("maker" is the MOBY name for constructor functions). The signature specifies that `hi` is an object; the type expression "**`typeof`**`(Hi)`" denotes the object type induced by reading off the public methods and fields of the class `Hi`. It is equivalent to the object type definition

```
objtype HiTy { public meth hello : Unit -> Unit }
```

The body of `M` defines a function `pr` for printing strings, and the definitions of `Hi` and `hi`. Since `pr` is not mentioned in the signature of `M`, it is not exported. Note that the `Hi` class has a field `msg` in its definition. Since this field does not have a **`public`** annotation, it is only visible to subclasses. Furthermore, since `msg` is not mentioned in `M`'s signature, it is not visible to subclasses of `Hi` outside of `M`. Thus, the `msg` field is *protected* inside `M` and is *private* outside. This example illustrates MOBY's use of module signatures to implement private class members.

B Dynamic Semantics of XMOC

B.1 Syntax

We use the following classes of identifiers in the syntax of XMOC.

α	$\in$	TYVAR	type variables
τ	$\in$	TYPE	types
σ	$\in$	INTERFACE	class interfaces
B, C	$\in$	CLASSNAME $= \{\mathbf{None}, \ldots\}$	class names
$\#B, \#C$	$\in$	CLASSTYPE	class types
m	$\in$	METHNAME	method names
$\mathcal{M}$	$\in$	Fin(METHNAME)	method name sets
x	$\in$	VAR	variables
con	$\in$	CONSTR	object constructors

We follow the convention of using C when referring to a class name other than $\mathbf{None}$. The following grammar describes the full syntax of XMOC. We mark the run-time forms with a $(*)$ on the right.

$$
\begin{array}{lcll}
p & ::= & d;\ p & \\
 & | & e & \\
d & ::= & \mathtt{class}\ C\ (x : \tau) & \\
 & & \quad \{\!|\ \mathtt{inherits}\ b;\ m = \mu_m{}^{m \in \mathcal{M}}\ |\!\} & \\
 & | & \mathtt{class}\ C : \sigma = C' & \\
\sigma & ::= & (\tau)\,\{\!|\ \mathtt{inherits}\ B;\ m : \tau_m{}^{m \in \mathcal{M}}\ |\!\} & \\
\tau & ::= & \alpha & \\
 & | & \tau \to \tau' & \\
 & | & \mathtt{obj}\,\alpha\,.\,\{\!|\ m : \tau_m{}^{m \in \mathcal{M}}\ |\!\} & \\
 & | & \#B & \\
b & ::= & \mathbf{None} & \\
 & | & C(e) & \\
\mu & ::= & (x : \tau) \Rightarrow e & \\
 & | & \mathbf{super} & (*)
\end{array}
$$

$$
\begin{array}{lcll}
e & ::= & x & \\
 & | & \mathtt{fn}(x : \tau) \Rightarrow e & \\
 & | & e(e') & \\
 & | & \mathtt{new}\ C(e) & \\
 & | & \mathtt{self} & \\
 & | & e\,.\,m & \\
 & | & e\,@\,C & \\
 & | & C(e) & \\
 & | & obj & (*) \\
 & | & \langle e,\ C\rangle & (*) \\
obj & ::= & C :: & (*) \\
 & & \quad \{\!|\ \mathtt{inherits}\ pc;\ m = C_m :: \mu_m{}^{m \in \mathcal{M}}\ |\!\} & \\
pc & ::= & b & (*) \\
 & | & obj & (*) \\
con & ::= & C :: (x : \tau) & (*) \\
 & & \quad \{\!|\ \mathtt{inherits}\ b;\ m = C_m :: \mu_m{}^{m \in \mathcal{M}}\ |\!\} &
\end{array}
$$

The *obj* form represents a raw object at run-time: C is its instantiating class, *pc* names its parent class, and the list of methods available from this object follows. This list includes stubs for inherited methods, whose bodies have the run-time form $\mathbf{super}$. Methods defined in an ancestor class, but hidden in C, do not appear in this list. The *con* form provides the representation of a class constructor in a dynamic class environment. It is similar to the *obj* form, except it is parameterized by the constructor's argument.

B.2 Evaluation

The dynamic semantics is split into two phases; the first phase *links* the class declarations to produce a dynamic class environment and a residual expression. The dynamic class environment is a finite map from class names to their constructors:

$$\mathcal{K} \in \text{DYNCLASSENV} = \text{CLASSNAME} \xrightarrow{\text{fin}} \text{CONSTR}$$

We write the linking relation as $\mathcal{K}, p \rightarrowtail \mathcal{K}', p'$. The second phase *evaluates* the residual expression using the dynamic class environment to instantiate objects and resolve method dispatch. The evaluation relation is written as $\mathcal{K} \vdash e \hookrightarrow e'$.

B.3 Evaluation Contexts and Values

The dynamic semantics is specified using the standard technique of evaluation contexts. We distinguish two kinds of contexts in the specification of the evaluation relation: *expression contexts*, E, and *object initialization contexts*, F. The syntax of these contexts is as follows:

$$\begin{array}{lcl} E & ::= & [\,] \mid E(e) \mid w(E) \mid \texttt{new}\ C(E) \mid E\,\texttt{.}\,m \mid E\ @\ C \mid \langle F,\ C\rangle \\ F & ::= & [\,] \mid C :: \{\!\mid \texttt{inherits}\ F;\ m = C_m :: \mu_m{}^{m\in\mathcal{M}} \mid\!\} \mid C(E) \end{array}$$

We also define the syntactic class of *values* (w) and *object values* (ov) by the following grammar:

$$\begin{array}{lcl} w & ::= & \textbf{fn}(x : \tau) \Rightarrow e \mid w\ @\ C \mid ov \\ ov & ::= & \textbf{None} \mid \{\!\mid \texttt{inherits}\ ov;\ m = C_m :: \mu_m{}^{m\in\mathcal{M}} \mid\!\} \end{array}$$

B.4 Method lookup

We define the auxiliary *method lookup* relation $\mathcal{K} \vdash \langle obj,\ C_v\rangle\,\texttt{.}\,m \leadsto w$, which specifies how method dispatching is resolved, as follows:

$$\frac{\begin{array}{c} ov = C :: \{\!\mid \texttt{inherits}\ pc;\ m = C_m :: \mu_m{}^{m\in\mathcal{M}} \mid\!\} \\ m \in \mathcal{M} \quad \mathcal{K} \vdash C_v \lessdot C_m \quad \mathrm{Def}(m, ov) = w \end{array}}{\mathcal{K} \vdash \langle ov,\ C_v\rangle\,\texttt{.}\,m \leadsto w} \tag{1}$$

$$\frac{\mathcal{K} \vdash \langle obj,\ C_v\rangle\,\texttt{.}\,m \leadsto w \quad m \notin \mathcal{M} \text{ or } \mathcal{K} \not\vdash C_v \lessdot C_m}{\mathcal{K} \vdash \langle C :: \{\!\mid \texttt{inherits}\ ov;\ m = C_m :: \mu_m{}^{m\in\mathcal{M}} \mid\!\},\ C_v\rangle\,\texttt{.}\,m \leadsto w} \tag{2}$$

The *definition of a method* in an object value is defined as

$$\mathrm{Def}(m, ov) = \begin{cases} \mathrm{Def}(m, ov') & \text{if } \mu_m = \textbf{super} \\ \textbf{fn}(x : \tau) \Rightarrow e & \text{if } \mu_m = (x : \tau) \Rightarrow e \end{cases}$$

where $ov = C :: \{\!\mid \texttt{inherits}\ ov';\ m = C_m :: \mu_m{}^{m\in\mathcal{M}} \mid\!\}$.

B.5 Class linking

$$\begin{array}{l} \mathcal{K}, \texttt{class}\ C\ (x : \tau)\ \{\!\mid \texttt{inherits}\ \textbf{None};\ m = \mu_m{}^{m\in\mathcal{M}} \mid\!\};\ p \\ \quad \rightarrowtail \quad \mathcal{K} \pm \{C \mapsto C :: (x : \tau)\ \{\!\mid \texttt{inherits}\ \textbf{None};\ m = C :: \overline{\mu}_m{}^{m\in\mathcal{M}} \mid\!\}\}, p \end{array} \tag{3}$$

where $\overline{\mu}_m = \mu_m[\mathit{self} \mapsto \mathit{self}\ @\ C]$

$$\mathcal{K}, \texttt{class}\ C\ (x : \tau)\ \{\!\!|\ \texttt{inherits}\ C'(e);\ m = \mu_m{}^{m\in\mathcal{M}}\ |\!\!\};\ p \quad \rightarrowtail \quad \mathcal{K}\pm\{C \mapsto con\}, p$$

$$\begin{aligned} \text{where} \quad & con = C :: (x : \tau)\ \{\!\!|\texttt{inherits}\ C'(e);\ \begin{array}{l} m = C'_m :: \mathbf{super}^{m\in\mathcal{M}'\setminus\mathcal{M}} \\ m = C'_m :: \overline{\mu}_m{}^{m\in\mathcal{M}'\cap\mathcal{M}} \\ m = C :: \overline{\mu}_m{}^{m\in\mathcal{M}\setminus\mathcal{M}'}\ |\!\!\} \end{array} \\ & \overline{\mu}_m = \mu_m[self \mapsto self \mathbin{@} C] \\ & \mathcal{K}(C') = C' :: (x' : \tau')\ \{\!\!|\ \texttt{inherits}\ b';\ m = C'_m :: \mu'_m{}^{m\in\mathcal{M}'}\ |\!\!\} \end{aligned} \tag{4}$$

$$\begin{aligned} & \mathcal{K}, \texttt{class}\ C : \sigma = C';\ p \\ & \quad \rightarrowtail \quad \mathcal{K}\pm\{C \mapsto C :: (x : \tau')\{\!\!|\ \texttt{inherits}\ C'(x);\ m = C'_m :: \mathbf{super}^{m\in\mathcal{M}}\ |\!\!\}\}, p \\ & \text{where} \quad \sigma = (\tau)\ \{\!\!|\ \texttt{inherits}\ B;\ m : \tau_m{}^{m\in\mathcal{M}}\ |\!\!\} \\ & \qquad \mathcal{K}(C') = C' :: (x' : \tau')\ \{\!\!|\ \texttt{inherits}\ b';\ m = C'_m :: \mu'_m{}^{m\in\mathcal{M}'}\ |\!\!\} \end{aligned} \tag{5}$$

B.6 Expression evaluation

$$\mathcal{K} \vdash E[w\,\texttt{.}\,m] \quad \hookrightarrow \quad E[w'[self \mapsto w]] \qquad \text{where } \mathcal{K} \vdash w\,\texttt{.}\,m \leadsto w' \tag{6}$$

$$\mathcal{K} \vdash E[\langle obj,\ C_v\rangle \mathbin{@} C'_v] \quad \hookrightarrow \quad E[\langle obj,\ C'_v\rangle] \tag{7}$$

$$\mathcal{K} \vdash E[\texttt{new}\ C(w)] \quad \hookrightarrow \quad E[\langle C(w),\ C\rangle] \tag{8}$$

$$\mathcal{K} \vdash E[\texttt{fn}(x\ :\ \tau) \Rightarrow e(w)] \quad \hookrightarrow \quad E[e[x \mapsto w]] \tag{9}$$

$$\begin{aligned} & \mathcal{K} \vdash F[C(w)] \quad \hookrightarrow \quad F[C :: \{\!\!|\ \texttt{inherits}\ b[x \mapsto w];\ m = C_m :: \mu_m[x \mapsto w]^{m\in\mathcal{M}}\ |\!\!\}] \\ & \text{where } \mathcal{K}(C) = C :: (x : \tau)\ \{\!\!|\ \texttt{inherits}\ b;\ m = C_m :: \mu_m{}^{m\in\mathcal{M}}\ |\!\!\} \end{aligned} \tag{10}$$

C Typing rules for XMOC

The typing rules for XMOC are written with respect to an environment Γ, which has four parts:

Γ	$\in$	$\textsc{Env} = \textsc{TyVarSet} \times \textsc{SubtypeEnv} \times \textsc{ClassEnv} \times \textsc{VarEnv}$	typing environment
$\mathcal{A}$	$\in$	$\text{Fin}(\textsc{TyVar})$	bound type variables
$\mathcal{S}$	$\in$	$\textsc{SubtypeEnv} = \textsc{TyVar} \xrightarrow{\text{fin}} \textsc{Type}$	subtyping assumptions
$\mathcal{C}$	$\in$	$\textsc{ClassEnv} = \textsc{ClassName} \xrightarrow{\text{fin}} \textsc{Interface}$	class typing environment
$\mathcal{V}$	$\in$	$\textsc{Var} \xrightarrow{\text{fin}} \textsc{Type}$	variable typing environment

The judgement forms used in the semantics are as follows:

$\Gamma \vdash \tau \triangleright \mathbf{Ok}$	Type τ is *well-formed* w.r.t. Γ.	(rules 11–15)
$\Gamma \vdash \sigma \triangleright \mathbf{Ok}$	Class interface σ is *well-formed* w.r.t. Γ.	(rule 16)
$\Gamma \vdash \mathbf{Ok}$	Environment Γ is *well-formed.*	(rule 17)
$\Gamma \vdash \tau = \tau'$	Type τ is *equal* to τ'.	(rules 18–23)
$\Gamma \vdash B \lessdot B'$	Class B *publicly inherits* from B'.	(rules 24–26)
$\Gamma \vdash \tau <: \tau'$	Type τ is a *subtype* of τ'.	(rules 27–32)
$\Gamma \vdash \sigma <: \sigma'$	Class interface σ is a *subtype* of σ'.	(rule 33)
$\Gamma \vdash p \triangleright \Gamma', \tau$	Program p *defines* environment Γ' and *has type* τ.	(rules 34–35)
$\Gamma \vdash d \triangleright \Gamma'$	Declaration d *defines* environment Γ'.	(rules 36–38)
$\Gamma \vdash \mu \triangleright \tau$	Method μ *has type* τ.	(rule 39)
$\Gamma \vdash e \triangleright \tau$	Expression e *has type* τ.	(rules 40–46)
$\mathcal{K} \vdash \mathit{con} \triangleright \mathbf{Ok}$	Constructor con is *well-formed.*	(rules 47–48)
$\mathcal{K} \vdash \mathbf{Ok}$	Dynamic class environment $\mathcal{K}$ is *well-formed.*	(rule 49)
$\mathcal{K} \vdash C \lessdot B'$	Class C *inherits* from B'.	(rules 50–52)
$\Gamma, \mathcal{K} \vdash \mathit{pc} \triangleright \#B$	Parent class pc *has class type* $\#B$.	(rules 53–55)
$\Gamma, \mathcal{K} \vdash e \triangleright \tau$	Run-time expression e *has type* τ.	(rules 54 and 56)
$\Gamma, \mathcal{K} \vdash \mathit{con} \triangleright \sigma$	Constructor con *has interface* σ.	(rules 57–58)
$\Gamma \vdash \mathcal{K}$	Static environment Γ *types* dynamic class environment $\mathcal{K}$.	(rule 59)

C.1 Well-formedness judgements

$$\frac{\Gamma \vdash \mathbf{Ok} \quad \alpha \in (\mathcal{A} \text{ of } \Gamma)}{\Gamma \vdash \alpha \triangleright \mathbf{Ok}} \tag{11}$$

$$\frac{\Gamma \vdash \mathbf{Ok}}{\Gamma \vdash \#\mathbf{None} \triangleright \mathbf{Ok}} \tag{12}$$

$$\frac{\Gamma \vdash \tau \triangleright \mathbf{Ok} \quad \Gamma \vdash \tau' \triangleright \mathbf{Ok}}{\Gamma \vdash \tau \rightarrow \tau' \triangleright \mathbf{Ok}} \tag{13}$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad \forall m \in \mathcal{M}\ \Gamma \cup \{\alpha\} \vdash \tau_m \triangleright \mathbf{Ok}}{\Gamma \vdash \mathtt{obj}\,\alpha\,\mathtt{.}\{\!\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\!\} \triangleright \mathbf{Ok}} \tag{14}$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad \#C \in \mathrm{dom}(\mathcal{C} \text{ of } \Gamma)}{\Gamma \vdash \#C \triangleright \mathbf{Ok}} \tag{15}$$

$$\frac{\Gamma \vdash \tau \triangleright \mathbf{Ok} \quad \Gamma \vdash \#B \triangleright \mathbf{Ok} \quad \forall m \in \mathcal{M}\ \Gamma \vdash \tau_m \triangleright \mathbf{Ok}}{\Gamma \vdash (\tau)\,\{\!\!|\, \mathtt{inherits}\ B;\ m : \tau_m{}^{m \in \mathcal{M}} \,|\!\!\} \triangleright \mathbf{Ok}} \tag{16}$$

$$\frac{\forall \tau \in \mathrm{rng}(\mathcal{S} \text{ of } \Gamma)\ \Gamma \vdash \tau \triangleright \mathbf{Ok} \quad \forall \sigma \in \mathrm{rng}(\mathcal{C} \text{ of } \Gamma)\ \Gamma \vdash \sigma \triangleright \mathbf{Ok} \quad \forall \tau \in \mathrm{rng}(\mathcal{V} \text{ of } \Gamma)\ \Gamma \vdash \tau \triangleright \mathbf{Ok}}{\Gamma \vdash \mathbf{Ok}} \tag{17}$$

C.2 Equality judgements

$$\frac{\Gamma \vdash \tau \triangleright \mathbf{Ok}}{\Gamma \vdash \tau = \tau} \tag{18}$$

$$\frac{\Gamma \vdash \tau' = \tau}{\Gamma \vdash \tau = \tau'} \tag{19}$$

$$\frac{\Gamma \vdash \tau = \tau' \quad \Gamma \vdash \tau' = \tau''}{\Gamma \vdash \tau = \tau''} \tag{20}$$

$$\frac{\Gamma \vdash \tau_1 = \tau_2 \quad \Gamma \vdash \tau_1' = \tau_2'}{\Gamma \vdash \tau_1 \rightarrow \tau_1' = \tau_2 \rightarrow \tau_2'} \tag{21}$$

$$\frac{\mathcal{M}' = \mathcal{M} \quad \forall m \in \mathcal{M}\ \Gamma \cup \{\alpha\} \vdash \tau_m = \tau_m'}{\Gamma \vdash \mathtt{obj}\,\alpha\,\mathtt{.}\{\!\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\!\} = \mathtt{obj}\,\alpha\,\mathtt{.}\{\!\!|\, m : \tau'_m{}^{m \in \mathcal{M}'} \,|\!\!\}} \tag{22}$$

$$\frac{\Gamma \vdash \texttt{obj}\,\alpha\,.\,\{\!\!\{\, m : \tau_m{}^{m\in\mathcal{M}} \,\}\!\!\} \rhd \mathbf{Ok}}{\Gamma \vdash \texttt{obj}\,\alpha\,.\,\{\!\!\{\, m : \tau_m{}^{m\in\mathcal{M}} \,\}\!\!\} = \texttt{obj}\,\alpha\,.\,\{\!\!\{\, m : \tau_m[\alpha \mapsto \texttt{obj}\,\alpha\,.\,\{\!\!\{\, m : \tau_m{}^{m\in\mathcal{M}} \,\}\!\!\}]^{m\in\mathcal{M}} \,\}\!\!\}} \tag{23}$$

C.3 Subtyping judgements

$$\frac{\Gamma \vdash \mathbf{Ok} \quad C \in \mathrm{dom}(\mathcal{C}\,\mathrm{of}\,\Gamma)}{\Gamma \vdash C \lessdot C} \tag{24}$$

$$\frac{\Gamma \vdash \mathbf{Ok}}{\Gamma \vdash B \lessdot \mathbf{None}} \tag{25}$$

$$\frac{(\mathcal{C}\,\mathrm{of}\,\Gamma)(C') = (\tau)\,\{\!\!\{\, \texttt{inherits}\; C'';\; m : \tau_m{}^{m\in\mathcal{M}} \,\}\!\!\} \quad \Gamma \vdash C'' \lessdot C}{\Gamma \vdash C' \lessdot C} \tag{26}$$

$$\frac{\Gamma \vdash \tau = \tau'}{\Gamma \vdash \tau <: \tau'} \tag{27}$$

$$\frac{\Gamma \vdash \tau <: \tau' \quad \Gamma \vdash \tau' <: \tau''}{\Gamma \vdash \tau <: \tau''} \tag{28}$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad \alpha \in \mathrm{dom}(\mathcal{S}\,\mathrm{of}\,\Gamma)}{\Gamma \vdash \alpha <: \mathcal{S}(\alpha)} \tag{29}$$

$$\frac{\Gamma \vdash \tau_2 <: \tau_1 \quad \Gamma \vdash \tau_1' <: \tau_2'}{\Gamma \vdash \tau_1 \to \tau_1' <: \tau_2 \to \tau_2'} \tag{30}$$

$$\frac{\begin{array}{c}\mathcal{M}' \subseteq \mathcal{M} \quad \alpha' \notin \mathrm{FV}(\texttt{obj}\,\alpha\,.\,\{\!\!\{\, m : \tau_m{}^{m\in\mathcal{M}} \,\}\!\!\}) \\ \forall m \in \mathcal{M}'\ \alpha \notin \mathrm{FV}(\tau_m') \text{ and } (\Gamma \cup \{\alpha'\}) \pm \{\alpha \mapsto \alpha'\} \vdash \tau_m <: \tau_m'\end{array}}{\Gamma \vdash \texttt{obj}\,\alpha\,.\,\{\!\!\{\, m : \tau_m{}^{m\in\mathcal{M}} \,\}\!\!\} <: \texttt{obj}\,\alpha'\,.\,\{\!\!\{\, m : \tau'_m{}^{m\in\mathcal{M}'} \,\}\!\!\}} \tag{31}$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad (\mathcal{C}\,\mathrm{of}\,\Gamma)(C) = (\tau)\,\{\!\!\{\, \texttt{inherits}\; B;\; m : \tau_m{}^{m\in\mathcal{M}} \,\}\!\!\}}{\Gamma \vdash \texttt{\#}C <: \texttt{obj}\,\alpha\,.\,\{\!\!\{\, m : \tau_m{}^{m\in\mathcal{M}} \,\}\!\!\}} \tag{32}$$

$$\frac{\Gamma \vdash \tau' <: \tau \quad \Gamma \vdash B \lessdot B' \quad \mathcal{M}' \subseteq \mathcal{M} \quad \forall m \in \mathcal{M}'\ \Gamma \vdash \tau_m = \tau_m' \quad \forall m \in (\mathcal{M} \setminus \mathcal{M}')\ \Gamma \vdash \tau_m \rhd \mathbf{Ok}}{\Gamma \vdash (\tau)\,\{\!\!\{\, \texttt{inherits}\; B;\; m : \tau_m{}^{m\in\mathcal{M}} \,\}\!\!\} <: (\tau')\,\{\!\!\{\, \texttt{inherits}\; B';\; m : \tau'_m{}^{m\in\mathcal{M}'} \,\}\!\!\}} \tag{33}$$

C.4 Typing judgements

$$\frac{\Gamma \vdash d \rhd \Gamma' \quad \Gamma' \vdash p \rhd \Gamma'', \tau}{\Gamma \vdash d;\; p \rhd \Gamma'', \tau} \tag{34}$$

$$\frac{\Gamma \vdash e \rhd \tau}{\Gamma \vdash e \rhd \Gamma, \tau} \tag{35}$$

$$\frac{\begin{array}{c} C \notin \mathrm{dom}(\mathcal{C}\,\mathrm{of}\,\Gamma) \quad \Gamma \vdash \tau \rhd \mathbf{Ok} \\ \Gamma' = \Gamma \pm \{C \mapsto (\tau)\,\{\!\!\{\, \texttt{inherits}\; \mathrm{None};\; m : \tau_m{}^{m\in\mathcal{M}} \,\}\!\!\}\} \\ \forall m \in \mathcal{M}\ \Gamma' \pm \{x \mapsto \tau, \mathit{self} \mapsto \texttt{\#}C\} \vdash \mu_m \rhd \tau_m \end{array}}{\Gamma \vdash \texttt{class}\; C\; (x : \tau)\,\{\!\!\{\, \texttt{inherits}\; \mathrm{None};\; m = \mu_m{}^{m\in\mathcal{M}} \,\}\!\!\} \rhd \Gamma'} \tag{36}$$

$$\frac{\begin{array}{c} C \notin \mathrm{dom}(\mathcal{C}\,\mathrm{of}\,\Gamma) \quad \Gamma \vdash \tau \triangleright \mathbf{Ok} \\ (\mathcal{C}\,\mathrm{of}\,\Gamma)(C') = (\tau')\,\{\!\!\{\, \mathtt{inherits}\; B';\; m : {\tau'_m}^{m \in \mathcal{M}'} \,\}\!\!\} \\ \Gamma' = \Gamma \pm \{C \mapsto (\tau)\,\{\!\!\{\, \mathtt{inherits}\; C';\; m : {\tau'_m}^{m \in (\mathcal{M}' \setminus \mathcal{M})}\; m : {\tau_m}^{m \in \mathcal{M}} \,\}\!\!\}\} \\ \Gamma' \pm \{x \mapsto \tau\} \vdash e \triangleright \tau'' \quad \Gamma' \vdash \tau'' <: \tau' \\ \forall m \in (\mathcal{M} \cap \mathcal{M}')\; \Gamma' \vdash \tau_m <: \tau'_m \\ \forall m \in \mathcal{M}\; \Gamma' \pm \{x \mapsto \tau, \mathit{self} \mapsto \#C\} \vdash \mu_m \triangleright \tau_m \end{array}}{\Gamma \vdash \mathtt{class}\; C\; (x : \tau)\,\{\!\!\{\, \mathtt{inherits}\; C'(e);\; m = {\mu_m}^{m \in \mathcal{M}} \,\}\!\!\} \triangleright \Gamma'} \quad (37)$$

$$\frac{\begin{array}{c} C \notin \mathrm{dom}(\mathcal{C}\,\mathrm{of}\,\Gamma) \quad \Gamma' = \Gamma \pm \{C \mapsto \sigma\} \\ \Gamma' \vdash (\mathcal{C}\,\mathrm{of}\,\Gamma)(C') <: \sigma \end{array}}{\Gamma \vdash \mathtt{class}\; C : \sigma = C' \triangleright \Gamma'} \quad (38)$$

$$\frac{\Gamma \pm \{x \mapsto \tau\} \vdash e \triangleright \tau'}{\Gamma \vdash (x\,:\,\tau) \Rightarrow e \triangleright \tau \to \tau'} \quad (39)$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad (\mathcal{V}\,\mathrm{of}\,\Gamma)(x) = \tau}{\Gamma \vdash x \triangleright \tau} \quad (40)$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad (\mathcal{V}\,\mathrm{of}\,\Gamma)(\mathtt{self}) = \tau}{\Gamma \vdash \mathtt{self} \triangleright \tau} \quad (41)$$

$$\frac{\Gamma \pm \{x \mapsto \tau\} \vdash e \triangleright \tau'}{\Gamma \vdash \mathtt{fn}(x\,:\,\tau) \Rightarrow e \triangleright \tau \to \tau'} \quad (42)$$

$$\frac{\begin{array}{c} \Gamma \vdash e \triangleright \tau' \to \tau \quad \Gamma \vdash e' \triangleright \tau'' \\ \Gamma \vdash \tau'' <: \tau' \end{array}}{\Gamma \vdash e(e') \triangleright \tau} \quad (43)$$

$$\frac{(\mathcal{C}\,\mathrm{of}\,\Gamma)(C) = (\tau)\,\{\!\!\{\, \mathtt{inherits}\; B;\; m : {\tau_m}^{m \in \mathcal{M}} \,\}\!\!\} \quad \Gamma \vdash e \triangleright \tau' \quad \Gamma \vdash \tau' <: \tau}{\Gamma \vdash \mathtt{new}\; C(e) \triangleright \#C} \quad (44)$$

$$\frac{\Gamma \vdash e \triangleright \#C' \quad \Gamma \vdash C' \lessdot C}{\Gamma \vdash e\,@\,C \triangleright \#C} \quad (45)$$

$$\frac{\begin{array}{c} \Gamma \vdash e \triangleright \tau \quad m \in \mathcal{M} \\ \Gamma \vdash \tau <: \mathtt{obj}\,\alpha\,.\,\{\!\!\{\, m : {\tau_m}^{m \in \mathcal{M}} \,\}\!\!\} \end{array}}{\Gamma \vdash e\,.\,m \triangleright \tau_m[\alpha \mapsto \mathtt{obj}\,\alpha\,.\,\{\!\!\{\, m : {\tau_m}^{m \in \mathcal{M}} \,\}\!\!\}]} \quad (46)$$

C.5 Typing rules for run-time forms

The typing rules for the run-time forms include rules 39 through 46 augmented with a dynamic class environment $\mathcal{K}$ on the left-hand side of the turnstile. The remaining rules are given below.

$$\frac{\forall m \in \mathcal{M}\; C_m \in \mathrm{dom}(\mathcal{K})}{\mathcal{K} \vdash C :: (x : \tau)\{\!\!\{\, \mathtt{inherits}\; \mathrm{None};\; m = C_m :: {\mu_m}^{m \in \mathcal{M}} \,\}\!\!\} \triangleright \mathbf{Ok}} \quad (47)$$

$$\frac{C' \in \mathrm{dom}(\mathcal{K}) \quad \forall m \in \mathcal{M}\; C_m \in \mathrm{dom}(\mathcal{K})}{\mathcal{K} \vdash C :: (x : \tau)\{\!\!\{\, \mathtt{inherits}\; C'(e);\; m = C_m :: {\mu_m}^{m \in \mathcal{M}} \,\}\!\!\} \triangleright \mathbf{Ok}} \quad (48)$$

$$\frac{\forall con \in \mathrm{rng}(\mathcal{K}) \quad \mathcal{K} \vdash con \triangleright \mathbf{Ok}}{\mathcal{K} \vdash \mathbf{Ok}} \tag{49}$$

$$\frac{B \in \mathrm{dom}(\mathcal{K}) \cup \{\mathbf{None}\}}{\mathcal{K} \vdash B \lessdot B} \tag{50}$$

$$\frac{C \in \mathrm{dom}(\mathcal{K})}{\mathcal{K} \vdash C \lessdot \mathbf{None}} \tag{51}$$

$$\frac{\mathcal{K}(C') = C' :: (x : \tau) \{\!\!\{ \mathtt{\textbf{inherits}}\, C''(e);\ m = C_m :: {\mu_m}^{m \in \mathcal{M}} \}\!\!\} \quad \mathcal{K} \vdash C'' \lessdot C}{\mathcal{K} \vdash C' \lessdot C} \tag{52}$$

$$\frac{(\mathcal{C} \text{ of } \Gamma)(C) = (\tau) \{\!\!\{ \mathtt{\textbf{inherits}}\, B;\ m : {\tau_m}^{m \in \mathcal{M}} \}\!\!\} \quad \Gamma, \mathcal{K} \vdash e \triangleright \tau' \quad \Gamma \vdash \tau' <: \tau}{\Gamma, \mathcal{K} \vdash C(e) \triangleright \#C} \tag{53}$$

$$\frac{\begin{array}{c}(\mathcal{C} \text{ of } \Gamma)(C) = (\tau) \{\!\!\{ \mathtt{\textbf{inherits}}\, B;\ m : {\tau_m}^{m \in \mathcal{M}} \}\!\!\} \\ \Gamma, \mathcal{K} \vdash pc \triangleright \#B' \quad \mathcal{K} \vdash B' \lessdot B \\ \forall m \in \mathcal{M} \ \text{ if } \mu_m \neq \mathbf{super} \text{ then } \Gamma \pm \{ \mathit{self} \mapsto \#C \}, \mathcal{K} \vdash \mu_m \triangleright \tau'_m \text{ and } \Gamma \vdash \tau'_m <: \tau_m\end{array}}{\Gamma, \mathcal{K} \vdash C :: \{\!\!\{ \mathtt{\textbf{inherits}}\, pc;\ m = C_m :: {\mu_m}^{m \in \mathcal{M}} \}\!\!\} \triangleright \#C} \tag{54}$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad \mathcal{K} \vdash \mathbf{Ok}}{\Gamma, \mathcal{K} \vdash \mathbf{None} \triangleright \#\mathbf{None}} \tag{55}$$

$$\frac{\Gamma, \mathcal{K} \vdash e \triangleright \#C' \quad \mathcal{K} \vdash C' \lessdot C}{\Gamma, \mathcal{K} \vdash \langle e,\ C \rangle \triangleright \#C} \tag{56}$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau \triangleright \mathbf{Ok} \quad \sigma = (\tau) \{\!\!\{ \mathtt{\textbf{inherits}}\, \mathbf{None};\ m : {\tau_m}^{m \in \mathcal{M}} \}\!\!\} \\ \forall m \in \mathcal{M}\ \Gamma \pm \{ x \mapsto \tau, \mathit{self} \mapsto \#C \} \vdash \mu_m \triangleright \tau_m\end{array}}{\Gamma, \mathcal{K} \vdash C :: (x : \tau) \{\!\!\{ \mathtt{\textbf{inherits}}\, \mathbf{None};\ m = C :: {\mu_m}^{m \in \mathcal{M}} \}\!\!\} \triangleright \sigma} \tag{57}$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau \triangleright \mathbf{Ok} \quad \sigma = (\tau) \{\!\!\{ \mathtt{\textbf{inherits}}\, C';\ m : {\tau_m}^{m \in \mathcal{M}} \}\!\!\} \\ \Gamma, \mathcal{K} \vdash \mathcal{K}(C') \triangleright (\tau') \{\!\!\{ \mathtt{\textbf{inherits}}\, B';\ m : {\tau'_m}^{m \in \mathcal{M}'} \}\!\!\} \\ \Gamma \pm \{ x \mapsto \tau \}, \mathcal{K} \vdash e \triangleright \tau'' \quad \Gamma \vdash \tau'' <: \tau' \\ \forall m \in \{ \mathcal{M} \mid \mu_m = \mathbf{super} \}\ \Gamma \vdash \tau_m = \tau'_m \\ \forall m \in \{ \mathcal{M} \mid \mu_m \neq \mathbf{super} \}\ \Gamma \pm \{ x \mapsto \tau, \mathit{self} \mapsto \#C \} \vdash \mu_m \triangleright \tau_m\end{array}}{\Gamma, \mathcal{K} \vdash C :: (x : \tau) \{\!\!\{ \mathtt{\textbf{inherits}}\, C'(e);\ m = C_m :: {\mu_m}^{m \in \mathcal{M}'}\ m = C :: {\mu_m}^{m \in (\mathcal{M} \setminus \mathcal{M}')} \}\!\!\} \triangleright \sigma} \tag{58}$$

$$\frac{\begin{array}{c}\mathrm{dom}(\mathcal{K}) = \mathrm{dom}(\mathcal{C} \text{ of } \Gamma) \\ \forall C \in \mathrm{dom}(\mathcal{K})\ \ \Gamma(C) = \sigma \text{ and } \Gamma, \mathcal{K} \vdash \mathcal{K}(C) \triangleright \sigma' \text{ and } \Gamma \vdash \sigma' <: \sigma\end{array}}{\Gamma \vdash \mathcal{K}} \tag{59}$$