Inheritance-based subtyping

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

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

April 21, 2000

Abstract

Classes play a dual rôle in mainstream statically-typed object-oriented languages, serving as both object generators and object types. In such languages, inheritance implies subtyping. In contrast, the theoretical language community has viewed this linkage as a mistake and has focused on subtyping relationships determined by the structure of object types, without regard to their underlying implementations. In this paper, we explore why inheritance-based subtyping relations are useful and we describe two different approaches to extending the MOBY programming language with inheritance-based subtyping relations. In addition, we present a typed object calculus that supports both structural and inheritance-based subtyping, and which provides a formal accounting of our extensions to MOBY.

1 Introduction

There is a great divide between the formal study of object-oriented languages and the practice of mainstream 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 object-oriented 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.[1] What is interesting about the distinction between inheritance-based and structural subtyping is that there are certain idioms, such as friend functions, binary methods, and object cloning operations, which are natural to write in an inheritance-based framework, but are difficult to express in a structural framework. In this paper, we explore how inheritance-based subtyping relations are useful by examining the common object-oriented idiom of *friend functions*. Through a series of examples in Section 2, we investigate how one might implement this idiom in MOBY [FR99a], a language with only structural subtyping. We selected friend functions for our case study because they are the canonical example of a class permitting outside code to access its internal representation, which is the forté

[1] It is important not to confuse inheritance-based subtyping with *by-name* subtyping. With *by-name* subtyping, relationships between object types are declared explicitly (*e.g.*, JAVA's interfaces). While both inheritance-based and by-name subtyping avoid the *accidental subtyping* problem that afflicts structural subtyping (the common example of which is a cowboy and widget that both have draw methods), the type names in an inheritance-based scheme are tied to specific implementations, whereas multiple, unrelated classes may be declared to implement the same type name in a by-name scheme.

of inheritance-based subtyping. This example illustrates a deficiency of relying solely on structural subtyping in object-oriented language design.

We describe how to extend the MOBY programming language with inheritance-based subtyping by adding *class types*. In discussing this extension, we often treat class names as object types that relate via inheritance-based subtyping. Class types are a form of object type that are in a one-to-one correspondence with the class hierarchy. If an object has a given class type, then that object was instantiated from the associated class or one of its descendants. One class type can be a subtype of a second only if the class associated with the first inherits from the class associated with the second, *i.e.*, inheritance is a necessary condition for inheritance-based subtyping. Inheritance need not be a sufficient condition for inheritance-based subtyping, however. In other words, two classes may be related via inheritance, but their associated class types need not be related via subtyping.

In designing Extended MOBY, we discovered two design alternatives that differ on precisely this question: when should a subclass be considered a subtype of its superclass? In the first alternative, a subclass is a subtype of its parent precisely when the interface of the subclass is a structural subtype of the interface of its parent. In the second, a subclass is always a subtype of its parent. The first alternative leads to a simpler type system, while the second is more expressive. Aside from this one distinction, the two alternatives are quite similar. Hence we use the name Extended MOBY to refer to either extension, using more precise terminology only when necessary to highlight a difference between the two alternatives. We explore these alternatives in Sections 3 and 4, respectively.

To validate the design of Extended MOBY, we designed a formal object calculus called XMOC which models the type system of Extended MOBY. Specifically, XMOC supports both structural and inheritance-based subtyping, as well as a privacy mechanism that allows the interface of a deriving class to be smaller than the interface of its parent. We present XMOC in Section 5 and sketch a type soundness proof for it. A more detailed proof appears in Appendix D.

In Section 6 we describe how the type system of Extended MOBY relates to the inheritance-based subtyping of C++ and JAVA. Finally, we conclude in Section 7.

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, which has only structural subtyping. This study demonstrates that while it is possible to encode the friends idiom in a language with only structural subtyping, the resulting encoding is not very appealing.

MOBY is a language that combines an ML-style module system with classes and objects [FR99a]. Object types are related by structural subtyping, which is extended to other types in the standard way. One of MOBY's most important features is that it provides flexible control over class-member visibility using a combination of the class and module mechanisms. The class mechanism provides a visibility distinction between clients that use the class to create objects and clients that use the class to derive subclasses, while the module mechanism provides hiding of class members via signature matching. The

```
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

names of class members that are hidden by signature matching are truly private and can be redefined in a subclass. A brief introduction to MOBY is given in Appendix A.

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;[2] the implementation of the `union` function exploits this access. Note that the `items` field is **`public`** inside the `BagM` module, but is not part of `BagM`'s interface outside the module. Thus, objects created from subclasses of `Bag` are not known to be structural subtypes of `Rep`.

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. This limitation also applies to objects created from subclasses of `Bag`.

[2]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

2.2 Friends via representation methods

To support friends and class extension in the same class requires a public mechanism for mapping from an object to its abstract representation type. With this 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` 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.

Although this example does not include friends for the `CBag` class, we have included the representation method in its interface to illustrate 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, this approach 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 interfaces, it is awkward and unpleasant.

3 Extended MOBY

In the previous section, we showed how one 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 and the next, we describe an extension to MOBY [FR99a] that has class types and inheritance-based subtyping. We describe two alternative approaches to typing this extension. These alternatives are distinguished by when they allow a subclass to be viewed as a subtype of its parent class. In the first design, a class `C` is viewed as a subtype of its parent class `B` only when **`typeof`**`(C)` is a subtype of **`typeof`**`(B)`. Because MOBY allows the subclass `C` to *hide* (or make private) inherited public

```
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

members, the corresponding object types may not be in the subtyping relation (or **typeof**(B) might even be a subtype of **typeof**(C)). In our second design, C may be viewed as a subtype of B even when **typeof**(C) is not a subtype of **typeof**(B). In both designs, it is possible to hide inheritance relationships so that they do not induce subtyping relationships.

Although it is not our main motivation, it is worth noting that method and field selection from an object with a class type can be implemented as a constant time operation without sophisticated compiler analysis. 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.

3.1 Adding inheritance-based subtyping

Our first approach to adding inheritance-based subtyping requires the following additions and modifications to MOBY's type system.

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

2. 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.

3. When a class interface declares that it inherits from a class B, we require that the object type defined by the public members of the interface be a subtype of **typeof**(B).

4. 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].

5. The existing typing judgements for method and field selection require the argument to have an object type that includes the given method or field. We modify these rules to require that the argument be a subtype of an object type that includes the given method or field. Thus when the argument type is a class type, Clause 4 allows the selection to be typed.

6. 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).

7. When typing a **new** expression, we assign the corresponding class type to the result.

3.2 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 4), 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

```
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 4: Bags with friends in Extended MOBY

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.

3.3 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, Figure 5 shows how the `union` function in the previous example can be implemented as a binary method.

```
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 }
}
```

Figure 5: Bags with a binary `union` method.

```
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 }

  public 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 (a : Int, b : Int) {
    super mkB(a);
    field y = b
  }
  maker copyC (orig : #C) {
    super copyB(orig)
    field y = self.y
  }

  field y : Int
}
```

Figure 6: Cloning with privacy in Extended MOBY

3.4 Object cloning

Another case where inheritance-based subtyping is useful is in the typing of *copy constructors*. Copy constructors are useful for implementing object cloning. For example, Figure 6 gives a small Extended MOBY class hierarchy that supports object cloning.[3] Each class in the hierarchy supports a `clone`

[3]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 `pvtX` field.

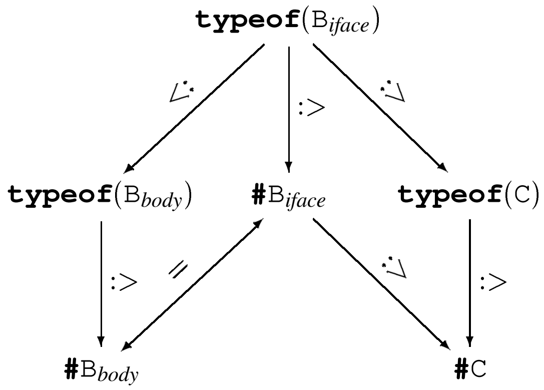

Figure 7: The subtyping relationships in the clone example

method that is implemented by creating a new object of the class using the class's copy constructor applied to **self**.[4] In the case of the derived class C, the copy constructor invokes B's copy constructor to copy the fields defined by B. These include the field pvtX, which is not mentioned in B's interface, and thus is hidden from C.

It is instructive to look at the relationships between the various types in this example. Using the notation $\mathtt{B}_{body}$ for the class B as seen from its body and $\mathtt{B}_{iface}$ for the class B as seen through its interface, Figure 7 illustrates the various type relationships. Note that because of the hiding of the pvtX field, **typeof**(C) is not related to **typeof**($\mathtt{B}_{body}$), but since **typeof**(C) is a subtype of **typeof**($\mathtt{B}_{iface}$), we are allowed to declare that C inherits from B.

If we were to replace the class types in this example with their corresponding object types (*e.g.*, **typeof**(C) for **#**C), the example would not typecheck because the type of the copyB maker function in B's interface would be a subtype of its type in B's body. This failure is what one would expect, since an object of **typeof**($\mathtt{B}_{iface}$) is not guaranteed to have a pvtX field in its implementation. Using class types, however, avoids this problem, since an object that has type **#**B is guaranteed to be implemented by class B (or one of its subclasses) and thus will have a pvtX field in its representation.

4 An alternative extension

An alternative to the previous design is to allow inheritance-based subtyping even when the object-type implemented by the subclass is not a subtype of the parent class's object type. More formally, this corresponds to dropping Clause 3 from the list in Section 3.1. Because this second form of inheritance-based subtyping admits more subtyping relationships, it is strictly more expressive than the first. In Section 4.2 we give an example that demonstrates this greater expressiveness. All of the previous examples of inheritance-based subtyping carry though without change.

We must also replace Clause 5 with a different approach because of a technical problem that we describe in Section 4.1. Instead of adding subsumption to the standard MOBY rules for method and

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

```
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 8: Example of reusing a private method name

field selection, we add new judgements for the case where the argument has a class type.

4.1 Inheritance-based subtyping vs. privacy

There is a potential problem in our second extension's type system involving the interaction between 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` defines a method with the same name as a method that exists in `B` but is hidden from `C`. Consider the code fragment in Figure 8, for example. Given these definitions, how do we typecheck the expression: "`(new mkC()).m1()`?" If we allow subsumption on the left-hand side of the method selection, then there are two incompatible ways to typecheck this expression. It is to avoid this ambiguity that we replace Clause 5 in Section 3.1.

4.2 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 9 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 subsumption 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 directly supports mixins.

This example will not typecheck under the more restrictive form of inheritance-based subtyping described in Section 3. The problem is that the implicit signature of the parameterized `PrintMix`

```
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 9: Encoding mixins in Extended MOBY

module declares that the `Pr` class inherits from the formal `M.B`, which means that **`typeof`**(`Pr`) must be a subtype of the object type of any class bound to `M.B`. The class `A` violates this requirement and thus the application of the `PrintMix` module will fail to typecheck.

5 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 type soundness results. The full system and an outline of our soundness theorem appears in Appendices B, C, and D.

5.1 Syntax

The term syntax of XMOC is given in Figure 10. An XMOC program consists of a sequence of class

$$
\begin{array}{lcl}
p & ::= & d;\ p \\
 & \mid & e \\
d & ::= & \mathbf{class}\ C\ (x : \tau)\ \{\!|\ b\ \ m = \mu_m{}^{m \in \mathcal{M}}\ |\!\} \\
 & \mid & \mathbf{class}\ C : \sigma = C' \\
b & ::= & \\
 & \mid & \mathbf{inherits}\ C(e); \\
\sigma & ::= & (\tau)\ \{\!|\ \mathbf{inherits}\ B;\ m : \tau_m{}^{m \in \mathcal{M}}\ |\!\} \\
\tau & ::= & \alpha \\
 & \mid & \tau \to \tau' \\
 & \mid & \mathbf{obj}\ \alpha\,.\{\!|\ m : \tau_m{}^{m \in \mathcal{M}}\ |\!\} \\
 & \mid & \#C
\end{array}
\qquad
\begin{array}{lcl}
\mu & ::= & (x\ :\ \tau) \Rightarrow e \\
e & ::= & x \\
 & \mid & \mathbf{fn}(x\ :\ \tau) \Rightarrow e \\
 & \mid & e(e') \\
 & \mid & \mathbf{new}\ C(e) \\
 & \mid & \mathbf{self} \\
 & \mid & e\,.\,m \\
 & \mid & \mathbf{self!state} \\
 & \mid & e\,@\,C
\end{array}
$$

Figure 10: Syntax of XMOC terms

declarations terminated by an expression. Each declaration binds a *class name* to a class definition; we do not allow rebinding of class names. The set of class names includes the distinguished name $\mathbf{None}$, which we use to denote the super-class of base classes. We follow the convention of using C for class names other than $\mathbf{None}$ and B for class names including $\mathbf{None}$. Class declarations come in two forms. In the first, a class C is defined by extending an optional parent class (b) by overriding and adding methods. When no parent is specified, we say that C is a *base-class*; otherwise, we say that C *inherits* from its parent class. To model the notion of object state, we parameterize this form of class declaration by a variable x. When an object is created, the argument bound to x is included in the representation of the object as its state. In addition, if the class has a superclass, x may be used to compute the argument to be passed to the superclass initialization function. Since an object's implementation is spread over a hierarchy of classes, it has a piece of state for each class in its implementation. Methods defined in a given class may access the state for that class using the form **self!state**. 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 a more restrictive class interface σ. A class interface σ specifies the type of the class's state, the name of the nearest revealed ancestor class (or $\mathbf{None}$), and a typed list of available methods. The types of XMOC include type variables, function types, recursive object types, and class types.

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), method dispatch, and access to the object's state. 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 4.1 without requiring two typing judgements for method dispatch. It is possible to insert these coercions automatically 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]).

5.2 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 a new dynamic class environment $\mathcal{K}'$ and a residual program p. 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 $\mathcal{K}, e \rightarrowtail \mathcal{K}, e'$. (We also use the notation $\mathcal{K} \vdash e \hookrightarrow e'$ to emphasize that the dynamic class environment does not change during this phase.) 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 of 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 8). 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. We use a variation of the Riecke-Stone dictionary technique [RS98, FR99b] to solve this problem. Intuitively, dictionaries provide the α-conversion needed to avoid capture by mapping method names to *slots* in method suites. To adapt this technique to XMOC, when we link a class C, we replace each occurrence of **self** in the methods that C defines with the object view expression "$\mathit{self} @ C$." Rule 5 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 consists of the object's state and the name of its instantiating class; this class name is used to find the object's method suite. The view represents the visibility context in which a message send occurs; those methods in scope in the view class are available. To lookup method m in runtime object $\langle \mathit{obj},\ C\rangle$, we use the dictionary associated with C in the class environment $\mathcal{K}$ to find the relevant slot. We use this slot to index into the method suite associated with obj. Rule 3 in Appendix B formally specifies method lookup.

5.3 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 ($\mathcal{V}$). We define these environments and give the complete set of XMOC typing judgements 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

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

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 subsumption, we can treat a class type as an object type according to the following subtyping judgement:

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

This rule exposes all the methods publicly available from class C because we install all such methods in the class interface that we associate with C in Γ. This rule corresponds to the property that `#C` is a subtype of **`typeof(C)`** in Extended MOBY.

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

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

The judgement $\mathcal{H}(\Gamma) \vdash C' \lessdot C$ states that C' inherits from C in the class hierarchy $\mathcal{H}(\Gamma)$ (the meta-function $\mathcal{H}$ projects the class hierarchy from the environment Γ). Because we do not treat inheritance directly as subtyping in XMOC, we only need one rule for typing method dispatch.

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

5.4 Soundness

We have proven the type soundness of XMOC. We outline this result here and give a more detailed description in Appendix D. We start with a *subject reduction* theorem, which states that the evaluation relation preserves the type of programs and class environments.

Theorem (Subject Reduction) If $\mathcal{K}$ is a closed dynamic class environment such that $\Gamma \vdash \mathcal{K}$ and for a closed program p we have $\Gamma, \mathcal{H}(\mathcal{K}) \vdash p \triangleright \Gamma_f, \tau_f$ and $\mathcal{K}, p \rightarrowtail \mathcal{K}', p'$, then there exists an environment Γ' and a type τ', such that

- $\Gamma', \mathcal{H}(\mathcal{K}') \vdash p' \triangleright \Gamma_f, \tau'$,
- $\Gamma' \vdash \tau' <: \tau_f$, and
- $\Gamma' \vdash \mathcal{K}'$

Furthermore, if p is an expression, then $\Gamma_f = \Gamma'$.

The proof has two steps: first we show that linking preserves types and produces a well-typed class environment, and then we show that expression evaluation preserves the type of expressions.

The next step is a *progress* theorem, which states that well-typed programs do not get *stuck*.

Theorem (Progress) If $\mathcal{K}$ is a closed dynamic class environment such that $\Gamma \vdash \mathcal{K}$ and for a program p we have $\Gamma, \mathcal{H}(\mathcal{K}) \vdash p \triangleright \Gamma_f, \tau_f$ with p and τ_f both closed, then either p is a value or there exists $\mathcal{K}'$ and p' such that $\mathcal{K}, p \rightarrowtail \mathcal{K}', p'$.

Here we use $\mathcal{H}$ to project the *dynamic* class hierarchy from the dynamic class environment $\mathcal{K}$.

Definition We say that a program p *yields* $\mathcal{K}', p'$ if $\emptyset, p \rightarrowtail \mathcal{K}', p'$ and there does *not* exist $\mathcal{K}'', p''$ such that $\mathcal{K}', p' \rightarrowtail \mathcal{K}'', p''$.

Finally, we show the type soundness of XMOC.

Theorem (Type Soundness) If $\emptyset \vdash p \triangleright \Gamma_f, \tau_f$ and p yields $\mathcal{K}', p'$, then p' is a value w such that $\Gamma_f, \mathcal{H}(\mathcal{K}') \vdash w \triangleright \tau'$ with $\Gamma_f \vdash \tau' <: \tau_f$.

6 Related work

Our class types are motivated by the rôle that classes play in languages like C++ and JAVA. There are several differences, however, between Extended MOBY and these other languages. Extended MOBY supports both inheritance-based and structural subtyping, whereas neither C++ or JAVA supports structural subtyping. Another important difference is in the way that abstraction is supported. MOBY and Extended MOBY allow 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 4.1). JAVA does not support private inheritance, but C++ allows a class to *privately* inherit from another class. In that case, all of the base-class members are hidden and there is no subtyping relationship. Extended MOBY is more flexible, since it allows hiding to be done on a per-member basis. It 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 C++'s private inheritance).

Fisher's Ph.D. dissertation [Fis96] is the earliest formalization of class types that we are aware of. In her work, each implementation is tagged with a row variable using a form of bounded existential rows. 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 dictionaries to specify the semantics of method dispatch in the presence of privacy is adapted from the work of Riecke and Stone [RS98]. The main difference is that XMOC has an explicit notion of class and we introduce dictionaries as a side-effect of linking classes.

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 that it models private members and narrowing of class interfaces. In addition, we have a notion of structural subtyping, and we relate the inheritance-based 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].

7 Conclusion

This paper presents an extension to MOBY that supports classes as types and inheritance-based subtyping. We have described two alternative type systems for this extension and have illustrated its utility with a number of examples. These alternatives are distinguished by when they allow a subclass to be viewed as a subtype of its parent class. Our first design restricts inheritance-based subtyping to those situations where the object type implemented by the subclass is a subtype of the object type implemented by its superclass. Our second design relaxes this restriction at the cost of more complicated rules for typing method and field selection and of requiring explicit type coercions in some situations. While the latter system is more expressive, and there are plausible examples of the usefulness of the extra expressiveness, we believe that the former design is a better choice for integrating into MOBY.

We believe that Extended MOBY is the first design that incorporates types for objects that range from class types to structural object types.[5] While this flexibility goes beyond what is found in other languages, the most interesting aspect of the design is the interaction between privacy (specified via module signature matching) and inheritance-based subtyping. Because our second design allows the inherits relation between two class types to be visible even when their corresponding object types are not in a subtyping relationship, one can use such class types to recover access to hidden methods. This property enables more flexible use of parameterized modules in combining class implementations, as illustrated in Section 4.2.

We have also developed a formal model of this extension and have proven type soundness for it. We are continuing to work on improving our formal treatment of class types and implementation-based inheritance. One minor issue is that XMOC requires that 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 36 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.

[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.

[5] JAVA is close to Extended Moby in this respect, but interface subtyping relations in JAVA must be declared ahead of time by the programmer, whereas object-type subtyping in MOBY is based on the structure of the types.

[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. A more detailed description of MOBY object-oriented features can be found in [FR99a].

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 { 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 Notation

If A and B are two sets, we write $A \setminus B$ for the set difference and $A \pitchfork B$ if they are *disjoint*. We use the notation $A \stackrel{\text{fin}}{\to} B$ to denote the set of finite maps from A to B. We write $\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$ for the finite map that maps a_1 to b_1, etc. For a finite map f, we write $\text{dom}(f)$ for its domain and $\text{rng}(f)$ for its range. If f and g are finite maps, we write $f \pm g$ for the finite map

$$\{x \mapsto g(x) \mid x \in \text{dom}(g)\} \cup \{x \mapsto f(x) \mid x \in \text{dom}(f) \setminus \text{dom}(g)\}$$

and $f\!\downarrow A$ for the restriction of f's domain to the set A. We write $t[x \mapsto t']$ for the *capture-free* substitution of t' for x in the term t.

B.2 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
μ	$\in$	METHBODY	method bodies
e	$\in$	EXP	expressions
obj	$\in$	OBJEXP	object expressions

We follow the convention of using C when referring to a class name other than $\mathbf{None}$.

Figure 11 describes the full syntax of XMOC. In this grammar, we mark the run-time forms with a $(*)$ on the right. The form $\langle obj, C\rangle$ denotes a dynamic object. The obj portion, which we refer to as a *raw object*, stores the object's state and the name of its defining class. The C portion indicates the

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

Figure 11: The full syntax of XMOC

current *view* of the object, which encodes the object's visibility context. Only those methods visible in class C are available in the current context.

During evaluation, we use several syntactic forms to denote raw objects. In the first such form, $C :: \{\!|\ e;\ obj\ |\!\}$, class C is the instantiating class of the object, e denotes its local state, and obj represents the portion of the object inherited from its parent class. The methods associated with class C are found in the dynamic class environment, defined below. The second such form, **None**, is used to denote the inherited portion of raw objects that were not defined via inheritance. The third form $C(e)$ is used during evaluation to specify the computation that initializes the portion of a raw object defined in class C. Expression e denotes the argument to this initialization function.

B.3 Evaluation

To define the evaluation relation, we need some additional definitions:

$$
\begin{array}{rcll}
 & & \textsc{Slot} & \text{method suite slots} \\
\phi & \in & \textsc{Dict} = \textsc{MethName} \stackrel{\text{fin}}{\to} \textsc{Slot} & \text{dictionaries} \\
\mu T & \in & \textsc{MethSuite} = \textsc{Slot} \stackrel{\text{fin}}{\to} \textsc{MethBody} & \text{method suites} \\
H & \in & \textsc{Hierarchy} = \textsc{ClassName} \stackrel{\text{fin}}{\to} \textsc{ClassName} & \text{class hierarchy}
\end{array}
$$

A *dictionary* ϕ is a one-to-one finite function from method names to slots. We say that ϕ is a *slot assignment* for a set of method names M if and only if $\mathrm{dom}(\phi) = M$. A *method suite* is a finite function from slots to method bodies. A class hierarchy describes the inheritance relationship between classes by mapping each class name to the name of its superclass.

The dynamic semantics is split into two phases; the first phase *links* class declarations to produce a dynamic class environment and a residual expression. The dynamic class environment maps each class name in its domain to a tuple that stores the name of the parent class, a constructor expression for initializing instantiated objects, a method suite, and a dictionary.

$$\mathcal{K} \in \textsc{DynClassEnv} = \textsc{ClassName} \stackrel{\text{fin}}{\to} (\textsc{ClassName} \times \textsc{Exp} \times \textsc{MethSuite} \times \textsc{Dict})$$

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 formally as $\mathcal{K}, e \rightarrowtail \mathcal{K}, e'$, but we abbreviate this notation to $\mathcal{K} \vdash e \hookrightarrow e'$ when convenient.

B.4 Dynamic Class Hierarchy

We construct the dynamic class hierarchy from the dynamic class environment $\mathcal{K}$ as follows:

$$\mathcal{H}(\mathcal{K}) = \{C \mapsto B \mid C \in \mathrm{dom}(\mathcal{K}) \text{ and } \mathcal{K}(C) = (B,\ \_,\ \_,\ \_)\}$$

B.5 Evaluation Contexts and Values

We specify the dynamic semantics 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:

$$\begin{array}{rcl} E & ::= & [\,] \mid E(e) \mid w(E) \mid \mathtt{new}\ C(E) \mid E\,\mathtt{!state} \mid E \mathbin{@} C \mid E\,.\,m \mid \langle F,\ C\rangle \\ F & ::= & [\,] \mid C(E) \mid C :: \{\!\mid E;\ obj \mid\!\} \mid C :: \{\!\mid w;\ F \mid\!\} \end{array}$$

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

$$\begin{array}{rcl} w & ::= & \mathbf{fn}(x\ :\ \tau) \Rightarrow e \mid \langle ov,\ C\rangle \\ ov & ::= & \mathbf{None} \mid C :: \{\!\mid w;\ ov \mid\!\} \end{array}$$

B.6 Field lookup

We use the auxiliary *field lookup* relation $C_v \vdash C :: \{\!\mid w;\ ov \mid\!\} \leadsto_f w'$ to specify the semantics of state selection. In this judgement form, the class name on the left of the turnstile denotes the visibility context of the field selection operation.

$$\frac{}{C \vdash C :: \{\!\mid w;\ ov \mid\!\} \leadsto_f w} \quad (1) \qquad \frac{C_v \neq C \qquad C_v \vdash ov \leadsto_f w'}{C_v \vdash C :: \{\!\mid w;\ ov \mid\!\} \leadsto_f w'} \quad (2)$$

In the first of these rules, the defining class of the raw object matches the context class. Hence the value w is the state we are looking for. In the second rule, the classes don't match. In this case, the desired state must have been defined by a parent class, so we continue our search with the parent raw object, ov.

B.7 Method lookup

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

$$\frac{\mathcal{K}(C_v) = (\_,\ \_,\ \_,\ \phi) \qquad \mathcal{K}(C) = (\_,\ \_,\ \mu T,\ \_) \qquad \mu T(\phi(m)) = (x\ :\ \tau) \Rightarrow e}{\mathcal{K} \vdash \langle C :: \{\!\mid w;\ ov \mid\!\},\ C_v\rangle\,.\,m \leadsto_m \mathbf{fn}(x\ :\ \tau) \Rightarrow e[\mathit{self} \mapsto C :: \{\!\mid w;\ ov \mid\!\}]} \quad (3)$$

In this rule, C_v is the view of the object being sent the message m, while C is the class that defined the object. We use C_v to index into the dynamic class environment to retrieve the method dictionary ϕ associated with class C_v. The method names in the domain of this dictionary are the methods currently available from the object. We use class C to access the method suite μT associated with C. The method bodies stored in μT are the implementations provided by the object. We lookup the slot associated with method name m in ϕ, and then index into μT using that slot to obtain the desired method body. The type system insures that the range of ϕ is a subset of the domain of μT. We convert the method body into a function and replace all occurrences of self with the receiver object. We return the resulting expression.

B.8 Class linking

The evaluation relation for class linking is defined by the following three rules, which each link one class declaration into the dynamic class hierarchy. The first rule describes the case of a base-class declaration. Since a base class has no superclass, the tuple associated with this class stores **None** as the name of the parent class and does not require a parent initialization expression. It constructs the method suite from the method bodies defined in the class, first rewriting all occurrences of self to $\mathit{self} @ C$ to set the appropriate view for invocations through self.

$$\begin{array}{l} \mathcal{K}, \texttt{class}\ C\ (x : \tau) \{\!\!\{\ m = \mu_m{}^{m \in \mathcal{M}}\ \}\!\!\};\ p \ \rightarrowtail\ \mathcal{K} \pm \{C \mapsto (\mathbf{None}, -, \mu T, \phi)\}, p \\ \text{where } \ \overline{\mu}_m = \mu_m[\mathit{self} \mapsto \mathit{self} @ C] \\ \qquad \phi \text{ is a slot assignment for } M. \\ \qquad \mu T = \{\phi(m) \mapsto \overline{\mu}_m | m \in M\} \end{array} \tag{4}$$

The second rule handles the case of a subclass declaration. This rule constructs the dictionary for the new class (C) by augmenting the dictionary of the parent class (C') with fresh slot assignments for newly defined methods (those in $M \setminus \mathrm{dom}(\phi_{C'})$). It constructs the method suite for C by extending the C' suite with mappings from the freshly allocated slots to the new method bodies and replacing the method bodies for overridden methods.

$$\begin{array}{l} \mathcal{K}, \texttt{class}\ C\ (x : \tau) \{\!\!\{\ \texttt{inherits}\ C'(e);\ m = \mu_m{}^{m \in \mathcal{M}}\ \}\!\!\};\ p \\ \quad \rightarrowtail \quad \mathcal{K} \pm \{C \mapsto (C', \mathbf{fn}(x : \tau) \Rightarrow e, \mu T_C, \phi_C)\}, p \\ \text{where } \ \mathcal{K}(C') = (\_, \_, \mu T_{C'}, \phi_{C'}) \\ \qquad \phi \text{ is a slot assignment for } M \setminus \mathrm{dom}(\phi_{C'}) \text{ such that } \mathrm{rng}(\phi) \pitchfork \mathrm{dom}(\mu T_{C'}) \\ \qquad \phi_C = \phi_{C'} \cup \phi \\ \qquad \overline{\mu}_m = \mu_m[\mathit{self} \mapsto \mathit{self} @ C] \\ \qquad \mu T_C = \mu T_{C'} \pm \{\phi_C(m) \mapsto \overline{\mu}_m | m \in M\} \end{array} \tag{5}$$

The third linking rule describes class signature ascription. This declaration hides the portions of the C' class not explicitly mentioned in the class signature. Dynamically, we capture this behavior by adopting for C the dictionary associated with C' after all but the methods in M have been removed from its domain.

$$\begin{array}{l} \mathcal{K}, \texttt{class}\ C : (\tau) \{\!\!\{\ \texttt{inherits}\ B;\ m : \tau_m{}^{m \in \mathcal{M}}\ \}\!\!\} = C';\ p \\ \quad \rightarrowtail \quad \mathcal{K} \pm \{C \mapsto (C', \mathbf{fn}(x : \tau) \Rightarrow x, \mu T_C, \phi_C)\}, p \\ \text{where } \ \mathcal{K}(C') = (\_, \_, \mu T_{C'}, \phi_{C'}) \\ \qquad \mu T_C = \mu T_{C'} \\ \qquad \phi_C = \phi_{C'} \downarrow M \end{array} \tag{6}$$

B.9 Expression evaluation

The following rules specify the semantics of expression evaluation. We use the notation $\mathcal{K} \vdash eo \hookrightarrow eo'$ (where meta-variable eo ranges over either e or obj) as a shorthand for the notation $\mathcal{K}, eo \rightarrowtail \mathcal{K}, eo'$ when the run-time environment doesn't change.

$$\mathcal{K} \vdash E[\mathbf{fn}(x : \tau) \Rightarrow e(w)] \quad \hookrightarrow \quad E[e[x \mapsto w]] \tag{7}$$

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

$$\mathcal{K} \vdash E[\langle ov,\, C_v\rangle\, \mathbf{!state}] \quad \hookrightarrow \quad E[w] \qquad \text{where } C_v \vdash ov \rightsquigarrow_f w \tag{9}$$

$$\mathcal{K} \vdash E[\langle ov,\, C_v\rangle \mathbin{@} C'_v] \quad \hookrightarrow \quad E[\langle ov,\, C'_v\rangle] \tag{10}$$

$$\mathcal{K} \vdash E[w \,\mathbf{.}\, m] \quad \hookrightarrow \quad E[w'] \qquad \text{where } \mathcal{K} \vdash w \,\mathbf{.}\, m \rightsquigarrow_m w' \tag{11}$$

$$\begin{array}{l} \mathcal{K} \vdash F[C(w)] \quad \hookrightarrow \quad F[C :: \{\!|\, w;\; obj \,|\!\}] \\ \text{where} \quad obj = \begin{cases} \mathbf{None} & \text{if } \mathcal{K}(C) = (\mathbf{None},\, \_,\, \_,\, \_) \\ C'(e(w)) & \text{if } \mathcal{K}(C) = (C',\, e,\, \_,\, \_) \end{cases} \end{array} \tag{12}$$

C Typing rules for XMOC

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

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

C.1 Static Class Hierarchy

We construct the static class hierarchy from the environment Γ as follows:

$$\mathcal{H}(\Gamma) = \{(C \mapsto B) \mid (\mathcal{C} \text{ of } \Gamma)(C) = (\tau)\, \{\!|\, \mathbf{inherits}\, B;\; m : \tau_m{}^{m \in \mathcal{M}} \,|\!\}\}$$

C.2 Judgement forms

The XMOC type system uses the following judgement forms:

$\Gamma \vdash \tau \triangleright \mathbf{Ok}$	Type τ is *well-formed* w.r.t. Γ.	(rules 13–17)
$\Gamma \vdash \sigma \triangleright \mathbf{Ok}$	Class interface σ is *well-formed* w.r.t. Γ.	(rule 18)
$H \vdash \mathbf{Ok}$	Class hierarchy H is *well-formed*.	(rule 19)
$\Gamma \vdash \mathbf{Ok}$	Environment Γ is *well-formed*.	(rule 20)
$\Gamma \vdash \tau = \tau'$	Type τ is *equal* to τ'.	(rules 21–26)
$H \vdash B_1 \lessdot B_2$	Class B_1 *inherits* from B_2.	(rules 27–30)
$\Gamma \vdash \tau <: \tau'$	Type τ is a *subtype* of τ'.	(rules 31–36)
$\Gamma \vdash \sigma <: \sigma'$	Class interface σ is a *subtype* of σ'.	(rule 37)
$\Gamma \vdash p \triangleright \Gamma', \tau$	Program p *defines* environment Γ' and *has type* τ.	(rules 38–39)
$\Gamma \vdash d \triangleright \Gamma'$	Declaration d *defines* environment Γ'.	(rules 40–42)
$\Gamma \vdash \mu \triangleright \tau$	Method μ *has type* τ.	(rule 43)
$\Gamma \vdash e \triangleright \tau$	Expression e *has type* τ.	(rules 44–51)
$\Gamma, H \vdash \mathit{obj} \triangleright \#B$	Parent class obj *has class type* $\#B$.	(rules 52–54)
$\Gamma, H \vdash e \triangleright \tau$	Run-time expression e *has type* τ.	(rules 53 and 56)
$\Gamma, H, C \vdash \mathit{con} \triangleright \sigma$	Constructor con, associated with class C, *has interface* σ.	(rules 57–58)
$\Gamma \vdash \mathcal{K}$	Static environment Γ *types* dynamic class environment $\mathcal{K}$.	(rule 59)

C.3 Well-formedness judgements

$$\frac{\Gamma \vdash \mathbf{Ok} \quad \alpha \in (\mathcal{A} \text{ of } \Gamma)}{\Gamma \vdash \alpha \triangleright \mathbf{Ok}} \quad (13) \qquad \frac{\Gamma \vdash \mathbf{Ok}}{\Gamma \vdash \#\mathbf{None} \triangleright \mathbf{Ok}} \quad (14)$$

$$\frac{\Gamma \vdash \tau \triangleright \mathbf{Ok} \quad \Gamma \vdash \tau' \triangleright \mathbf{Ok}}{\Gamma \vdash \tau \to \tau' \triangleright \mathbf{Ok}} \quad (15) \qquad \frac{\Gamma \vdash \mathbf{Ok} \quad \#C \in \mathrm{dom}(\mathcal{C} \text{ of } \Gamma)}{\Gamma \vdash \#C \triangleright \mathbf{Ok}} \quad (16)$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad \forall m \in \mathcal{M}\ \Gamma \cup \{\alpha\} \vdash \tau_m \triangleright \mathbf{Ok}}{\Gamma \vdash \mathtt{obj}\,\alpha\,\mathtt{.}\,\{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\} \triangleright \mathbf{Ok}} \quad (17)$$

$$\frac{\Gamma \vdash \tau \triangleright \mathbf{Ok} \quad \Gamma \vdash \#B \triangleright \mathbf{Ok} \quad \forall m \in \mathcal{M}\ \Gamma \vdash \tau_m \triangleright \mathbf{Ok}}{\Gamma \vdash (\tau)\,\{\!|\, \mathtt{inherits}\,B;\ m : \tau_m{}^{m \in \mathcal{M}} \,|\!\} \triangleright \mathbf{Ok}} \quad (18)$$

$$\frac{\forall C \in \mathrm{dom}(H) \quad \mathcal{H}(C) \in \mathrm{dom}(H) \cup \{\mathbf{None}\}}{H \vdash \mathbf{Ok}} \quad (19)$$

$$\frac{\begin{array}{c}\forall \tau \in \mathrm{rng}(\mathcal{S} \text{ of } \Gamma)\ \Gamma \vdash \tau \triangleright \mathbf{Ok} \\ \forall \sigma \in \mathrm{rng}(\mathcal{C} \text{ of } \Gamma)\ \Gamma \vdash \sigma \triangleright \mathbf{Ok} \\ \forall \tau' \in \mathrm{rng}(\mathcal{V} \text{ of } \Gamma)\ \Gamma \vdash \tau' \triangleright \mathbf{Ok}\end{array}}{\Gamma \vdash \mathbf{Ok}} \quad (20)$$

C.4 Equality judgements

$$\frac{\Gamma \vdash \tau \triangleright \mathbf{Ok}}{\Gamma \vdash \tau = \tau} \quad (21) \qquad \frac{\Gamma \vdash \tau' = \tau}{\Gamma \vdash \tau = \tau'} \quad (22)$$

$$\frac{\Gamma \vdash \tau = \tau' \quad \Gamma \vdash \tau' = \tau''}{\Gamma \vdash \tau = \tau''} \quad (23) \qquad \frac{\Gamma \vdash \tau_1 = \tau_2 \quad \Gamma \vdash \tau_1' = \tau_2'}{\Gamma \vdash \tau_1 \to \tau_1' = \tau_2 \to \tau_2'} \quad (24)$$

$$\frac{\mathcal{M}' = \mathcal{M} \quad \forall m \in \mathcal{M}\ \Gamma \cup \{\alpha\} \vdash \tau_m = \tau_m'}{\Gamma \vdash \mathbf{obj}\,\alpha\,.\,\{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\} = \mathbf{obj}\,\alpha\,.\,\{\!|\, m : \tau'_m{}^{m \in \mathcal{M}'} \,|\!\}} \quad (25)$$

$$\frac{\Gamma \vdash \mathbf{obj}\,\alpha\,.\,\{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\} \triangleright \mathbf{Ok}}{\begin{array}{l}\Gamma \vdash \mathbf{obj}\,\alpha\,.\,\{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\} \\ \qquad = \mathbf{obj}\,\alpha\,.\,\{\!|\, m : \tau_m[\alpha \mapsto \mathbf{obj}\,\alpha\,.\,\{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\}]^{m \in \mathcal{M}} \,|\!\}\end{array}} \quad (26)$$

C.5 Class hierarchy judgements

$$\frac{B \in \mathrm{dom}(H) \cup \{\mathbf{None}\}}{H \vdash B \lessdot B} \quad (27) \qquad \frac{C \in \mathrm{dom}(H)}{H \vdash C \lessdot \mathbf{None}} \quad (28)$$

$$\frac{H \vdash B_1 \lessdot B_2 \qquad H \vdash B_2 \lessdot B_3}{H \vdash B_1 \lessdot B_3} \quad (29) \qquad \frac{H(C) = B}{H \vdash C \lessdot B} \quad (30)$$

C.6 Subtyping judgements

$$\frac{\Gamma \vdash \tau = \tau'}{\Gamma \vdash \tau <: \tau'} \quad (31) \qquad \frac{\Gamma \vdash \tau <: \tau' \quad \Gamma \vdash \tau' <: \tau''}{\Gamma \vdash \tau <: \tau''} \quad (32)$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad \alpha \in \mathrm{dom}(\mathcal{S}\,\mathrm{of}\,\Gamma)}{\Gamma \vdash \alpha <: \mathcal{S}(\alpha)} \quad (33) \qquad \frac{\Gamma \vdash \tau_2 <: \tau_1 \quad \Gamma \vdash \tau_1' <: \tau_2'}{\Gamma \vdash \tau_1 \to \tau_1' <: \tau_2 \to \tau_2'} \quad (34)$$

$$\frac{\begin{array}{c}\mathcal{M}' \subseteq \mathcal{M} \quad \alpha' \notin \mathrm{FV}(\mathbf{obj}\,\alpha\,.\,\{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\}) \\ \forall m \in \mathcal{M}'\ \alpha \notin \mathrm{FV}(\tau_m') \text{ and } (\Gamma \cup \{\alpha'\}) \pm \{\alpha \mapsto \alpha'\} \vdash \tau_m <: \tau_m'\end{array}}{\Gamma \vdash \mathbf{obj}\,\alpha\,.\,\{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\} <: \mathbf{obj}\,\alpha'\,.\,\{\!|\, m : \tau'_m{}^{m \in \mathcal{M}'} \,|\!\}} \quad (35)$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad (\mathcal{C}\,\mathrm{of}\,\Gamma)(C) = (\tau)\,\{\!|\,\mathbf{inherits}\,B;\ m : \tau_m{}^{m \in \mathcal{M}} \,|\!\}}{\Gamma \vdash \#C <: \mathbf{obj}\,\alpha\,.\,\{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\}} \quad (36)$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau' <: \tau \quad H \vdash B \lessdot B' \quad \mathcal{M}' \subseteq \mathcal{M} \quad \forall m \in \mathcal{M}'\ \Gamma \vdash \tau_m = \tau_m' \\ \forall m \in (\mathcal{M} \setminus \mathcal{M}')\ \Gamma \vdash \tau_m \triangleright \mathbf{Ok}\end{array}}{\Gamma \vdash (\tau)\,\{\!|\,\mathbf{inherits}\,B;\ m : \tau_m{}^{m \in \mathcal{M}} \,|\!\} <: (\tau')\,\{\!|\,\mathbf{inherits}\,B';\ m : \tau'_m{}^{m \in \mathcal{M}'} \,|\!\}} \quad (37)$$

C.7 Typing judgements

As a notational convenience, we define the argument type of a class C in an environment Γ (written $\mathrm{ArgTy}(\Gamma, C)$) to be τ if $\Gamma(C) = (\tau)\,\{\!|\, \mathtt{inherits}\, B;\ m : \tau_m{}^{m\in\mathcal{M}} \,|\!\}$.

$$\frac{\Gamma \vdash d \triangleright \Gamma' \quad \Gamma' \vdash p \triangleright \Gamma'', \tau}{\Gamma \vdash d;\ p \triangleright \Gamma'', \tau} \quad (38) \qquad \frac{\Gamma \vdash e \triangleright \tau}{\Gamma \vdash e \triangleright \Gamma, \tau} \quad (39)$$

$$\frac{\begin{array}{c} C \notin \mathrm{dom}(\mathcal{C}\,\mathrm{of}\,\Gamma) \quad \Gamma \vdash \tau \triangleright \mathbf{Ok} \\ \Gamma' = \Gamma \pm \{C \mapsto (\tau)\,\{\!|\, \mathtt{inherits}\,\mathrm{None};\ m : \tau_m{}^{m\in\mathcal{M}} \,|\!\}\} \\ \forall m \in \mathcal{M}\ \Gamma' \pm \{\mathit{self} \mapsto \#C\} \vdash \mu_m \triangleright \tau'_m \qquad \Gamma' \vdash \tau'_m <: \tau_m \end{array}}{\Gamma \vdash \mathtt{class}\ C\ (x : \tau)\,\{\!|\ m = \mu_m{}^{m\in\mathcal{M}} \,|\!\} \triangleright \Gamma'} \quad (40)$$

$$\frac{\begin{array}{c} C \notin \mathrm{dom}(\mathcal{C}\,\mathrm{of}\,\Gamma) \\ (\mathcal{C}\,\mathrm{of}\,\Gamma)(C') = (\tau')\,\{\!|\, \mathtt{inherits}\, B';\ m : \tau'_m{}^{m\in\mathcal{M}'} \,|\!\} \\ \Gamma' = \Gamma \pm \{C \mapsto (\tau)\,\{\!|\, \mathtt{inherits}\, C';\ m : \tau'_m{}^{m\in(\mathcal{M}'\setminus\mathcal{M})}\ m : \tau_m{}^{m\in\mathcal{M}} \,|\!\}\} \\ \Gamma' \pm \{x \mapsto \tau\} \vdash e \triangleright \tau'' \quad \Gamma' \vdash \tau'' <: \tau' \\ \forall m \in (\mathcal{M} \cap \mathcal{M}')\ \Gamma' \vdash \tau_m <: \tau'_m \\ \forall m \in \mathcal{M}\ \Gamma' \pm \{\mathit{self} \mapsto \#C\} \vdash \mu_m \triangleright \tau''_m \qquad \Gamma' \vdash \tau''_m <: \tau_m \end{array}}{\Gamma \vdash \mathtt{class}\ C\ (x : \tau)\,\{\!|\, \mathtt{inherits}\, C'(e);\ m = \mu_m{}^{m\in\mathcal{M}} \,|\!\} \triangleright \Gamma'} \quad (41)$$

$$\frac{C \notin \mathrm{dom}(\mathcal{C}\,\mathrm{of}\,\Gamma) \quad \Gamma' = \Gamma \pm \{C \mapsto \sigma\} \quad \Gamma' \vdash (\mathcal{C}\,\mathrm{of}\,\Gamma)(C') <: \sigma}{\Gamma \vdash \mathtt{class}\ C : \sigma = C' \triangleright \Gamma'} \quad (42)$$

$$\frac{\Gamma \pm \{x \mapsto \tau\} \vdash e \triangleright \tau'}{\Gamma \vdash (x\ :\ \tau) \Rightarrow e \triangleright \tau \to \tau'} \quad (43) \qquad \frac{\Gamma \vdash \mathbf{Ok} \quad (\mathcal{V}\,\mathrm{of}\,\Gamma)(x) = \tau}{\Gamma \vdash x \triangleright \tau} \quad (44)$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad (\mathcal{V}\,\mathrm{of}\,\Gamma)(\mathtt{self}) = \tau}{\Gamma \vdash \mathtt{self} \triangleright \tau} \quad (45) \qquad \frac{\Gamma \pm \{x \mapsto \tau\} \vdash e \triangleright \tau'}{\Gamma \vdash \mathtt{fn}(x\ :\ \tau) \Rightarrow e \triangleright \tau \to \tau'} \quad (46)$$

$$\frac{\Gamma \vdash e \triangleright \tau' \to \tau \quad \Gamma \vdash e' \triangleright \tau'' \quad \Gamma \vdash \tau'' <: \tau'}{\Gamma \vdash e(e') \triangleright \tau} \quad (47)$$

$$\frac{\Gamma \vdash e \triangleright \tau' \quad \Gamma \vdash \tau' <: \mathrm{ArgTy}(\Gamma, C)}{\Gamma \vdash \mathtt{new}\ C(e) \triangleright \#C} \quad (48)$$

$$\frac{\Gamma \vdash e \triangleright \#C}{\Gamma \vdash e\,\mathtt{!state} \triangleright \mathrm{ArgTy}(\Gamma, C)} \quad (49) \qquad \frac{\Gamma \vdash e \triangleright \#C' \quad \mathcal{H}(\Gamma) \vdash C' \lessdot C}{\Gamma \vdash e\,@\,C \triangleright \#C} \quad (50)$$

$$\frac{\Gamma \vdash e \triangleright \tau \quad m \in \mathcal{M} \quad \Gamma \vdash \tau <: \mathtt{obj}\,\alpha\,\mathtt{.}\,\{\!|\, m : \tau_m{}^{m\in\mathcal{M}} \,|\!\}}{\Gamma \vdash e\,\mathtt{.}\,m \triangleright \tau_m[\alpha \mapsto \mathtt{obj}\,\alpha\,\mathtt{.}\,\{\!|\, m : \tau_m{}^{m\in\mathcal{M}} \,|\!\}]} \quad (51)$$

C.8 Typing rules for run-time forms

The typing rules for the run-time forms include rules 43 through 51 augmented with a dynamic class hierarchy H on the left-hand side of the turnstile. Rules 40 through 42 return an augmented class hierarchy in addition to an augmented static environment. In each case, the augmented hierarchy is just the old one extended with a mapping from the newly declared class name to the name of its parent (or $\mathbf{None}$ for base classes). In addition, rule 50 is replaced by rule 55. The remaining rules are given below.

$$\frac{\Gamma, H \vdash e \triangleright \tau' \quad \Gamma \vdash \tau' <: \mathrm{ArgTy}(\Gamma,\, C)}{\Gamma, H \vdash C(e) \triangleright \#C} \tag{52}$$

$$\frac{H(C) = B \quad \Gamma, H \vdash \mathit{obj} \triangleright \#B \quad \Gamma, H \vdash e \triangleright \tau' \quad \Gamma \vdash \tau' <: \mathrm{ArgTy}(\Gamma,\, C)}{\Gamma, H \vdash C :: \{\!|\, e;\ \mathit{obj} \,|\!\} \triangleright \#C} \tag{53}$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad \mathcal{K} \vdash \mathbf{Ok}}{\Gamma, H \vdash \mathbf{None} \triangleright \#\mathbf{None}} \tag{54}$$

$$\frac{\Gamma, H \vdash e \triangleright \#C' \quad H \vdash C' \lessdot C}{\Gamma, H \vdash e\,@\,C \triangleright \#C} \tag{55}$$

$$\frac{\Gamma, H \vdash \mathit{obj} \triangleright \#C' \quad H \vdash C' \lessdot C}{\Gamma, H \vdash \langle \mathit{obj},\ C\rangle \triangleright \#C} \tag{56}$$

$$\frac{\begin{array}{c} \sigma = (\tau)\, \{\!|\, \texttt{inherits}\, \mathrm{None};\ m : {\tau_m}^{m \in \mathcal{M}} \,|\!\} \\ \mathcal{M} = \mathrm{dom}(\phi) \qquad \mathrm{rng}(\phi) \subseteq \mathrm{dom}(\mu T) \\ \forall m \in\ \mathcal{M}\ \Gamma \pm \{\mathit{self} \mapsto \#C\}, H \vdash \mu T(\phi(m)) \triangleright \tau'_m \\ \Gamma \vdash \tau'_m <: \tau_m \end{array}}{\Gamma, H, C \vdash (\mathbf{None},\ \_,\ \mu T,\ \phi) \triangleright \sigma} \tag{57}$$

$$\frac{\begin{array}{c} \sigma = (\tau)\, \{\!|\, \texttt{inherits}\, C';\ m : {\tau_m}^{m \in \mathcal{M}} \,|\!\} \\ \Gamma(C') = (\tau')\, \{\!|\, \texttt{inherits}\, B';\ m : {\tau'_m}^{m \in \mathcal{M}'} \,|\!\} \\ \Gamma, H \vdash e \triangleright \tau \to \tau'' \quad \Gamma \vdash \tau'' <: \tau' \\ \mathcal{M} = \mathrm{dom}(\phi) \qquad \mathrm{rng}(\phi) \subseteq \mathrm{dom}(\mu T) \\ \forall m \in\ \mathcal{M}\ \Gamma \pm \{\mathit{self} \mapsto \#C\}, H \vdash \mu T(\phi(m)) \triangleright \tau''_m \\ \Gamma \vdash \tau''_m <: \tau_m \\ \forall m \in\ \mathcal{M} \cap \mathcal{M}'\ \Gamma \vdash \tau_m <: \tau'_m \end{array}}{\Gamma, H, C \vdash (C',\ e,\ \mu T,\ \phi) \triangleright \sigma} \tag{58}$$

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

D Proofs

The proof of soundness proceeds via a standard subject reduction argument, *i.e.*, we first prove that evaluation preserves types in Theorem 1 and then show that no well-typed program can get stuck in

Theorem 2. Consequently, well-typed programs either diverge or yield a value of the appropriate type (Theorem 3).

The subject reduction proof breaks into two pieces. In the first, we show that linking in a new class declaration (using Rule 4, 5, or 6) preserves the program type and produces a well-typed dynamic class environment. The challenge in this portion of the proof is to show that the new class environment is well-typed. These cases follow directly from an analysis of the reduction rules.

In the second part, we show that evaluating the program expression preserves its type. This portion follows the standard pattern for establishing subject reduction. In particular, we show in Lemma 1 that every closed expression can be factored uniquely into an evaluation context and a redex. Lemma 2 shows that evaluation does not introduce any free variables, so that if we start with a closed program, every intermediate program during reduction will be closed, and hence may be factored into an evaluation context and a redex. Lemma 3 proves that if we have typed an expression and that expression has been divided into an evaluation context and a redex, then we may also type the redex in the context that we typed the original expression. Furthermore, if we replace the redex with another expression with a better (in the sense of subtyping) type, then the new expression will have a better type than the original. This lemma means that we may prove the second piece of our subject reduction theorem by showing for each reduction rule (Rules 7 to 12) that the type of the reduct on the right-hand side of the rule is better than the type of the redex on the left-hand side. For Rule 7, this property follows from a standard substitution lemma (Lemma 4). For Rules 9 and 11, this property follows from Lemmas 5 and 6, respectively. The remaining cases are straightforward.

The proof of progress also breaks into two pieces. The first piece shows that well-typed programs do not get stuck during linking, while the second shows that well-typed expressions do not get stuck during evaluation. The linking piece proceeds by a case analysis on the rule used to link in the pending class. The expression piece proceeds by induction on the typing derivation used to infer that the expression was well-typed.

Type soundness follows from Theorem 1 and Theorem 2.

In the remainder of this section, we formally state the definitions and major lemmas and theorems necessary to prove our subject reduction progress, and type soundness theorems.

Definition 1 (Closure)

- We say that an expression form e or an object form obj is closed if it contains no free variables. We say that it is $self$*-closed* if it is closed and does not contain the special symbol $self$.
- A class declaration is closed if its method bodies contain no free variables and any initialization expression is $self$-closed after that expression is converted to a function from the class parameter.
- A program is closed if its class declarations are closed and its final expression is $self$-closed.
- A dynamic class environment $\mathcal{K}$ is closed if for all $C \in \text{dom}(\mathcal{K})$, if $\mathcal{K}(C) = (\_, e, \mu T, \_)$, then e is $self$-closed and for all $i \in \text{dom}(\mu T)$, $\mu T(i)$ is closed.

To state our unique decomposition lemma, we need to define expression redeces (er) and object redeces (or).

$$
\begin{array}{lcl}
er & ::= & w_1(w_2) \mid \mathbf{new}\ C(w) \mid w\,\mathtt{!state} \mid w\,@\,C \mid w\,\mathtt{.}\,m \mid \langle ov,\ C\rangle \\
or & ::= & C(w) \mid C :: \{\!\!\{\, w;\ ov \,\}\!\!\}
\end{array}
$$

In the following lemma, we use the symbol $\equiv$ to denote syntactic equality.

Lemma 1 (Unique Decomposition)

- If e is a *self*-closed expression, then either e is a value w or there exists a unique evaluation context E and redex er such that $e \equiv E[er]$.
- If obj is a *self*-closed raw object, then either obj is a value ov or there exists a unique evaluation context F and redex or such that $obj \equiv F[or]$.

The proof is by mutual induction on the structure of e and obj.

Lemma 2 (Evaluation Preserves Closure) If $\mathcal{K}$ and p are both closed and $\mathcal{K}, p \rightarrowtail \mathcal{K}', p'$, then $\mathcal{K}'$ and p' are both closed as well.

The proof is by a case analysis on the rule used to derive the reduction $\mathcal{K}, p \rightarrowtail \mathcal{K}', p'$.

Lemma 3 (Context) If we may derive the judgement $\Gamma, H \vdash G[e] \triangleright \tau$, where the meta-variable G ranges over either E or F, then for some type τ_g, we may also derive the judgement $\Gamma, H \vdash e \triangleright \tau_g$. Furthermore, if the judgement $\Gamma, H \vdash e' \triangleright \tau_g'$ is also derivable, where $\Gamma \vdash \tau_g' <: \tau_g$, then we may derive $\Gamma, H \vdash G[e'] \triangleright \tau'$, where $\Gamma \vdash \tau' <: \tau$.

The proof is by induction on the derivation of $\Gamma, H \vdash G[e] \triangleright \tau$.

Lemma 4 (Substitution) If we may derive the judgements $\Gamma \pm \{x \mapsto \tau_a\}, H \vdash e \triangleright \tau$ and $\Gamma, H \vdash e' \triangleright \tau_a'$, and $\Gamma \vdash \tau_a' <: \tau_a$, then we may also derive the judgement $\Gamma, H \vdash e[x \mapsto e'] \triangleright \tau'$, where $\Gamma \vdash \tau' <: \tau$.

The proof is by induction on the derivation of $\Gamma \pm \{x \mapsto \tau_a\}, H \vdash e \triangleright \tau$.

Lemma 5 (State Selection) If we may derive the judgements $\Gamma, H \vdash obj \triangleright \#C'$ and $C \vdash obj \rightsquigarrow_f w$, then we may derive the judgements $\Gamma, H \vdash w \triangleright \tau$ and $\Gamma \vdash \tau <: \mathrm{ArgTy}(\Gamma,\ C)$.

The proof is by induction on the derivation of $C \vdash obj \rightsquigarrow_f w$.

Lemma 6 (Method Selection) If we may derive the judgements $\Gamma \vdash \mathcal{K}$ and $\Gamma, \mathcal{H}(\mathcal{K}) \vdash w\,\mathtt{.}\,m \triangleright \tau$ and $\mathcal{K} \vdash w\,\mathtt{.}\,m \rightsquigarrow_m w'$, then we may also derive the judgements $\Gamma, \mathcal{H}(\mathcal{K}) \vdash w' \triangleright \tau'$ and $\Gamma \vdash \tau' <: \tau$.

This proof is the crux of the subject reduction theorem. Hence, we describe its structure in some detail. To have derived the judgement $\mathcal{K} \vdash w\,\mathtt{.}\,m \rightsquigarrow_m w'$, we must have determined that

- $w = \langle C :: \{\!\!\{\, w_s;\ ov \,\}\!\!\},\ C_v\rangle$,
- $\mathcal{K}(C) = (\_,\ \_,\ \mu\, T_C,\ \_)$,

- $\mathcal{K}(C_v) = (\_, \_, \_, \phi_v)$, and
- $w' = \mu T_C(\phi_v(m))[\mathit{self} \mapsto C :: \{\!|\, w_s;\ ov \,|\!\}]$

To have derived the judgement $\Gamma, \mathcal{H}(\mathcal{K}) \vdash w\,\mathtt{.}\,m \triangleright \tau$, we must have derived the judgements

- $\mathcal{H}(\mathcal{K}) \vdash C \lessdot C_v$,
- $\Gamma \vdash \mathtt{\#}C_v <: \mathtt{obj}\,\alpha\,\mathtt{.}\,\{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\}$, and
- $\Gamma, \mathcal{H}(\mathcal{K}) \vdash C :: \{\!|\, w_s;\ ov \,|\!\} \triangleright \mathtt{\#}C$

Furthermore, it must be the case that τ is syntactically equivalent to $\tau_m[\alpha \mapsto \mathtt{obj}\,\alpha\,\mathtt{.}\,\{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\}]$. Let $C_v \equiv C_0, \ldots, C_N \equiv C$ be the inheritance chain from C_v to C in $\mathcal{K}$. Let C_L be the class in the chain the furthest from C_v such that $\phi_v(m) = \phi_L(m)$, where ϕ_L is the dictionary that $\mathcal{K}$ associates with class C_L. Note that C_L can be equal to C_v, C, or both. By the construction of method tables during class linking, it must be the case that

$$\begin{array}{rcl} \mu T_C(\phi_v(m)) & = & \mu T_L(\phi_v(m)) \\ & = & \mu T_L(\phi_L(m)) \end{array}$$

Since $\Gamma \vdash \mathcal{K}$, we must have derived the judgement $\Gamma \pm \{\mathit{self} \mapsto \mathtt{\#}C_L\}, \mathcal{H}(\mathcal{K}) \vdash \mu T_L(\phi_L(m)) \triangleright \tau_m^{L''}$, where $\tau_m^{L''}$ is the type inferred by Rule 58 for method m in class C_L. By tracing the necessary subtyping and equality relations through Rules 36, 37, 58, and 59, we may determine that

$$\Gamma \vdash \tau_m^{L''} <: \tau_m[\alpha \mapsto \mathtt{obj}\,\alpha\,\mathtt{.}\,\{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\}]$$

By the construction of method suites, all occurrences of self in $\mu T_L(\phi_L(m))$ appear in the form $\mathit{self}\,@\,C'$, where C' is some ancestor class of C. Consequently, we may derive via a modified substitution lemma the following judgement:

$$\Gamma, \mathcal{H}(\mathcal{K}) \vdash \mu T_L(\phi_L(m))[\mathit{self} \mapsto C :: \{\!|\, w_s;\ ov \,|\!\}] \triangleright \tau_m^{L''}$$

which is what we needed to show.

Theorem 1 (Subject Reduction) If $\mathcal{K}$ is a closed dynamic class environment such that $\Gamma \vdash \mathcal{K}$ and for a closed program p we have $\Gamma, \mathcal{H}(\mathcal{K}) \vdash p \triangleright \Gamma_f, \tau_f$ and $\mathcal{K}, p \rightarrowtail \mathcal{K}', p'$, then there exists an environment Γ' and a type τ', such that

- $\Gamma', \mathcal{H}(\mathcal{K}') \vdash p' \triangleright \Gamma_f, \tau'$,
- $\Gamma' \vdash \tau' <: \tau_f$, and
- $\Gamma' \vdash \mathcal{K}'$

Furthermore, p' is closed and if p is an expression, then $\Gamma_f = \Gamma'$.

The proof proceeds by a case analysis on the reduction rule used to derive $\mathcal{K}, p \rightarrowtail \mathcal{K}', p'$.

Theorem 2 (Progress) If $\mathcal{K}$ is a closed dynamic class environment such that $\Gamma \vdash \mathcal{K}$ and for a program p we have $\Gamma, \mathcal{H}(\mathcal{K}) \vdash p \triangleright \Gamma_f, \tau_f$ with p and τ_f both closed, then either p is a value or there exists $\mathcal{K}'$ and p' such that $\mathcal{K}, p \rightarrowtail \mathcal{K}', p'$.

The proof is by induction on the derivation of $\Gamma, \mathcal{H}(\mathcal{K}) \vdash p \triangleright \Gamma_f, \tau_f$.

Definition 2 We say that a program p *yields* $\mathcal{K}', p'$ if $\emptyset, p \rightarrowtail \mathcal{K}', p'$ and there does *not* exist $\mathcal{K}'', p''$ such that $\mathcal{K}', p' \rightarrowtail \mathcal{K}'', p''$.

Theorem 3 (Type Soundness) If $\emptyset \vdash p \triangleright \Gamma_f, \tau_f$ and p yields $\mathcal{K}', p'$, then p' is a value w such that $\Gamma_f, \mathcal{H}(\mathcal{K}') \vdash w \triangleright \tau'$ with $\Gamma_f \vdash \tau' <: \tau_f$.

The proof is by induction on the number of steps needed to produce $\mathcal{K}', p'$.