A Core Calculus of Metaclasses

Sam Tobin-Hochstadt Eric Allen

Sun Microsystems Laboratories
Burlington MA 01803
<first>.<last>@sun.com

Abstract

Metaclasses provide a useful method of abstraction for programmers working with object-oriented languages, but they have not seen the formal exploration applied to more conventional object-oriented programming features. In order to elucidate the structure of metaclasses and their relationship with static typing, we present a core calculus for a nominally-typed object-oriented language with metaclasses and prove type soundness over this core. This calculus is presented as an adaptation of Featherweight GJ [13], and is powerful enough to capture metaclass relationships beyond what are expressible in common object-oriented languages, including arbitrary metaclass hierarchies and classes as values. We also explore the integration of object-oriented design patterns with metaclasses, and show how our calculus integrates several such patterns cleanly.

1 Introduction

One of the stated benefits of object-oriented languages is their ability to model aspects of the world in the object hierarchy. However, most such languages are unable to model many simple and common relationships. For example, consider the relationship between physical quantities, units of measurement, and physical dimensions. It is obvious that "length" is a dimension, and that "3 feet" is a length. However, modeling this relationship in a conventional language such as the Java™ Programming Language is impossible without attributing to some of these concepts properties which do not properly belong to them. It is natural to model `3 feet` as an instance of class `Length`. But if we were to define a class `Dimension` and define `Length` to be an instance of class `Dimension`, then `Length` would not be a class and so `3 feet` could not be an instance of `Length`. Alternatively, we might define `Length` to be a subclass of `Dimension`. But then we still cannot define `3 feet` to be an instance of `Length` because anything that is an instance of `Length` would also be an instance of class `Dimension` and `3 feet` is obviously not a dimension.

The fundamental limitation of conventional object-oriented languages is that they do not allow a single concept to serve both as a class and as an instance. In [18], Welty and Ferrucci lay out a comprehensive case that this limitation prevents conventional modeling languages from capturing many important aspects of the world. In their primary example, they show the difficulties in modeling a simple ontology including the notions `Species` and `Eagle`, while also capturing the relationship between `Eagle` and `Harry`, a particular eagle. One solution they consider and reject is to separate the notion of "eagle" into two concepts: `the Eagle` and `Eagle`, with the former an instance of `Species` and the latter a class whose instances are specific birds. This approach is inadequate because it fails to express a relationship between the notions `the Eagle` and `Eagle`.

In order to overcome these limitations in expressiveness, several object-oriented languages that are not statically typed, such as Smalltalk, Python, and Self, allow for more flexible class relationships [11], [14], [17]. In the case of Smalltalk and Python, classes are instances of metaclasses. Static members of a class are modeled as ordinary members of the corresponding metaclass. But traditional metaclass systems have placed important limitations on expressiveness. For example, in Smalltalk, the metaclass hierarchy is only two levels deep. If we want to represent the relationships "`A` is an instance of `B`", "`B` is an instance of `C`", and "`C` is an instance of `D`", we can only do this in Smalltalk if `D` is the special Smalltalk class `Metaclass`. This does not allow the multi-level hierarchies required to model the examples discussed above [11]. Self, on the other hand, is a prototype-based object-oriented language,

Submission for FOOL 2005

where there are no classes at all; object instantiation consists of cloning an existing instance. The members of the clone can be modified and added to at will, making static checking difficult.

An extremely expressive system of metaclasses, very similar to ours, was presented by Cointe in [6]. In his paper, he makes a number of persuasive arguments for the benefits of metaclasses. In response to complaints that the Smalltalk80 metaclass system was too complex, he suggested that a more powerful and flexible system, as he and we present, has the potential to be much simpler. He claims "The ObjVlisp model's primary advantage is uniformity." We agree, and believe that the treatment of metaclasses presented here unifies both disparate metaclass systems, and different ad-hoc mechanisms describing the properties of classes as objects in languages without metaclasses.

Cointe also quotes from [4]: "With respect to Simula, Smalltalk also abandons ... strong typing, allowing it to ... introduce the notion of meta-classes". In this paper, we show that such a compromise is not necessary, and that static typing can coexist with the flexibility of metaclasses.

We present and formalize a statically typed calculus with nominal subtyping for metaclasses where every class can serve as both an instance of a class and as a class of instances, and there is no limitation on the levels of nesting of metaclasses. Our work is motivated by [2], which presents a system for integrating the static checking of dimensions of physical quantities with a conventional object-oriented type system. In that system, metaclasses are used extensively to model the type relationships of dimensions. The notion of metaclass in that system is more general than that of languages such as Smalltalk, where each class has a unique metaclass that defines its class members. In this paper, we formalize the core of that system related to metaclasses and prove it sound. In the process, we discover several interesting properties about the structure that a nominal type system for metaclasses must take.

In the interests of economy and clarity of presentation, in the following discussion we restrict ourselves to simple examples, and do not delve in the the complexities of object-oriented analysis, or of dimension types. This is certainly not because we believe that metaclasses are most useful for the development of toy object-oriented programs. On the contrary, metaclasses are likely to show their value in large systems, both in terms of modeling and in the treatment of design patterns in section 3.

The remainder of this paper is organized as follows. In section 2 we present a core calculus for metaclasses we dub MCJ. We introduce the language and describe its key features, as well as some of the important design choices. The motivating examples from the introduction are presented in MCJ in section 2.3, demonstrating the benefits of metaclasses. We discuss in section 3 how the need for many object-oriented "design patterns" is obviated through the use of metaclasses. We then describe its formal properties in section 4. In section 4.2 we provide a formal semantics for the language, and in section 4.3 we prove a type soundness result. In section 5 we consider related work and in section 6 we describe conclusions and future directions.

2 MCJ

2.1 Overview

MCJ is a generically-typed object-oriented calculus based on Featherweight GJ that includes metaclasses. An MCJ program consists of a sequence of class definitions followed by a trailing expression. This trailing expression is evaluated in the context of the class table induced by the class definitions. Before discussing the formal specification of MCJ, we first give a high-level overview of the nature of MCJ class definitions.

Each class is an instance of a *metaclass*. A metaclass is either a user-defined class or class `Object`. We say that if a class `C` is an instance of a class `D`, then `C` is an *instance class* of `D`, and that `D` is the *immediate containing class*, or *kind*, of `C`. If class `E` is a superclass of `D` then we also say that `C` is an instance class of `E` and that `E` is a containing class of `C`.

There are two key distinguishing features of MCJ. First, a class (or an instantiation of a generic class) can be used as an expression. Second, all classes have both a superclass and a kind. Instances of a class inherit behavior from the superclass. The class itself, when used in an expression context, inherits behavior from its kind.

A class definition consists of a header plus a collection of fields and methods associated with the class. The header names the class and specifies the type parameters, as well as the superclass and the kind. For example, the following header declares a class `C` with superclass `Object` and kind `D` and no type parameters:

```
class C kind D extends Object {...}
```

Each class may have both class members and instance members. Class members define the behavior of the class when used as an instance. Instance members define the behavior of instances of a class. In the abstract syntax presented in this paper, class members are distinguished from instance members by order, and with the keyword `class` [1]. Members of a class definition

[1]This syntax is not ambiguous since MCJ has no inner classes.

are laid out as follows: first, class fields are defined, followed by class methods, then instance fields and finally instance methods. For example, the following class definition includes a class method `m` and an instance method `n`:

```
class C kind Object extends Object {
   class Object m() {...}
   Object n() {...}
}
```

Class fields and methods bear a resemblance to static fields and methods, but there are important differences. For a given class `C`, instance members of `C`'s kind are inherited as class members of `C`. For example, a class method `m` in `C` with the same name as an instance method of `C`'s kind overrides the definition of `m` in the kind (and it must have the same signature as the overridden method). Note that if an instance class `C` of type `T` is assigned to a variable `v` of type `T`, references to the instance members of `v` refer to the class members of `C`. As in [2], a class is allowed to define both a static method and an instance method of the same name. We prevent ambiguity in member references by requiring that all references to class members in MCJ must explicitly denote the receiver.

Superclasses behave as in Featherweight GJ, providing implementation inheritance of instance behavior and subtyping. A class, when used as an expression, is an instance of its kind. If a method is invoked on a class `C`, first the class methods of `C` are examined. If the method is found there, it is invoked. Otherwise, the *instance* methods of the kind `D` of `C` are searched, and if it is found there, the method is invoked. Otherwise, the superclass hierarchy of `D` is examined as it is for an ordinary instance of `D`.

For example, consider the following class definitions:

```
class A kind Object extends Object {...}
class B kind Object extends A {...}
class C kind B extends Object {...}
```

To resolve the method call `C.m()` first the class methods of `C` are examined, and if a match is found, it is invoked. Then the *instance* methods of `B` and then `A` are examined.

If the method call were instead `b.m()`, where `b` denotes an instance of `B`, then the instance methods of first `B` then `A` would be examined.

The generic type system of MCJ is similar to the generic type system in Featherweight GJ. However, as in [5, 2], we allow for type variables to occur in type-dependent contexts such as casts, preventing the use of type erasure as an implementation technique. In addition, the receiver of a reference to a class member may be a type variable. Unlike the system presented in [1], MCJ does not support first-class genericity because a naked type variable must not appear in the `extends` clause or `kind` clause of a class definition and `new` expressions on naked type variables are not supported. Also, polymorphic methods (which are orthogonal to the features we explore) are not supported.

Because classes can be used as expressions, we need a bound on the behavior of classes when used in expression contexts. Therefore, we place two bounds on every type variable `T` in the header of the class definition. The first bound on `T` is a bound on the kind of an instantiation of `T`; the second bound is a bound on the superclass of `T`. For example, the following class header declares a class `C` with one type parameter `T` that must be instantiated with a type of kind `D` that is a subtype of `E`:

```
class C<T extends (D,E)>
   kind Object
   extends Object {...}
```

2.2 Key Design Points

Private Constructors One benefit of a metaclass system is that constructors no longer need to play such a central role in the language. Class instances exist at the beginning of a program execution; they need not be constructed. Non-class instances are naturally constructed with class (factory) methods [10]; in this way a class instance can be passed as a type parameter and factory methods can be called on the class instance to construct new instances without stipulating the class of the constructed instance. This flexibility is similar to that provided with first-class genericity [5, 1], but obviates the need for `with` clauses because the bound on the kind of a type variable stipulates what factory methods can be called on it. A class can define class methods with arbitrary signatures that return new instances of the class, but each such method must ultimately includes a `new` expression, which can occur only within the scope of the class whose instance it returns. A `new` expression looks like a call to method named `new`, that takes a single argument for each type parameter and each instance field of the syntactically enclosing class. The result of a `new` expression is a new instance of the enclosing class whose type parameters are instantiated with, and whose fields are initialized with, the given arguments. In our formal semantics, `new` expressions are annotated with the name of the enclosing class. These annotations need not be added by a programmer; they could be added easily by a straightforward syntactic preprocessing over the program, since they can be determined merely by the lexical scopes of the `new` expressions.

All uses of constructors in Featherweight GJ [13] are macro-expressible [9] in MCJ, that is, they can be expressed via local transformations. Of course, not all Featherweight GJ programs can be expressed in MCJ, because of the differences in the type system. Namely, MCJ does not include polymorphic methods.

Metaclasses allow other kinds of flexibility in object creation, as discussed in section 3.

Fields and Initialization To ensure type safety, we must have an initial value for every field, or prevent fields from being used before they are initialized. For instance fields of objects created with a constructor, this is achieved by requiring that the constructor have an argument for every instance field. Combined with the solution for flexibility in object creation outlined above, this allows us the simplicity in semantics of FGJ, and the flexibility of more general constructors.

However, classes which are themselves instances of other classes have fields as well, and these fields must also be initialized before they are used. These fields include those instance fields of the kind which become class fields of the new class. Our solution is to require that *all class fields, including those obtained by inheritance, must be redeclared* and provided with values.

typeOf Given that class references can be used as expressions, it is natural to ask: what is the type of a class reference? In the world of the λ-calculus, such types are known as kinds. In MCJ, however, the kind of a class does not capture all of the properties of the class as a value. For example, a class may add new class fields or class methods which are not present in the kind. Therefore, each class freely generates a new type, which is the type of the class considered as a value. We represent this type with the `typeOf` operator, which is produces a type that is not also a class. Since this type is not a class, it cannot be the superclass or kind of another class, and it cannot serve as the instantiation of a generic type parameter.

These new types can create complex relationships between classes and types. For example, the following class headers:

```
class A() kind Object extends Object {...}
class B() kind A extends Object {...}
```

induce the following type relationships:

$$\begin{array}{c} \texttt{B} : \texttt{typeOf[B]} \\ \texttt{typeOf[B]} <: \texttt{A} \\ \texttt{B instanceof A} \end{array}$$

The name `typeOf` has been used in other contexts to refer to a runtime operation that determines the dynamic type of a object. However, despite the name similarity, these functions bear no relationship to the operator described here. In MCJ, an application of `typeOf` is a static type reference.

Non-transitivity The standard subclassing relationship, like subtyping, is transitive. That is, if `B` is subclass of `C` and `C` is a subclass of `D`, then `B` is a subclass of `D`. However, this relationship does not hold for `instanceof` relationships. Therefore, if if `B` is an instance of `C` and `C` is an instance of `D`, then no judgment about the relationship of `B` to `D` can be inferred.

Cyclic class hierarchies One property of metaclasses that we have not yet discussed is that cycles can occur among instance class relationships. The simplest such cycle is:

$$\texttt{class C<>} : \texttt{C<>} \lhd \texttt{Object}\{\}$$

Although this pathology appears to be dangerously close to Russell's paradox, it does not lead to inconsistencies. Class `C` is an instance of itself. It can be used in any context where one of its instances can be used. `C` can be thought of as a self-replicating value. This class hierarchy causes no problems for method lookup because method resolution never proceeds through more than a single containing class to an instance class. Therefore, we see no reason to disallow it.

2.3 Examples, Reprised

Having seen the outlines of MCJ, we return to the motivational examples discussed earlier. First, the relationships between dimensions is easy to capture. [2] We define

```
class Dimension kind Object extends Object { ... }
class Length kind Dimension extends Object { ... }
class Meter kind Length extends Object { ... }
```

Here `Meter` is a singleton class (there is only one Meter, in the Platonic sense). We have easily expressed the desired type relationships, and can statically check program invariants that rely on dimensional relationships.

The example from [18] is also easily expressed:

```
class Species kind Object extends Object { ... }
class Eagle kind Species extends Object {
      class Eagle make(String name) { ... }
}
Eagle harry = Eagle.make("Harry")
```

[2] Of course, dimensions have many subtleties, not captured here. All we present is the essence of the type relationships.

Here we simply create an object to represent our concrete instance of an Eagle. The naive implementation of the above code in a conventional OO language would look something like this:

```
class Species extends Object { ... }
class TheEagle extends Species { ... }
class Eagle extends Object { ... }
Eagle harry = new Eagle("Harry")
```

There are at least two substantial problems with this code. First, `TheEagle` and `Eagle` have no relationship to each other in the type system. One is a singleton class, representing a particular species, and one is a name for a set of objects, being the members of that species. The fundamental nature of species has a two-level containment relationship, which the type system is failing to express. This means that our type checker cannot determine what we want, and cannot help us avoid mistakes.

Another problem, which is in some ways just a symptom of the first problem, but which bears special attention, is the use of generic types. Imagine the following method, where T is a type parameter in scope:[3]

```
foo(T x) { ... }
```

If we want to perform the call `foo(harry)`, then the bound on `T` must be either `Object`, in which case `foo` must have no knowledge of the object, or Eagle, in which case the function cannot handle multiple `Species`. The "solution" in conventional languages is to use a bound of `Object` and insert a downcast, which fails at runtime if the wrong argument is passed. In MCJ `T` can be bound by kind `Species`, and instantiated with `Eagle`, allowing the original call to be type-checked. This gives the programmer both safety and expressiveness.

3 Design Patterns as Language Features

Design Patterns, as exemplified in [10], have had a very substantial impact on the world of object-oriented programming and beyond. Design patterns allow programmers to increase the flexibility and abstraction of their software designs. However, few design patterns have been integrated into programming languages.

One of the advantages of our metaclass framework is that it allows clean expression of several patterns in the language, rather than adding additional abstraction techniques on top of the language. For example, a number of the classic design patterns deal with object creation, as discussed above in section 2.2. It is clear that the concept of a Factory method is easily (and necessarily) expressed in MCJ. However, a number of other patterns also find easy expression. The Abstract Factory pattern is a simple application of inheritance in the metaclass hierarchy. The Prototype pattern is, as mentioned in the Design Patterns book, supported in languages with metaclasses. Finally, there is no need for the Singleton pattern, since the class itself can serve as a singleton. While this is possible in the Java Programming Language and in C++, it is undesirable since the static behavior that would need to be used cannot inherit from another class. In fact, the inflexibility of class or static operations is one of the rationales cited for the Singleton pattern.

Seeing that a number of patterns can be expressed quite simply in MCJ, the question arises: is the simple expression of these patterns a benefit? We argue that it is. The existence of design patterns is a sign of shortcomings in language design. Fundamentally, a design pattern is an abstraction that cannot be expressed in the language. For example, the Singleton and Visitor patterns are abstractions that have obvious invariants, but they cannot be expressed directly and they require complex cooperation from many parts of the system.

Sophisticated macro systems, such as those found in Common Lisp [16] or Scheme [8] allow for expression of many forms of abstraction, as do extremely flexible object systems such as CLOS. However, these solutions add significantly to what a programmer must understand in order to use the language productively.

An alternative solution explored in this paper is to provide additional flexibility in the language, which does not allow for arbitrary additional abstraction, but instead makes the abstractions commonly needed easy to express. One does not need the full power of macros to encode the Factory pattern. In MCJ, we have presented a language where several kinds of patterns are easy to express, without adding an additional language on top of the original.

4 Formal Specification of MCJ

Having outlined the motivation for metaclasses, and the basics of their use, we now turn to a formal exposition of the syntax and semantics of MCJ, followed by an outline of the proof of soundness for the calculus.

4.1 Syntax

The syntax of MCJ is given in Figure 1. When describing the formal semantics of MCJ, we use the following metavariables:

- Expressions or mappings: `e`, `d`, `r`

[3] Generic methods would be straightforward to add to MCJ, but we excluded them to simplify the presentation of the type system, and since they do not add complexity to the proofs.

CL	::=	$\mathtt{class\ C\langle\overline{X} \triangleleft (\overline{T},\overline{T})\rangle : A \triangleleft A\ \{class\ \overline{T}\ \overline{f}\ \overline{e};\ \overline{M};\ \overline{T}\ \overline{f};\ \overline{N};\}}$	class declaration
N	::=	$\mathtt{T\ m\ (\overline{T}\ \overline{x})\ \{return\ e;\}}$	method declaration
M	::=	$\mathtt{class\ T\ m\ (\overline{T}\ \overline{x})\ \{return\ e;\}}$	class method declaration
e	::=	$\mathtt{x}$	variable reference
	|	$\mathtt{e.f}$	field access
	|	$\mathtt{e.m(\overline{e})}$	method invocation
	|	$\mathtt{new_C\langle\overline{R}\rangle(\overline{e})}$	instance creation
	|	$\mathtt{(T)e}$	cast
	|	$\mathtt{R}$	type reference
	|	$\Phi_\mathtt{G}$	mapping
T	::=	$\mathtt{R}$	non-`typeOf` type
	|	$\mathtt{typeOf[R]}$	`typeOf` application
R	::=	$\mathtt{X}$	type variable
	|	$\mathtt{A}$	
A	::=	$\mathtt{C\langle\overline{R}\rangle}$	type application
	|	$\mathtt{Object}$	

Figure 1: MCJ Syntax

- Field names: `f`, `g`
- Variables: `x`
- Method names: `m`
- Values: `v`
- Method declarations: `M`, `N`
- Type names: `C`, `D`
- Types: `I`, `J`, `K`, `T`, `U`, `V`, `W`
- Non-`typeOf` Types: `O`, `P`, `Q`, `R`, `S`
- Types that are either `Object` or a type application: `A`, `B`
- Ground types (contain no type variables): `G`
- Type variables: `X`, `Y`, `Z`
- Mappings (field name $\mapsto$ value): Φ

As in Featherweight Java and many other works, $\overline{\mathtt{x}}$ stands for a possibly-empty sequence of `x`. $[\overline{\mathtt{X}} \mapsto \overline{\mathtt{S}}]$ denotes a substitution of the $\overline{\mathtt{S}}$ for the $\overline{\mathtt{X}}$, which can be applied to either an expression or a type, and which can substitute either type or expression variables.

A number of symbols are used to abbreviate keywords: $\triangleleft$ stands for `extends` and : stands for `kind`. Also @ represents concatenation of sequences of syntactic constructs. Finally, $CT(\mathtt{T})$ is a lookup in the class table for the definition of `T`.

A number of restrictions on MCJ programs are implicit in the formal rules. First, we assume that all sequences of methods and fields are free of duplicates. Second, there is an implicit well-formedness constraint on programs that no class be a superclass of itself, either directly or indirectly (however, as discussed below, cycles in the kind hierarchy are allowed). Third, we assume that `this` is never used as the name of a variable, method or field. We also take `Object` to be a distinguished member of the hierarchy, with no specific definition, which does not have a superclass or a kind.

For example, the simplest MCJ class is:

$\mathtt{class\ C\langle\rangle : Object \triangleleft Object\{\}}$

This is a class named `C`, with no type arguments, with kind `Object` and superclass `Object`, which has no fields or methods whatsoever. In examples that follow, we omit empty `<>`. Note that both the kind and the superclass are `Object`. MCJ does not have a distinguished class `Class`. Interestingly, we determined that a class `Class` would need no consideration from the type system. Since both `Class` and `Object` would sit atop the hierarchy with no contents, we determined there was no need to include both of them. In a more substantial language, where `Object` might have methods or fields, there might be a use for a distinguished `Class` class. However, it would not need any special treatment by the type system.

A more complicated MCJ class is:[4]

```
class Pair<A extends (Object, Object),
           B extends (Object, Object)>
  kind Object extends Object
```

[4]Here we use `extends` for $\triangleleft$ and `kind` for :.

```
{
 class Pair<A,B> make(A a, B b){ new(A,B) (a,b) }
 A fst;
 B snd;
 Pair<B,B> setfst(B b){
     return new <B,B> (b, this.snd);
 };
}
```

This class specifies a polymorphic pair data structure. It also demonstrates a number of important MCJ features. First, we see two different kinds of methods. The `make` method creates a new instance of `Pair`. This method contains a `new` expression, which initializes the instance fields of the class positionally, so that the first argument to `make` becomes `fst`, and the second becomes `snd`. Also, note that we must mention the type parameters of the newly created instance.

Second, we have an instance method, `setfst`, which creates a new pair with the old second element, and a new first element which has the same type as the second element.

Finally, adding the following to the above definition of `Pair` gives a full MCJ program:

```
class O kind Object extends Object
{
 class O make(){return new () ()};
}

Pair<O,O>.make(O.make(), O.make()).fst
```

This program, which creates two instances of `O`, then creates a `Pair` to hold them both, and finally selects one of the two to become the value of the program, demonstrates three of the expression forms in MCJ. `Pair<O,O>` is a type, which, by itself, can be a value. `O.make()` is a method invocation (here with a class, not an instance, as the receiver). The entire expression is a field access, of the `fst` instance field.

Other kinds of expressions are seen in the body of the `Pair` class. `new<A,B> (a,b)` is a new expression, which creates an instance of the enclosing class (here `Pair`). In the body of `make` is a reference to the variable `b`, which has the obvious value. The final directly expressible expression is the cast, which works as follows: `(Object)O.make()` is an expression of type `Object`that evaluates to an instance of `O`.

With an understanding of expressions, we can examine the rest of the `Pair` class. There are two methods: a class method (`make`) and an instance method (`setfst`). There are also two fields, both of which are instance fields and initialized the `new` expressions. In MCJ, a `new` expression is different from those in Featherweight GJ; each `new` expression evaluates to an instance of the syntactically enclosing class.[5] Had there been additional fields in the superclass of `Pair`, it would have been necessary to initialize them in the `new` expression as well. `new` expressions are explained in detail in section 2.2.

4.2 Semantics

There are two forms that a value can take in MCJ: that of an instance class, and that of a conventional (non-class) value. If we were to distinguish these two forms of value, we would significantly increase the number of rules necessary to describe our semantics because every rule referring to a value would have to be written twice. To avoid this complexity, we introduce a special *mapping construct* (similar to a record value) to denote the results of computations.[6] A mapping takes field names to expressions, and is annotated with a ground type. Mappings from a sequence of fields $\overline{\mathtt{f}}$ to a sequence of expressions $\overline{\mathtt{e}}$ with type `G` are written $\{\overline{\mathtt{f}} \mapsto \overline{\mathtt{e}}\}_{\mathtt{G}}$. A mapping denoting an instance class `C` consists of a sequence of class fields mapped to a sequence of expressions and is annotated with the type `typeOf(C)`. Note that the right hand sides of such maps need not always be values, and thus computation can take place inside of a mappings. Mappings are not available to the programmer, and thus can only be created by operation of the reduction rules. Therefore, unlike Featherweight Java, our reductions do not operate entirely in the expressible syntax of the language. We use the metavariable Φ to range over mappings. When we need to refer to the type annotation of a mapping explicitly, the metavariable is written $\Phi_{\mathtt{G}}$.

The semantics are given in figures 2, 3 ,4, and 5 with auxiliary functions given in figure 6.

4.2.1 Typing

Rules governing the typing of MCJ programs are given in figures 2, 3 and 4. The $bound_\Delta$ function is defined as follows:

$$bound_\Delta(\mathtt{X}) = \Delta(\mathtt{X})$$
$$bound_\Delta(\mathtt{typeOf[X]}) = \Delta(\mathtt{typeOf[X]})$$
$$bound_\Delta(\mathtt{S}) = \mathtt{S}$$

This function maps type variables and `typeOf` applied to type variables to their bounds, and leaves others unchanged.

The metavariables Δ and Γ range over bounds environments, written $\overline{\mathtt{X}} \triangleleft \overline{\mathtt{S}}$ and type environments, written

[5] In the grammar of figure 1, `new` expressions are annotated with this enclosing class. In the examples, this redundant information is elided.

[6] This simplification was suggested by Jan-Willem Maessen in response to an earlier draft of the MCJ semantics.

$\overline{\mathtt{x}} : \overline{\mathtt{T}}$ respectively. A bounds environment contains two bounds for each type variable, so that if $\Delta = \mathtt{X} <: (\mathtt{A}, \mathtt{B})$, then $\Delta(\mathtt{X}) = \mathtt{A}$ and $\Delta(\mathtt{typeOf[X]}) = \mathtt{B}$.

The notation $\mathtt{A} <: \mathtt{B}$ means A is a subtype of B. Subtyping judgments are made in the context of a bounds environment, which relates a type to the declared bound of that type from the class header.

Type judgments are of the form $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$, which states that in bounds environment Δ and type environment Γ, expression e has type T. Type judgments are not transitive in the way that `instanceof` relationships are with respect to subtyping: if $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$, and $\Delta \vdash \mathtt{T} <: \mathtt{S}$, it is not necessarily the case that $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{S}$. This is important, since our proofs depend on having unique derivations of a given typing judgment. We represent empty environments with $\emptyset$ and we abbreviate judgments of the form $\emptyset; \emptyset \vdash \mathtt{e} : \mathtt{T}$ and $\emptyset \vdash \mathtt{T} <: \mathtt{S}$ as $\mathtt{e} : \mathtt{T}$ and $\mathtt{T} <: \mathtt{S}$ respectively.

We now discuss several non-obvious aspects of our type system resulting from the need to statically check uses of metaclasses.

Casts and Mappings We follow [1] in typing all casts as statically correct, so as to avoid the complications of "stupid casts". Mappings, which are not expressible in the source language, are typed merely by a well-formedness constraint.

4.2.2 Well-Formedness

There are three kinds of well-formedness constraints on MCJ programs. First, type well-formedness, written "$\Delta \vdash \mathtt{T}$ ok", states that type T refers to a defined class, and that if it is a type application, the arguments satisfy the bounds.

More important are the class and method well-formedness constraints, which together determine if a class table is well-formed, and thus part of a legal MCJ program. Method well-formedness is a judgment of the form "M ok in G", where M is a method. We make use of the latter portion (G) of this judgment as a second argument, allowing us to use this rule to check both instance and class methods. Method well-formedness consists mostly in fitting the body to the return type, and checking for invalid overrides.

Class well-formedness involves all of the above checks. All methods must be well-formed, and all class fields must have correct initialization expressions. Further, class fields must contain the instance fields of the kind. Finally, all `new` expressions must have the correct class name annotation.

4.2.3 Evaluation

The evaluation rules for MCJ are given in figure 5. The evaluation relation, written $\mathtt{e} \rightarrow \mathtt{e}'$, states that e transitions to $\mathtt{e}'$ in one step. The reflexive transitive closure of this relation is written $\mathtt{e} \rightarrow^* \mathtt{e}'$.

Bad Casts There is one way that evaluation in MCJ can get stuck: the bad cast. This problem occurs for all languages that allow static downcasts. A expression e is a bad cast iff $\mathtt{e} = (\mathtt{S})\Phi_{\mathtt{T}}$ where S is not a supertype of T.

The evaluation relation given here is non-deterministic. For example, no order is prescribed for evaluating the arguments to a `new` expression or method call, or for evaluating the initializers for class fields. There are also a number of places where either congruence or reduction rules can be applied. Therefore, in the presence of non-termination or bad casts, the results may differ depending on evaluation order. However, confluence can be regained simply by requiring that [C-Mapping] is applied whenever possible. This restriction, combined with the requirement in the premise of [R-New] that the arguments be values, ensures that every program with an error or non-termination will either cause an error, or fail to terminate.

4.3 Type Soundness

With the above definitions, we are now able to turn to a proof of type soundness. Given the simplicity we are able to achieve in the definitions, the proof is not significantly more complicated than the soundness proof given for Featherweight GJ [13]. However, there are a number of significant lemmas, of which we list the most important:

Lemma 1 (Fields are Preserved by Subtypes) *If* $\mathit{fields}(\mathtt{U}) = \overline{\mathtt{F}}$ *and* $\emptyset \vdash \mathtt{V} <: \mathtt{U}$ *then* $\exists\ \overline{\mathtt{G}}$ *where* $\mathit{fields}(\mathtt{V}) = \overline{\mathtt{F}} \mathbin{@} \overline{\mathtt{G}}$.

Proof We prove this by induction over the derivation for $\mathit{fields}(\mathtt{V})$.

Case $\mathtt{V} = \mathtt{Object}$**:** Then $\mathtt{U} = \mathtt{Object}$ and $\emptyset = \emptyset \mathbin{@} \emptyset$.

Case $\mathtt{V} = \mathtt{C}\texttt{<}\overline{\mathtt{R}}\texttt{>}$**:** Then $\mathit{CT}(\mathtt{C}) = \mathtt{class\ C}\texttt{<}\overline{\mathtt{X}} \lhd (\overline{\mathtt{I}}, \overline{\mathtt{J}})\texttt{>} : \mathtt{B} \lhd \mathtt{A}\{...\overline{\mathtt{H}}...\}$

Continue by induction on the derivation of $\emptyset \vdash \mathtt{V} <: \mathtt{U}$. The only interesting cases are [S-Super] and [S-Trans].

Subcase [S-Super]: $[\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}]\mathtt{A} = \mathtt{U}$. Thus $\overline{\mathtt{G}} = \overline{\mathtt{H}}$.

$$\Delta \vdash \mathtt{T} <: \mathtt{T}\ [\textsc{S-Reflex}] \qquad \frac{\mathtt{X} \lhd \mathtt{T} \in \Delta}{\Delta \vdash \mathtt{X} <: \mathtt{T}}\ [\textsc{S-Bound}] \qquad \frac{\Delta \vdash \mathtt{V} <: \mathtt{T} \quad \Delta \vdash \mathtt{T} <: \mathtt{U}}{\Delta \vdash \mathtt{V} <: \mathtt{U}}\ [\textsc{S-Trans}]$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \lhd (\overline{\mathtt{I}}, \overline{\mathtt{J}})\mathtt{>} : \mathtt{B} \lhd \mathtt{A}\ \{...\}}{\Delta \vdash \mathtt{typeOf[C<}\overline{\mathtt{S}}\mathtt{>]} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{S}}]\mathtt{B}}\ [\textsc{S-Kind}] \qquad \frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \lhd (\overline{\mathtt{I}}, \overline{\mathtt{J}})\mathtt{>} : \mathtt{B} \lhd \mathtt{A}\ \{...\}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{S}}\mathtt{>} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{S}}]\mathtt{A}}\ [\textsc{S-Super}]$$

$$\Delta \vdash \mathtt{Object}\ \text{ok}\ [\textsc{WF-Object}] \qquad \frac{\Delta \vdash \mathtt{X} <: \mathtt{T}}{\Delta \vdash \mathtt{X}\ \text{ok}}\ [\textsc{WF-Var}] \qquad \frac{\Delta \vdash \mathtt{T}\ \text{ok}}{\Delta \vdash \mathtt{typeOf[T]}\ \text{ok}}\ [\textsc{WF-Typeof}]$$

$$\frac{\begin{array}{c} CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \lhd (\overline{\mathtt{I}}, \overline{\mathtt{J}})\mathtt{>} : \mathtt{B} \lhd \mathtt{A}\ \{...\} \\ \Delta \vdash \overline{\mathtt{S}} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{S}}]\overline{\mathtt{I}} \quad \Delta \vdash \mathtt{typeOf[}\overline{\mathtt{S}}\mathtt{]} <: [\overline{\mathtt{X}} \mapsto \overline{\mathtt{S}}]\overline{\mathtt{J}} \quad \Delta \vdash \overline{\mathtt{S}}\ \text{ok} \end{array}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{S}}\mathtt{>}\ \text{ok}}\ [\textsc{WF-Class}]$$

Figure 2: Subtyping and Well-Formed Types

$$\Delta;\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})\ [\textsc{T-Var}] \qquad \frac{\Delta \vdash \mathtt{T}\ \text{ok} \quad \Delta;\Gamma \vdash \mathtt{e} : \mathtt{U}}{\Delta;\Gamma \vdash \mathtt{(T)e} : \mathtt{T}}\ [\textsc{T-Cast}] \qquad \frac{\Delta \vdash \mathtt{S}\ \text{ok}}{\Delta;\Gamma \vdash \mathtt{S} : \mathtt{typeOf[S]}}\ [\textsc{T-Class}]$$

$$\frac{\begin{array}{c} \Delta;\Gamma \vdash \mathtt{e} : \mathtt{U} \\ \mathit{fields}(\mathit{bound}_\Delta(\mathtt{U})) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \end{array}}{\Delta;\Gamma \vdash \mathtt{e.f}_i : \mathtt{T}_i}\ [\textsc{T-Field}] \qquad \frac{\begin{array}{c} \Delta \vdash \mathtt{C<}\overline{\mathtt{S}}\mathtt{>}\ \text{ok} \quad \mathit{fields}(\mathtt{C<}\overline{\mathtt{S}}\mathtt{>}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \\ \Delta \vdash \overline{\mathtt{U}} <: \overline{\mathtt{T}} \quad \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \end{array}}{\Delta;\Gamma \vdash \mathtt{new}_{\mathtt{C}}\mathtt{<}\overline{\mathtt{S}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{C<}\overline{\mathtt{S}}\mathtt{>}}\ [\textsc{T-New}]$$

$$\frac{\begin{array}{c} \mathit{fields}(\mathtt{G}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \\ \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \quad \Delta \vdash \overline{\mathtt{S}} <: \overline{\mathtt{T}} \end{array}}{\Delta;\Gamma \vdash \{\overline{\mathtt{f}} \mapsto \overline{\mathtt{e}}\}_{\mathtt{G}} : \mathtt{G}}\ [\textsc{T-Mapping}] \qquad \frac{\begin{array}{c} \mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{W})) = \overline{\mathtt{U}} \to \mathtt{T} \\ \Delta;\Gamma \vdash \mathtt{r} : \mathtt{W} \quad \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{V}} \quad \Delta \vdash \overline{\mathtt{V}} <: \overline{\mathtt{U}} \end{array}}{\Delta;\Gamma \vdash \mathtt{r.m(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{T}}\ [\textsc{T-Invk}]$$

Figure 3: Expression Typing

$$\frac{\begin{array}{c} \mathit{params}(\mathtt{G}) = \overline{\mathtt{X}} \sqsubset (\overline{\mathtt{I}}, \overline{\mathtt{J}}) \\ \Delta = \{\overline{\mathtt{X}} \lhd \overline{\mathtt{I}}, \mathtt{typeOf[}\overline{\mathtt{X}}\mathtt{]} \lhd \overline{\mathtt{J}}\} \quad \Gamma = \{\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{G}\} \\ \Delta \vdash \overline{\mathtt{T}}\ \text{ok} \quad \Delta \vdash \mathtt{U}\ \text{ok} \quad \mathit{override}(\mathtt{m}, \mathit{super}(\mathtt{G}), \overline{\mathtt{T}} \to \mathtt{U}) \\ \Delta;\Gamma \vdash \mathtt{e} : \mathtt{W} \quad \Delta \vdash \mathtt{W} <: \mathtt{U} \end{array}}{\mathtt{U\ m\ (}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)}\ \{\mathtt{return\ e;}\}\ \text{ok in}\ \mathtt{G}}\ [\textsc{WF-Method}]$$

$$\frac{\begin{array}{c} \overline{\mathtt{M}}\ \text{ok in}\ \mathtt{typeOf[C<}\overline{\mathtt{X}}\mathtt{>]} \quad \overline{\mathtt{N}}\ \text{ok in}\ \mathtt{C<}\overline{\mathtt{X}}\mathtt{>} \quad \Delta = \{\overline{\mathtt{X}} \lhd \overline{\mathtt{I}}, \mathtt{typeOf[}\overline{\mathtt{X}}\mathtt{]} \lhd \overline{\mathtt{J}}\} \\ \Delta \vdash \mathtt{B}\ \text{ok} \quad \Delta \vdash \mathtt{A}\ \text{ok} \quad \Delta \vdash \overline{\mathtt{I}}\ \text{ok} \quad \Delta \vdash \overline{\mathtt{J}}\ \text{ok} \quad \Delta \vdash \overline{\mathtt{T}}\ \text{ok} \quad \Delta \vdash \overline{\mathtt{W}}\ \text{ok} \\ \mathit{fields}(\mathtt{B}) \subseteq \overline{\mathtt{T}}\ \overline{\mathtt{f}} \quad \Delta;\emptyset \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}'} \quad \emptyset \vdash \overline{\mathtt{T}'} <: \overline{\mathtt{T}} \\ \forall\ \mathtt{new}_{\mathtt{D}}\mathtt{<}\overline{\mathtt{S}}\mathtt{>(}\overline{\mathtt{d}}\mathtt{)} \in \overline{\mathtt{e}}, \overline{\mathtt{M}}, \overline{\mathtt{N}}.\ \ \mathtt{D} = \mathtt{C} \end{array}}{\mathtt{class\ C<}\overline{\mathtt{X}} \lhd (\overline{\mathtt{I}}, \overline{\mathtt{J}})\mathtt{>} : \mathtt{B} \lhd \mathtt{A}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}}\ \overline{\mathtt{e}};\ \overline{\mathtt{M}};\ \overline{\mathtt{W}}\ \overline{\mathtt{g}};\ \overline{\mathtt{N}};\}\ \text{ok}}\ [\textsc{WF-ClassDef}]$$

Figure 4: Well-Formed Constructs

Subcase [S-Trans]: Let $\mathtt{T}$ be the intermediate type. Then $\mathit{fields}(\mathtt{V}) = \mathit{fields}(\mathtt{T})\ @\ \overline{\mathtt{H}'}$ and $\mathit{fields}(\mathtt{V}) = \mathit{fields}(\mathtt{T})\ @\ \overline{\mathtt{H}''}$ by the induction hypothesis. Thus $\overline{\mathtt{G}} = \overline{\mathtt{H}'}\ @\ \overline{\mathtt{H}''}$.

Case $\mathtt{V} = \mathtt{typeOf[C<}\overline{\mathtt{S}}\mathtt{>]}$**:** Then $CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \lhd (\overline{\mathtt{I}}, \overline{\mathtt{J}})\mathtt{>} : \mathtt{B} \lhd \mathtt{A}\{\overline{\mathtt{W}}\ \overline{\mathtt{h}}\ \overline{\mathtt{e}}...\}$. We continue by induction over the derivation of $\emptyset \vdash \mathtt{V} <: \mathtt{U}$. The only complex cases are [S-Kind] and [S-Trans].

Subcase [S-Trans]: As above.

Subcase [S-Kind]: $[\overline{\mathtt{X}} \mapsto \overline{\mathtt{S}}]\mathtt{A} = \mathtt{U}$ Then $\mathit{fields}(\mathtt{V}) = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{S}}]\overline{\mathtt{W}}\ \overline{\mathtt{h}}$. By well-formedness of $\mathtt{C}$, we know that $\mathit{fields}(\mathtt{B}) \subseteq \overline{\mathtt{W}}\ \overline{\mathtt{h}}$. Then $[\overline{\mathtt{X}} \mapsto \overline{\mathtt{S}}]\mathit{fields}(\mathtt{B}) \subseteq [\overline{\mathtt{X}} \mapsto \overline{\mathtt{S}}]\overline{\mathtt{W}}\ \overline{\mathtt{h}}$, and $\mathit{fields}([\overline{\mathtt{X}} \mapsto \overline{\mathtt{S}}]\mathtt{B}) \subseteq [\overline{\mathtt{X}} \mapsto \overline{\mathtt{S}}]\overline{\mathtt{W}}\ \overline{\mathtt{h}}$ by lemma Substitution Distributes over *fields*.

Lemma 2 (Subtyping Preserves Method Typing) *If* $\mathit{mtype}(\mathtt{m}, \mathtt{U}) = \overline{\mathtt{T}} \to \mathtt{S}$ *and* $\emptyset \vdash \mathtt{V} <: \mathtt{U}$ *then* $\mathit{mtype}(\mathtt{m}, \mathtt{V}) = \overline{\mathtt{T}} \to \mathtt{S}$.

Proof By induction over the derivation of $\emptyset \vdash \mathtt{V} <: \mathtt{U}$.

Case [S-Reflex]: Trivial.

Case [S-Bound]: Not applicable.

$$\mathtt{Object} \rightarrow \{\}_{\mathtt{Object}} \ \text{[R-OBJECT]} \qquad \frac{\mathit{fields}(\mathtt{C{<}\overline{S}{>}}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\mathtt{new}_{\mathtt{C}}\mathtt{<\overline{S}>(\overline{v})} \rightarrow \{\overline{\mathtt{f}} \mapsto \overline{\mathtt{v}}\}_{\mathtt{C<\overline{S}>}}} \ \text{[R-NEW]}$$

$$\frac{\emptyset \vdash \mathtt{G} <: \mathtt{T}}{\mathtt{(T)}\Phi_{\mathtt{G}} \rightarrow \Phi_{\mathtt{G}}} \ \text{[R-CAST]} \qquad \frac{\mathit{field\text{-}vals}(\mathtt{typeOf\,[C<\overline{S}>]}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}\ \overline{\mathtt{e}}}{\mathtt{C<\overline{S}>} \rightarrow \{\overline{\mathtt{f}} \mapsto \overline{\mathtt{e}}\}_{\mathtt{typeOf\,[C<\overline{S}>]}}} \ \text{[R-CLASS]}$$

$$\Phi_{\mathtt{G}}\mathtt{.f} \rightarrow \Phi_{\mathtt{G}}(\mathtt{f}) \ \text{[R-FIELD]} \qquad \frac{\mathit{mbody}(\mathtt{m}, \mathtt{G}) = (\overline{\mathtt{x}}, \overline{\mathtt{e}}_0)}{\Phi_{\mathtt{G}}\mathtt{.m(\overline{d})} \rightarrow [\overline{\mathtt{x}} \mapsto \overline{\mathtt{d}}, \mathtt{this} \mapsto \Phi_{\mathtt{G}}]\overline{\mathtt{e}}_0} \ \text{[R-INVK]}$$

$$\frac{\mathtt{e} \rightarrow \mathtt{e}'}{\mathtt{e.f} \rightarrow \mathtt{e'.f}} \ \text{[C-FIELD]} \qquad \frac{\mathtt{e} \rightarrow \mathtt{e}'}{\mathtt{e.m(\overline{d})} \rightarrow \mathtt{e'.m(\overline{d})}} \ \text{[C-RCVR]} \qquad \frac{\overline{\mathtt{e}} \rightarrow \overline{\mathtt{e}'}}{\mathtt{new}_{\mathtt{C}}\mathtt{<\overline{S}>(\overline{e})} \rightarrow \mathtt{new}_{\mathtt{C}}\mathtt{<\overline{S}>(\overline{e'})}} \ \text{[C-NEW]}$$

$$\frac{\overline{\mathtt{e}} \rightarrow \overline{\mathtt{e}'}}{\mathtt{d.m(\overline{e})} \rightarrow \mathtt{d.m(\overline{e'})}} \ \text{[C-ARG]} \qquad \frac{\mathtt{e} \rightarrow \mathtt{e}'}{\mathtt{(T)e} \rightarrow \mathtt{(T)e'}} \ \text{[C-CAST]} \qquad \frac{\overline{\mathtt{e}} \rightarrow \overline{\mathtt{e}'}}{\{\overline{\mathtt{f}} \mapsto \overline{\mathtt{e}}\}_{\mathtt{G}} \rightarrow \{\overline{\mathtt{f}} \mapsto \overline{\mathtt{e}'}\}_{\mathtt{G}}} \ \text{[C-MAPPING]}$$

Figure 5: Computation Rules

$$\mathit{fields}(\mathtt{Object}) = \emptyset \ \text{[F-OBJECT]} \quad \mathit{field\text{-}vals}(\mathtt{Object}) = \emptyset \ \text{[FV-OBJECT]} \quad \mathit{field\text{-}vals}(\mathtt{C<\overline{S}>}) = \emptyset \ \text{[FV-CLASS]}$$

$$\frac{\begin{array}{c} \mathit{CT}(\mathtt{C}) = \mathtt{class\ C<\overline{X} \triangleleft (\overline{I}, \overline{J})> : B \triangleleft A\ \{\overline{T}\ \overline{f}\ \overline{e};\ \overline{M};\ \overline{W}\ \overline{g};\ \overline{N};\}} \\ \mathit{fields}([\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}]\mathtt{A}) = \overline{\mathtt{U}}\ \overline{\mathtt{h}} \end{array}}{\mathit{fields}(\mathtt{C<\overline{R}>}) = \overline{\mathtt{U}}\ \overline{\mathtt{h}}\ \cup\ [\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}]\overline{\mathtt{W}}\ \overline{\mathtt{g}}} \ \text{[F-CLASS]}$$

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C<\overline{X} \triangleleft (\overline{I}, \overline{J})> : B \triangleleft A\ \{\overline{T}\ \overline{f}\ \overline{e};\ \overline{M};\ \overline{W}\ \overline{g};\ \overline{N};\}}}{\mathit{fields}(\mathtt{typeOf\,[C<\overline{R}>]}) = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}]\overline{\mathtt{T}}\ \overline{\mathtt{f}}} \ \text{[F-TYPEOF]}$$

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C<\overline{X} \triangleleft (\overline{I}, \overline{J})> : B \triangleleft A\ \{\overline{T}\ \overline{f}\ \overline{e};\ \overline{M};\ \overline{W}\ \overline{g};\ \overline{N};\}}}{\mathit{field\text{-}vals}(\mathtt{typeOf\,[C<\overline{R}>]}) = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}]\overline{\mathtt{T}}\ \overline{\mathtt{f}}\ \overline{\mathtt{e}}} \ \text{[FV-TYPEOF]}$$

$$\frac{\mathtt{U\ m\ (\overline{T}\ \overline{x})\ \{return\ e;\}} \in \mathit{methods}(\mathtt{G})}{\mathit{mtype}(\mathtt{m}, \mathtt{G}) = \overline{\mathtt{T}} \rightarrow \overline{\mathtt{U}}} \ \text{[MTYPE]} \qquad \frac{\mathtt{U\ m\ (\overline{T}\ \overline{x})\ \{return\ e;\}} \in \mathit{methods}(\mathtt{G})}{\mathit{mbody}(\mathtt{m}, \mathtt{G}) = (\overline{\mathtt{x}}, \overline{\mathtt{e}})} \ \text{[MBODY]}$$

$$\mathit{methods}(\mathtt{Object}) = \emptyset \ \text{[METHODSOBJECT]}$$

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C<\overline{X} \triangleleft (\overline{I}, \overline{J})> : B \triangleleft A\ \{\overline{T}\ \overline{f}\ \overline{e};\ \overline{M};\ \overline{W}\ \overline{g};\ \overline{N};\}}}{\mathit{methods}(\mathtt{C<\overline{R}>}) = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}]\overline{\mathtt{N}}\ \cup\ \mathit{methods}([\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}]\mathtt{A})} \ \text{[METHODSCLASS]}$$

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C<\overline{X} \triangleleft (\overline{I}, \overline{J})> : B \triangleleft A\ \{\overline{T}\ \overline{f}\ \overline{e};\ \overline{M};\ \overline{W}\ \overline{g};\ \overline{N};\}}}{\mathit{methods}(\mathtt{typeOf\,[C<\overline{R}>]}) = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}]\overline{\mathtt{M}}\ \cup\ \mathit{methods}([\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}]\mathtt{B})} \ \text{[METHODSTYPEOF]}$$

$$\frac{\mathit{mtype}(\mathtt{m}, \mathtt{G}) = \overline{\mathtt{U}} \rightarrow \overline{\mathtt{V}} \text{ implies } \overline{\mathtt{W}} = \overline{\mathtt{U}} \text{ and } \overline{\mathtt{T}} = \overline{\mathtt{V}}}{\mathit{override}(\mathtt{m}, \mathtt{G}, \overline{\mathtt{W}} \rightarrow \overline{\mathtt{T}})} \ \text{[OVERRIDE]}$$

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C<\overline{X} \triangleleft (\overline{I}, \overline{J})> : B \triangleleft A\ \{...\}}}{\mathit{params}(\mathtt{C<\overline{T}>}) = \overline{\mathtt{X}} \sqsubset (\overline{\mathtt{I}}, \overline{\mathtt{J}})} \ \text{[PARAMS]} \qquad \frac{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C<\overline{X} \triangleleft (\overline{I}, \overline{J})> : B \triangleleft A\ \{...\}}}{\mathit{params}(\mathtt{typeOf\,[C<\overline{T}>]}) = \overline{\mathtt{X}} \sqsubset (\overline{\mathtt{I}}, \overline{\mathtt{J}})} \ \text{[PARAMSTYPEOF]}$$

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C<\overline{X} \triangleleft (\overline{I}, \overline{J})> : B \triangleleft A\ \{...\}}}{\mathit{super}(\mathtt{C<\overline{T}>}) = \mathtt{A}} \ \text{[SUPER]} \qquad \frac{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C<\overline{X} \triangleleft (\overline{I}, \overline{J})> : B \triangleleft A\ \{...\}}}{\mathit{super}(\mathtt{typeOf\,[C<\overline{T}>]}) = \mathtt{B}} \ \text{[SUPERTYPEOF]}$$

Figure 6: Auxiliary Functions

Case [S-Super]: In this case, $\mathtt{V} = \mathtt{C<\overline{R}>}$, where $\mathit{CT}(\mathtt{C}) = \mathtt{class\ C<\overline{X} \triangleleft (\overline{I}, \overline{J})> : B \triangleleft A\{...\overline{M}\}}$ and $\mathtt{U} = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}]\mathtt{A}$. We need to show that $\mathit{mtype}(\mathtt{m}, \mathtt{C<\overline{R}>}) = \overline{\mathtt{T}} \rightarrow \mathtt{S}$. By lemma *methods* is well-defined, it suffices to show that $\mathtt{U\ m\ (\overline{T}\ \overline{x})\ \{return\ e;\}} \in \mathit{methods}(\mathtt{C<\overline{R}>})$.

We know that $\mathtt{U\ m\ (\overline{T}\ \overline{x})\ \{return\ e;\}} \in \mathit{methods}(\mathtt{U})$. So, we proceed by induction on the derivation of $\mathit{methods}(\mathtt{U})$.

Subcase $\mathtt{U} = \mathtt{Object}$: Impossible, since `Object` has no methods.

Subcase $\mathtt{U} = \mathtt{D<\overline{W}>}$: Then, $\mathtt{U} = [\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}]\mathtt{A}$. By [METHODSCLASS], $\mathit{methods}(\mathtt{V}) = \mathit{methods}(\mathtt{C<\overline{R}>}) = ([\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}]\overline{\mathtt{M}}) \cup \mathit{methods}([\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}]\mathtt{A}) = ([\overline{\mathtt{X}} \mapsto \overline{\mathtt{R}}]\overline{\mathtt{M}}) \cup \mathit{methods}(\mathtt{U})$. Therefore, any method in $\mathit{methods}(\mathtt{U})$ must be in $\mathit{methods}(\mathtt{V})$.

Case [S-Typeof]: Analagous to [S-SUPER].

Case [S-Trans]: Let the intermediate type be $\mathtt{T}$. Then $\emptyset \vdash \mathtt{V} <: \mathtt{T}$ and $\emptyset \vdash \mathtt{T} <: \mathtt{U}$. Thus, by the induction hypothesis, $\mathit{mtype}(\mathtt{m}, \mathtt{V}) = \mathit{mtype}(\mathtt{m}, \mathtt{T}) = \mathit{mtype}(\mathtt{m}, \mathtt{U})$.

Lemma 3 (Substitution Preserves Typing)

If $\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\Gamma = \overline{\mathtt{x}} : \overline{\mathtt{S}}$ *and* $\Delta;\emptyset \vdash \overline{\mathtt{d}} : \overline{\mathtt{U}}$ *and* $\Delta \vdash \overline{\mathtt{U}} <: \overline{\mathtt{S}}$ *then* $\Delta;\emptyset \vdash [\overline{\mathtt{x}} \mapsto \overline{\mathtt{d}}]\mathtt{e} : \mathtt{T}'$ *where* $\Delta \vdash \mathtt{T}' <: \mathtt{T}$.

Proof With the above two lemmas, this one follows by straightforward structural induction over the derivation of $\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}$.

Given the above lemmas, the following subject reduction proof is a simple structural induction with a case analysis on the typing rule used to derive $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{S}$.

Theorem 1 (Subject Reduction) *If* $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{S}$ *and* $\mathtt{e} \to \mathtt{e}'$ *then* $\emptyset;\emptyset \vdash \mathtt{e}' : \mathtt{T}$ *where* $\emptyset \vdash \mathtt{T} <: \mathtt{S}$.

Proof We prove this by structural induction on the derivation of $\mathtt{e} \to \mathtt{e}'$.

Case [R-Object]: Immediate.

Case [R-New]: Immediate from the premises of [T-New] and [R-New].

Case [R-Class]: Immediate from the premises of [WF-ClassDef] and [R-Class].

Case [R-Cast]: By [T-Cast], $\mathtt{e} = \mathtt{(T)}\Phi_{\mathtt{G}}$ must have type $\mathtt{T}$. By [T-Mapping], $\mathtt{e}' = \Phi_{\mathtt{G}}$ must have type $\mathtt{G}$. By hypothesis of the reduction rule, $\emptyset \vdash \mathtt{G} <: \mathtt{T}$.

Case [R-Field]: We know that $\mathtt{e} = \{\overline{\mathtt{f}} \mapsto \overline{\mathtt{d}}\}_{\mathtt{G}}\mathtt{.f}_i$ and that $\mathtt{e}' = \mathtt{d}_i$. Since $\mathtt{e}$ must have been typed by [T-Field], we know that *fields*$(\mathtt{G}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ and $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{T}_i$. Further, since $\{\overline{\mathtt{f}} \mapsto \overline{\mathtt{d}}\}_{\mathtt{G}}$ must have been typed by [T-Mapping], we know that $\emptyset;\emptyset \vdash \mathtt{d}_i : \mathtt{S}$ where $\emptyset \vdash \mathtt{S} <: \mathtt{T}$.

Case [R-Invk]: We know that $\mathtt{e} = \Phi_{\mathtt{G}}\mathtt{.m}(\overline{\mathtt{d}})$ and that $\mathtt{e}' = [\overline{\mathtt{x}} \mapsto \overline{\mathtt{d}}, \mathtt{this} \mapsto \Phi_{\mathtt{G}}]\mathtt{e}_0$. Further, $\mathtt{e}$ was typed by [T-Invk] to have type $\mathtt{U}$ where *mtype*$(\mathtt{m},\mathtt{G}) = \overline{\mathtt{T}}\to\mathtt{U}$ and $\emptyset;\emptyset \vdash \overline{\mathtt{d}} : \overline{\mathtt{T}'}$ and $\emptyset \vdash \overline{\mathtt{T}'} <: \overline{\mathtt{T}}$. As a premise of [R-Invk], we know that *mbody*$(\mathtt{m},\mathtt{G}) = (\overline{\mathtt{x}}, \mathtt{e}_0')$.

By the lemma *mtype* and *mbody* agree, $\emptyset; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{G} \vdash \mathtt{e}_0' : \mathtt{U}'$ where $\emptyset \vdash \mathtt{U}' <: \mathtt{U}$. Then by the lemma Substitution Preserves Typing, $\emptyset;\emptyset \vdash \mathtt{e}' = [\overline{\mathtt{x}} \mapsto \overline{\mathtt{d}}, \mathtt{this} \mapsto \Phi_{\mathtt{G}}]\mathtt{e}_0 : \mathtt{U}''$ where $\emptyset \vdash \mathtt{U}'' <: \mathtt{U}'$.

Case [C-Cast]: Trivial, since $\emptyset \vdash \mathtt{(T)e} : \mathtt{T}$ for any $\mathtt{e}$.

Case [C-Map]: Immediate from the induction hypothesis and the transitivity of subtyping.

Case [C-New]: Immediate from the induction hypothesis and the transitivity of subtyping.

Case [C-Arg]: Immediate from the induction hypothesis and the transitivity of subtyping.

Case [C-Rcvr]: We know that $\mathtt{e} = \mathtt{e}_0\mathtt{.m}(\overline{\mathtt{d}})$ and $\mathtt{e}' = \mathtt{e}_0'\mathtt{.m}(\overline{\mathtt{d}})$. Further, $\mathtt{e}$ must have been typed by [T-Invk], which means that $\emptyset;\emptyset \vdash \mathtt{e}_0 : \mathtt{W}$ for some ground type $\mathtt{W}$, and that $\emptyset;\emptyset \vdash \overline{\mathtt{d}} : \overline{\mathtt{V}}$ and $\emptyset \vdash \overline{\mathtt{V}} <: \overline{\mathtt{U}}$, and also *mtype*$(\mathtt{m},\mathtt{W}) = \overline{\mathtt{U}}\to\mathtt{T}$. By the induction hypothesis, $\emptyset;\emptyset \vdash \mathtt{e}_0' : \mathtt{W}'$ where $\emptyset \vdash \mathtt{W}' <: \mathtt{W}$. Therefore, by lemma Subtyping Preserves Method Typing, *mtype*$(\mathtt{m},\mathtt{W}') = \overline{\mathtt{U}}\to\mathtt{T}$ and thus $\emptyset;\emptyset \vdash \mathtt{e}_0'\mathtt{.m}(\overline{\mathtt{d}}) : \mathtt{T}$ by [T-Invk].

Case [C-Field]: If $\mathtt{e.f}_i \to \mathtt{e}'\mathtt{.f}_i$, then $\mathtt{e} \to \mathtt{e}'$. Further, $\mathtt{e.f}_i$ must have been typed by [T-Field] to have type $\mathtt{T}_i$. Therefore, by the induction hypothesis, $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{S}$ and $\emptyset;\emptyset \vdash \mathtt{e}' : \mathtt{S}'$ where $\emptyset \vdash \mathtt{S} <: \mathtt{S}'$. Then, by the lemma Fields are Preserved by Subtypes, *fields*$(\mathtt{S}') =$ *fields*$(\mathtt{S})$ @ $\overline{\mathtt{F}}$ for some $\overline{\mathtt{F}}$, and by [T-Field], $\emptyset;\emptyset \vdash \mathtt{e}'\mathtt{.f}_i : \mathtt{T}_i$.

The proof of progress presents no additional complications, and requires only one new lemma.

Lemma 4 (Agreement of ***fields*** **and** ***field-vals*****)** *If* *fields*$(\mathtt{G}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ *and* *field-vals*$(\mathtt{G}) = \overline{\mathtt{T}'}\ \overline{\mathtt{f}'}\ \overline{\mathtt{e}}$ *then* $\overline{\mathtt{T}} = \overline{\mathtt{T}'}$, $\overline{\mathtt{f}} = \overline{\mathtt{f}'}$ *and* $\emptyset;\emptyset \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}}$ *where* $\emptyset \vdash \overline{\mathtt{S}} <: \overline{\mathtt{T}}$.

Theorem 2 (Progress) *If* $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{S}$ *then one of the following holds:*

- $\mathtt{e} = \{\overline{\mathtt{f}} \mapsto \overline{\mathtt{v}}\}_{\mathtt{G}}$
- $\mathtt{e} \to \mathtt{e}'$
- $\mathtt{e} = \mathtt{(S)e}'$ *and* $\emptyset;\emptyset \vdash \mathtt{e}' : \mathtt{T}$ *and* $\emptyset \vdash \mathtt{T} \not<: \mathtt{S}$.

Proof By induction over the derivation of $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{S}$.

Case [T-Var]: This is a contradiction, since $\mathtt{e}$ is ground.

Case [T-Class]: In this case $\mathtt{e} = \mathtt{C<}\overline{\mathtt{S}'}\mathtt{>}$ where $\mathtt{C<}\overline{\mathtt{S}'}\mathtt{>}$ is ground. Then by the lemma Agreement of *field-vals* and *fields*, *params*$(\mathtt{C<}\overline{\mathtt{S}'}\mathtt{>}) = \overline{\mathtt{X}} \sqsubset (\overline{\mathtt{I}}, \overline{\mathtt{J}})$ for some Further, *field-vals*$(\mathtt{typeOf[C<}\overline{\mathtt{S}'}\mathtt{>]}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}\ \overline{\mathtt{e}}$ for some $\overline{\mathtt{T}}$, $\overline{\mathtt{f}}$, and $\overline{\mathtt{e}}$. Therefore, [R-Class] applies and $\mathtt{S} \to \{\overline{\mathtt{f}} \mapsto [\overline{\mathtt{X}} \mapsto \overline{\mathtt{S}'}]\mathtt{C<}\overline{\mathtt{S}'}\mathtt{>}\}_{\mathtt{typeOf[C<}\overline{\mathtt{S}'}\mathtt{>]}}$.

Case [T-Mapping]: Either $\mathtt{e}$ is already a value, or $\mathtt{e} = \{\overline{\mathtt{f}} \mapsto \overline{\mathtt{e}}\}_{\mathtt{G}}$ where not all of the $\overline{\mathtt{e}}$ are values. Then by the induction hypothesis, there is some i such that either $\mathtt{e}_i \to \mathtt{e}'_i$, in which case [C-Mapping] applies, or $\mathtt{e}_i$ contains a bad cast, and the case is complete.

Case [T-Cast]: Here there are three cases:

- $\mathtt{e} = \mathtt{(S)e}'$ where $\mathtt{e}'$ is not a mapping. Then [C-Cast] applies.

- $\mathtt{e} = \mathtt{(S)}\Phi_{\mathtt{T}}$ where $\emptyset \vdash \mathtt{T} <: \mathtt{S}$. Then [R-Cast] applies.
- $\mathtt{e} = \mathtt{(S)}\Phi_{\mathtt{T}}$ where $\vdash \mathtt{T} \not<: \mathtt{S}$. Then $\mathtt{e}$ is a bad cast.

Case [T-New]: From the antecedent of [T-New] the premise of [R-New] applies.

Case [T-Invk]: Here there are two cases.

- $\mathtt{e} = \mathtt{r.m}(\overline{\mathtt{d}})$ where $\mathtt{r}$ is not a mapping. Then by the induction hypothesis, either $\mathtt{r}$ contains a bad cast or $\mathtt{r} \to \mathtt{r}'$ and [C-Rcvr] applies.
- $\mathtt{e} = \Phi_{\mathtt{T}}\mathtt{.m}(\overline{\mathtt{d}})$ We know from the antecedent that $\mathit{mtype}(\mathtt{m}, \mathit{bound}\mathtt{T}) = \overline{\mathtt{U}}{\to}\mathtt{V}$ and therefore $\mathit{mtype}(\mathtt{m}, \mathtt{T}) = \overline{\mathtt{U}}{\to}\mathtt{V}$ since $\mathtt{T}$ is ground. Therefore, since *mbody* is defined everywhere *mtype* is defined, $\mathit{mbody}(\mathtt{m}, \mathtt{T}) = (\overline{\mathtt{x}}, \mathtt{e}_0)$ for some $\overline{\mathtt{x}}$ and $\mathtt{e}_0$. Thus [R-Invk] applies.

Case [T-Field]: Here there are two cases, either the reciever is a mapping or not. In the first, by the antecedent of the typing rule, we can lookup the field successfully and apply [R-Field]. Otherwise, we can apply [C-Field].

From these, we can conclude the desired type safety result.

Theorem 3 (Soundness) *If* $\mathtt{e} : \mathtt{S}$ *then either*

- $\mathtt{e} \to^* \{\overline{\mathtt{f}} \mapsto \overline{\mathtt{v}}\}_{\mathtt{G}}$
- $\mathtt{e} \to^* \mathtt{e}'$ *where* $\mathtt{e}'$ *is an invalid cast*
- $\mathtt{e}$ *reduces infinitely.*

Proof Immediate from Subject Reduction and Progress.

5 Related Work

A number of object-oriented languages have included some form of metaclass system. Most notable among these is Smalltalk [11], but others include Common Lisp with CLOS [14].

All of these systems share a common architecture of the metaclass system in which each class has its own freely generated metaclass, defined by the class methods and fields of the class. In contrast, MCJ provides a hierarchy for structuring metaclass relationships, which provides significantly more modeling and abstraction flexibility.

The metaclass system present in Python [17] is quite similar to that provided here, with inheritance of instance methods as class methods from arbitrary other classes as metaclasses. However, Python is dynamically-typed, and many of the uses to which Python metaclasses are put are not possible in a statically typed language. The work on static type systems for Python has not included metaclasses [15].

Several type systems have also been proposed for languages with metaclasses, including the Strongtalk language [3], which turns Smalltalk into a structurally-typed language with extensive static checking. However, because the Smalltalk metaclass system is so different from the one in MCJ, many of the interesting aspects of the type system do not carry over. Furthermore, the Strongtalk papers do not provide a formal semantics and analysis of the system. A formal analysis of inheritance in Smalltalk is provided in [7] but this again does not consider the hierarchy of metaclasses presented here.

Graver and Johnson [12] present another type system for Smalltalk, with a formalism and sketch of a safety proof. Again, the metaclass type system is substantially different, reflecting the underlying Smalltalk system. Additionally, the paper is concerned primarily with optimization as opposed to static checking.

Cointe [6] presents a model of metaclasses extremely similar to that presented here. In it, he provides several overlapping motivations to our own. One is to regularize the metaclass system of Smalltalk, and another is to enable additional programming flexibility. To this we add modeling freedom, and a relation to static methods in more recent OO languages. Cointe's work, however, does not provide a formal model, so it is difficult to determine the exact relationship between the systems. Additionally, his work is in a untyped setting (Lisp and Smalltalk) and thus the safety theorems proved here are not possible.

When viewed as "instance generators", our metaclasses are similar to prototypes in untyped languages such as Self. Prototypes generate new instances, which themselves can generate new instances. Our language is more restrictive than prototype-based languages in the sense that all metaclasses and instance classes must be declared statically (i.e., by writing down class definitions). But our language is more expressive in the sense that we include classes and subclassing relationships. Also, unlike typical prototype-based languages, our language is statically typed.

Formalized calculi for object-oriented languages are abundant in the programming languages community today, including Featherweight Java [13], upon which MCJ is based. However, none of them have considered static methods, which are the closest analogue of the metaclass functionality in MCJ.

Finally, the motivation for this work comes from the language MetaGen, introduced in [2]. MetaGen can be seen as an extension of MCJ, which provides numerous other advanced type features. Here we restrict ourselves to metaclasses and analyze the properties of the system formally.

6 Conclusions and Future Work

With MCJ, we have devised a core calculus for metaclasses that is more flexible than that available in more traditional metaclass systems such as Smalltalk and that allows clean expression of many common design patterns. In doing so, we have demonstrated that metaclasses can be added to a nominally-typed, statically-checked language without either significant complication of the semantics or difficulty in the proof of soundness.

In addition to this contribution, we have elucidated several other important points about the integration of metaclasses into an object-oriented system. The `typeOf` type operator is to our knowledge unique, and plays a key role in the soundness of the system. The discovery that the `Class` class played no special role, and that thus the class and metaclass hierarchies could both be rooted at `Object` is also novel. Finally, we have shown how an expressive framework of metaclasses has positive effects in other areas of the language, such as mechanisms for object construction.

A natural extension of the work in this paper is to expand MCJ to include more of the features presented in [2], so as to allow inclusion of the system for checking dimensions of physical quantities. Such an extension would allow for a proof of "dimensional soundness" in the resulting system. Further, any realistic system will have imperative features, and while such features do not seem likely to interact badly with metaclasses, no sound system is built on such assumptions.

Another interesting extension would be to expand our calculus to include either multiple inheritance (as does Python) or some alternative such as mixins or traits, as there may be interesting interactions between metaclasses and these features that have yet to be discovered.

Acknowledgments

We would like to thank Jan-Willem Maessen for his extremely helpful feedback on the formal rules presented in this paper. We also thank David Chase for not believing in us unless we were right, Victor Luchangco for many valuable discussions, and Guy Steele for his many helpful comments.

References

[1] Eric Allen, Jonathan Bannet, and Robert Cartwright. A first-class approach to genericity. In *Proceedings of the 2003 ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 96–114. ACM Press, 2003.

[2] Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, and Guy Steele. Object-Oriented Units of Measurement. In *Proceedings of the 2004 ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, 2004. Available at http://research.sun.com/projects, under Programming Languages.

[3] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *Proceedings of the OOPSLA '93 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 215–230, September 1993.

[4] Luca Cardelli. A semantics of multiple inheritance. In *Proc. of the international symposium on Semantics of data types*, pages 51–67. Springer-Verlag New York, Inc., 1984.

[5] Robert Cartwright and Guy L. Steele, Jr. Compatible genericity with run-time types for the java programming language. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 201–215. ACM Press, 1998.

[6] Pierre Cointe. Metaclasses are first class: the ObjVLisp model. In *Proceedings of the OOPSLA '87 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 156–162. ACM Press, 1987.

[7] William R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, May 15 1989.

[8] R. Kent Dybvig. Writing Hygienic Macros in Scheme with Syntax-Case. Technical report, Indiana University Computer Science Dept., July 03 1992.

[9] Matthias Felleisen. On the expressive power of programming languages. In Neil D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 134–151, Copenhagen, Denmark, 15–18 May 1990. Springer.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, 1994.

[11] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, Massachusetts, 1989.

[12] Justin O. Graver and Ralph E. Johnson. A type system for Smalltalk. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 136–150, San Francisco, California, January 1990.

[13] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 132–146. ACM Press, 1999.

[14] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[15] Michael Salib. Static Type Inference with Starkiller. In *PyCon DC*, 2004. http://www.python.org/pycon/dc2004/papers/1/paper.pdf.

[16] Guy L. Steele Jr. *Common Lisp: The Language, Second Edition*. Digital Press, Bedford, Massachusetts, 1990.

[17] Guido van Rossum. Unifying types and classes in Python 2.2. http://www.python.org/2.2.2/descrintro.html, 2002.

[18] Christopher A. Welty and David A. Ferrucci. What's in an instance? Technical report, Rochester Polytechnic Institute Computer Science Dept., 1994.