Object-Oriented Units of Measurement

Eric Allen David Chase Victor Luchangco Jan-Willem Maessen Guy L. Steele Jr.

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

ABSTRACT

Programs that manipulate physical quantities typically represent these quantities as raw numbers corresponding to the quantities' measurements in particular units (e.g., a length represented as a number of meters). This approach eliminates the possibility of catching errors resulting from adding or comparing quantities expressed in different units (as in the Mars Climate Orbiter error [11]), and does not support the safe comparison and addition of quantities of the same dimension. We show how to formulate dimensions and units as classes in a nominally typed object-oriented language through the use of statically typed metaclasses. Our formulation allows both parametric and inheritance polymorphism with respect to both dimension and unit types. It also allows for integration of encapsulated measurement systems, dynamic conversion factors, declarations of scales (including nonlinear scales) with defined zeros, and nonconstant exponents on dimension types. We also show how to encapsulate most of the "magic machinery" that handles the algebraic nature of dimensions and units in a single metaclass that allows us to treat select static types as generators of a free abelian group.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design,Languages

1. INTRODUCTION

Physical units and dimensions are a commonly used computational construct in science and engineering, but there is relatively little support for them in programming languages. Instead, physical quantities are typically represented as numbers that correspond to the quantities' measurement in a particular unit (e.g., a length represented as a number of meters); the unit and dimension of the measurement are not represented in the program at all (except possibly in the names of variables used to hold these numbers). Thus, the units and dimensions cannot be used in the analysis of the program. In particular, they cannot be used to detect errors resulting from mismatched quantities. While trivial, such errors are not uncommon, and can have costly repercussions. For example, the loss of the Mars Climate Orbiter in September 1999 was ultimately traced to a failure in the software to convert between English and metric units [11]. In another example, a space shuttle rolled itself away from the Earth in response to an instruction to bounce a laser off a mountain it was told was 10,000 miles high, rather than the intended 10,000 feet [24]. Units were not programmed as part of the input data and therefore all inputs were implicitly interpreted as measurements in miles.

At least superficially, dimensions and units bear a strong similarity to types. However, most contemporary type systems, even with generic types, are insufficiently expressive to capture several properties necessary for dimension checking. For example, dimension checking at compile-time requires the compiler to manipulate dimensions according to a dimensional algebra (e.g., `Time` × `Length` = `Length` × `Time`). Most previous formulations of dimension checking in programming languages have encoded dimensions as an ad hoc language feature rather than integrating them into a general type system. The few attempts at integrating dimensions into more general type systems have focused on structural type systems over functional and procedural languages with significantly different design considerations than those of mainstream object-oriented languages [20].

In this paper, we show how to integrate dimensions and units with classes in a nominally typed object-oriented language (such as the Java™ Programming Language or C#) in a way that supports static checking of units and dimensions, and with a performance cost no greater than that of boxing primitive values.[1]

[1]Boxing is a technique that enables a primitive value (e.g., `0`) to be used in contexts requiring a reference value by wrapping the primitive in a new reference (`new Integer(0)`). It can negatively impact execution time because boxed values must be dereferenced when accessed. However, clever compiler optimizations, particularly in just-in-time compilers,

OOPSLA'04, October 24-28, 2004 Vancouver, British Columbia, Canada,

ACM 1-58113-712-5/03/0010 .

Our formulation allows dimensions and units to be treated as ordinary class-based types, and allows both parametric and inheritance polymorphism with respect to both dimension and unit types. We describe the new language features required to support this integration in the context of the MixGen extension [1] of the Java Programming Language, where generic types are first-class. MixGen allows type-dependent operations such as casts on arbitrary generic types, and even expression of *mixins* (i.e., generic classes parameterized by their own superclasses).

First-class generic types play a crucial role in our formulation of dimensions. We present our language as an extension of MixGen to ground it in a sound, statically typed object-oriented language that supports first-class generic types. A formulation in the context of C++ templates, in particular, would not meet this criterion. MixGen includes many subtle features to handle pathologies that do not arise in our treatment of dimensions and units—for example, support for polymorphic recursion, hygienic method overriding, and nontrivial `with` clauses; we avoid discussing these features. Therefore, we expect our formulation to be accessible to readers familiar only with C++, C#, or the Java Programming Language, and with the notion of encoding mixins via generic types, as can be done with C++ templates. Also, because our real interest is in understanding the relationships of dimension and unit checking with object-oriented type systems in general, we make no attempt to maintain backward compatibility when extending MixGen.

Our main extension to MixGen is the addition of statically typed *metaclasses*, which we use to model values that have associated dimensional units. We also define a special metaclass `abelian class` that provides the algebraic properties necessary to model dimensions and units accurately. Although we strive to introduce as few features as possible, the language we end up with is quite different from the language we started with; we dub our new language *MetaGen*.

In Section 2, we briefly analyze some concepts important to dimension and unit checking, discuss design constraints for a system to do this checking, and explain why existing object-oriented languages do not satisfy these constraints. In Section 3, we describe several common uses of physical quantities for which we want to have static checking, and the features of MetaGen that support this ability, and we describe further extensions in Section 4. We discuss various syntactic and performance issues in Section 5, survey related work in Section 6, and conclude with future directions in Section 7.

2. ANALYSIS AND DESIGN CHALLENGES

In this section, we lay out the basic framework and terminology that we use throughout this paper. Although the view we take is not the only way to understand dimensions and units, we have found it useful for elucidating many design issues. We also discuss several pragmatic requirements that influence our design.

can eliminate much of this overhead by specializing code with variants for unboxed values in contexts where observable behavior is unaffected.

2.1 Basic terminology and analysis

The fundamental notion in our formulation is that of a *quantity*. Every quantity exists in a particular *dimension*[2] (e.g., `Length`, `Time`). When two quantities Q and Q' exist in the same dimension, we can answer the question, "how much of Q does it take to equal Q'?"

Some quantities are designated *units* and are denoted with special symbols (e.g., `Meter`, `Second`). Every quantity can be denoted by a *measurement* in terms of any unit U in its dimension. If it takes x of U to equal Q then we denote Q with the measurement xU, and we call x the *magnitude* of this measurement. For example, the denotation `5 Meters` is a measurement for a quantity in the dimension `Length` whose magnitude in terms of the unit `Meter` is `5`.

One advantage of using units is that it allows us to compare two measurements of the same dimension algebraically. To express a quantity Q denoted by a measurement $M = xU$ in unit U as a measurement in another unit V, we need not remeasure the same quantity in V (which is seldom practical, and often impossible). Instead, we can measure our unit U with unit V, resulting in a measurement yV; we say y is the *conversion factor* from U to V. Then we can form the measurement M' of Q in unit V by substituting yV for U in M: $M = xU = xyV = (xy)V$. Two measurements in the same unit can be compared by comparing their magnitudes arithmetically.

Magnitudes can be taken from an arbitrary algebraic field. For now, we assume that magnitudes are always real numbers (i.e., instances of a class `Real`, which we intentionally leave undefined). Furthermore, we use the standard literals and mathematical operators to denote real numbers and their methods, and we do not worry about errors due to computational limitations such as imprecision and overflow. We discuss taking magnitudes from arbitrary fields in Section 4.2, and implementation issues in Section 5.2.

We can also combine quantities of distinct dimension via multiplication and division to form new quantities. Such a quantity exists in a new dimension denoted by combining the respective dimensions with combinators $\times$ and /, which satisfy certain algebraic properties (e.g., `Length` $\times$ `Time` = `Time` $\times$ `Length`). The set of dimensions is an abelian group under $\times$ [21]. We define a special dimension, called *unity*, to serve as the multiplicative identity for dimensions, and we denote the inverse of a dimension D by D^{-1}.

Units of combined measurements are combinations of units of the constituent measurements. Just as we can have several units of a given primary dimension, we can have several derived units for a given derived dimension. It is also possible to define new primitive units of combined dimensions.

2.2 Design Constraints

We want a design that accurately models our understanding of dimensions and units. In addition, we impose the following pragmatic design constraints:

[2]We adopt for now the naïve view that every quantity exists in a single, fixed, dimension. In Section 3.6, we discuss some ways in which this naïve view needs to be refined.

1. It should be easy to convert a measurement in one unit to a measurement in another unit of the same dimension.

2. Type-dependent operations on dimensions and units, such as casting and `instanceof` tests, as well as the ability to print and store measurements, should be supported.

3. It should be possible to *require* that a quantity be denoted by a measurement in a specific unit (e.g., a length that must be measured in meters) as an additional error check, to avoid unnecessary conversions, and to manage the inaccuracies inherent in floating point calculations.

4. Programmers should be able to define new units and dimensions and to define behavior specific to that dimension (e.g., a method `dilate` for `Time`). Even if all programmers and users were willing to use the SI system, there are contexts in which units and dimension checking can be employed (currency, information, etc.) that are not part of this system.

5. We should allow for as much polymorphism as possible. In particular, we should support type parameters for dimensions and units.

6. Unit annotations on quantities should cost no more in execution time and space than boxing of primitive values.

7. We should add as few features to the language as possible without sacrificing conceptual coherence (i.e., "as simple as possible, but no simpler").

2.3 Why we need a new language

Ideally, we could model systems for static checking of dimensions and units as libraries in existing object-oriented languages. Unfortunately, even with first-class generic types, we quickly run into obstacles when trying to do so. The fundamental difficulty, which we encounter in many guises, is the need to treat a single entity as both a type and a value. For example, dimensions must be types so that we can statically check expressions for dimensional correctness. However, there are several ways in which dimensions are more like values than types. For example, we must be able to multiply dimensions together to form new dimensions, and this operation must obey algebraic properties such as commutativity. Even syntactically, the form of dimensions is value-like: the dimension `Length`/`Time`^{2} syntactically contains the value `2`.

There is another more subtle way in which units and dimensions behave like both types and values: a natural object-oriented modeling of dimensions identifies each dimension as a class—containing as instances the quantities of that dimension—as well as an instance of a class `Dimension`. For example, `Length` is a `Dimension` and `5 Meters` is a `Length`. We want to define a class `Dimension` so that we can specify polymorphic contexts that can be instantiated with any dimension, and so we can define functionality common to all dimensions in a single class. For example, we want to declare that for every dimension `D`, every quantity of `D` has a method `inUnit` that allows conversion to a measurement in any unit of `D`. We may also want to declare abstract methods such as `shortName` on dimensions, or a method `devices` that returns a collection of the devices used to measure quantities of a dimension. We cannot represent individual dimensions as subclasses of `Dimension` because the instances of particular dimensions—naturally identified as quantities in that dimension—should not be used in contexts requiring a dimension. If we model quantities as instances of their corresponding dimension class, and we model dimension classes as subclasses of `Dimension`, then by transitivity of subtyping, every quantity would also be an instance of `Dimension`, which is clearly wrong: `5 Meters` is not an instance of `Dimension`.

We might try to break this inheritance relationship with a shallow workaround, such as treating dimensions as subclasses of `Dimension` and introducing a parametric class `Quantity` to model quantities of a particular dimension:

```
abstract class Dimension {...}
class Length extends Dimension {...}
class Time extends Dimension {...}
...
class Quantity<D extends Dimension> {...}
```

Then lengths would be instances of `Quantity<Length>` and times would be instances of `Quantity<Time>`. This approach introduces a spurious distinction between `Quantity<Length>` and `Length`; these two classes are naturally identified. Furthermore, it prevents us from defining behavior specific to all quantities of a particular dimension. For example, there is no place to define methods such as `dilate` that are common to all instances of `Quantity<Time>`: If they were defined in class `Time` then they could not be accessed as methods of `D` from within `Quantity<D>`. If they were defined in `Quantity<D>` then they could be accessed by all instantiations of `Quantity`. If we instead define class `Quantity<D>` as a mixin that extends class `D`, then quantities of a particular dimension would again be instances of `Dimension` and we would be back to square one.

Continuing with yet another workaround, we could define an ad hoc subclass for each (actual) instantiation of `Quantity<D>` (e.g., `QuantityOfLength extends Quantity<Length>`). Because of the limitations of contemporary generic type systems, this sort of workaround is quite common. But it is unsatisfactory because it introduces critical invariants that cannot be statically checked (e.g., there must be *exactly one* ad hoc subclass of `Quantity<D>` for each subclass of `Dimension`; the name of each ad hoc subclass has a precise relationship with the instantiation of `Quantity<D>` it extends; etc.). Moreover, these unchecked invariants were introduced solely to work around the constraints of the static type system!

What we really want is a language that allows us to express the correct relationships among the concepts we are modeling. We want to define classes that can be used as values as well as static types. The inability to define such constructs is a pervasive but seldom recognized problem in contemporary object-oriented languages. Sometimes the problem is circumvented by defining unnatural class hierarchies with unenforced invariants, such as our hierarchy involving class `Quantity<D>`. Other times, the Singleton Pattern [12] is employed. This pattern involves constructing unique instances of classes and storing them in static final fields with the

name ONLY. For example, we can define class Length as a Singleton,

```
class Length extends Dimension {
  static final Length ONLY = new Length();
  private Length() {}
  ...
}
```

creating a one-one correspondence between class Length and its sole instance. We can treat the instance as a value and the class as a type while conceptually identifying the two.

The Singleton Pattern introduces extra machinery and verbosity into a program, and does not really address the underlying problem. We cannot use the (conceptually unique) instance of a type in a context requiring a value without first dereferencing the singleton instance ONLY, and we cannot access the ONLY field of a Singleton type in a parametric context. Furthermore, this pattern is useless in the current context: defining dimension classes as Singleton subclasses of Dimension does not help us to statically check algebraic combinations of dimensions and it does not address the problem that instances of a particular dimension should not be instances of class Dimension.

3. OUR SOLUTION

To properly capture the dual nature of dimensions and units as both values and types, we generalize the conventional notion of class inheritance. In addition to defining subtype relationships between classes, we also allow classes to be *instances* of other classes. These *instance classes* may be used like values that are defined statically. In this way, they obviate the need for the Singleton Pattern. But we also allow instance classes themselves to contain instance classes, and we allow class extension of instance classes.

These new instanceof relationships are expressed by replacing the keyword class with programmer-defined classes as necessary. For example, we can define the classes

```
abstract class Dimension {...}
Dimension Length {...}
```

where class Length is an instance of class Dimension.

The generalization of class relationships turns out to be the key language feature needed for integrating dimensions and units with object-oriented types. Furthermore, this generalization has applicability beyond just dimension checking. It allows us to more accurately model many mathematical structures based on set membership and subsets. Metaphorically, we can think of classes as sets. Class extension can be thought of as the subset relationship, and instanceof can be thought of as set membership. In this view, allowing classes to contain other classes is quite natural.

We now describe the core of the new semantics of *MetaGen*, an extension of the MixGen extension of the Java Programming Language that allows us to express the type relationships we want.

3.1 Metaclasses

First, we establish some terminology in order to describe our new semantics. A program consists of a collection of class definitions. The class definition of a class C specifies a *metaclass* D, of which C is an instance. C is also an instance of all superclasses of D. A metaclass is either a user-defined class or a special metaclass such as class, abstract class, or interface. We call D the *kind* of C. We also say that C is an *instance class* of D and all of D's superclasses, and that D (and all its superclasses) are *containing classes* of C. The kind of a class C is specified in its header:

kind C<*type params*> [extends *superclass*] {...}

For example, we may define the core classes in our dimension-checking library as follows:

```
abstract class Dimension {}
Dimension Length {...}
Dimension Time {...}
...
```

Classes Length and Time are instance classes of Dimension, and Dimension is the kind of classes Length and Time. The kind of Dimension is abstract class.

In general, there is no relationship between a class's kind and its superclasses. Instance classes should be viewed as special values that also denote a static type. Note that instance classes (like all classes) are defined statically; there is no way to allocate a new instance class dynamically. Instance classes may also have subclasses and instances of their own. An interface may serve as the kind of an instance class.

It is often useful to specify that type parameters are of a specific kind (e.g., that a parameter is of kind Dimension). Thus, we extend MixGen syntax so that we can specify a type parameter's kind as well as its bounding superclass:

kind parameter extends *bound*

For example, we define generic classes for both units and measurements. Units are tied to particular dimensions; we parameterize the class representing them by a type of kind Dimension. Measurements are tied to particular units; we parameterize the class representing them by a unit in the appropriate dimension.

```
class* Unit<Dimension D> extends D {...}

class Measurement<Dimension D, Unit<D> U> extends D {
   private final Real _magnitude;
   ...
}
```

The special kind class* is introduced to designate a class whose instances are all instance classes. Unit<D> is defined to be of kind class* so we can enforce the property that every unit will be associated with its own type. An instantiation of D must be an instance of Dimension (and a subtype of Object). For example, we can define the following instance classes:

```
Unit<Time> Minute {...}
Unit<Length> Foot {...}
Unit<Length> Mile {...}
...
```

Both `Unit<D>` and `Measurement<D,U>` extend their dimension `D`, allowing us to refer to the type `D` in places we expect measurements of `D`, (e.g., we can say that a method `m` takes a `Length` and call `m` with `new Measurement<Length,Mile>(5)`, denoting `5 Miles`). We also get a single point of control for specifying functionality specific to a particular dimension `D`: the class `D` itself. We stipulate that a naked type parameter occurring in an `extends` clause or as a kind of a type parameter must be instantiated only with classes that contain a zeroary constructor and no abstract methods (ensuring there is a superconstructor to call and that a mixin will not accidentally inherit abstract methods without defining them).[3] Because `Unit` and `Measurement` both have a superclass type parameter of kind `Dimension`, they implicitly require that all instance classes of `Dimension` include a zeroary constructor.

Because the unit of a measurement is a type parameter, we need not add an extra field to hold the unit; that is, the implementation need not include an extra unit reference in each instance. As in MixGen, instantiations of type parameters of a generic type can be stored in a class table entry shared by all instances of that instantiation. Therefore, unit support is no more expensive than boxing provided that magnitudes are represented with primitive values.

3.1.1 Static members and instance members

As in MixGen, a class definition includes both static members and instance members. Each member is either a field or a method (we leave open the question of how to integrate inner classes into our metaclass semantics).

If `C` is an instance class of `D` then the instance members of `D` are *static* members of `C` (i.e., members of the class object `C`). `C` may override the instance methods of `D` by declaring static methods with matching signatures. If `D` includes abstract methods (via either inheritance or direct definition), those methods must be defined as static methods in `C`. For example, we add the following definitions to our dimension classes:

```
abstract class Dimension {
  ...
  abstract String shortName();
}

Dimension Length {
  ...
  static String shortName() { return "L"; }
}

Dimension Time {
  ...
  static String shortName() { return "T"; }
}
```

The expression `Time.shortName()` evaluates to `"T"`. Also, if we defined a method:

```
String dimName(Dimension d) { return d.shortName(); }
```

then the method call `dimName(Time)` evaluates to `"T"` as well.

[3]Readers familiar with MixGen semantics may note that our restrictions on type parameters allow us to drop the use of MixGen's `with` clauses on type parameters.

In MetaGen, a class is allowed to define both a static method and an instance method of the same name. However, this allowance introduces the potential for ambiguity when a method so defined is called from within an instance method of the class it is defined in. Our solution is simply to require references to static members from instance contexts to explicitly denote the receiver. The receiver of a static method call may be any legal type, including a type parameter.

Static methods of `C` must not conflict with `final` instance methods of `C`'s kind and its superclasses. As with MixGen, static methods must not be abstract. The static members of a class define the members of that class *when considered as a value.* They are not members of a class's instances, nor are they inherited by subclasses. For example, we cannot write the expression `Meter.shortName()`.

3.1.2 Constructors and instance classes

As with instance methods, the instance fields of a kind `D` of `C` are static fields of `C`. Because these fields are normally initialized with a call to a constructor of `D`, we must answer the question of how they are initialized for an instance class such as `C`. We allow such fields to be initialized as follows: class `C` is allowed a single isolated constructor call to its kind in a static block. This constructor call must occur before all other static actions in the class definition. The new keyword `container` is used to designate this call.

For example, we define an instance class of class

```
Measurement<Length,Mile>
```

that we name `CircumferenceOfEarth`. This class must call the constructor of `Measurement` from a static block:

```
Measurement<Length,Mile> CircumferenceOfEarth {
   static { container(24889); }
}
```

Then `CircumferenceOfEarth`, viewed as an instance, has a single field `_magnitude` initialized with value `24889`. Viewed as a class, it has a single static field `_magnitude` with value `24889`.

If the immediate containing class of a class `C` is known to have a zeroary constructor and the definition of `C` does not include a call to a containing constructor in a static block, then `C` acts as if the static block `static { container(); }` were inserted at the beginning of the body of `C`.

3.1.3 Classes as values

Henceforth, we use *variable* to refer to fields, local variables, and method parameters (but not type parameters).

A class can be assigned to a field or variable and passed as an argument to a method, provided that it is an instance class of the variable's type.

In general, a variable may be bound to either an instance class or an ordinary (non-class) instance. Member accesses (i.e., field accesses and method calls) to a variable bound to an instance class are interpreted as accesses to static members of the instance class. Member access to a non-class instance behaves in the usual way.

A variable may not be used in type contexts (e.g., the type of a method parameter or the type specified in a cast operation), even if it is bound to an instance class. Also, there is no way to call a constructor of a bound instance class on a variable it is bound to (we are intentionally ignoring interactions of metaclasses with reflection facilities). For example, the constructor call in the following code is not allowed:

```
Length measurementMaker(Measurement<Length,Mile> m) {
   return new m(); // not allowed
}
```

A type parameter of kind `K` can be used in all type contexts. It can also be used in any context requiring a value of class `K`. In particular, it can be assigned to variables of class `K` and instance methods of `K` may be called on it, just like ground instances or other variables of class `K`. For example, we include the following method definition in class `Unit<D>`:

```
class* Unit<Dimension D> extends D {
  ...
  String dimName() { return D.shortName(); }
}
```

Furthermore, `Minute.dimName()` evaluates to `"T"`.

With this core semantics in hand, we now discuss advanced aspects of our library definition, as well as some peripheral language features motivated by it. Many of these additional aspects are quite complex, but this complexity results precisely from our intention to provide programmers with the behavior they intuitively expect. It is often the case with language design that a semantics that correctly matches programmers' intuitive expectations necessarily includes subtleties that are seldom thought about or even noticed by most users of the language.

3.2 Unit conversion and primary units

To convert between measurements in different units of the same dimension, we must specify conversion factors between various units of that dimension. A natural place to keep this information is in the definition of a unit: each unit specifies how to convert measurements in that unit to measurements in any other defined unit (for the same dimension). Although the number of such conversion factors is quadratic in the number of units, it is not necessary to maintain so many factors explicitly: if we can convert between measurements in units A and B, and between measurements in units B and C, then we can convert between measurements in A and C via B. Thus, it is sufficient to include in the definition of every unit a single conversion factor to a *primary unit* of that dimension, and convert between any two commensurable units via their common primary unit.

```
class* Unit<Dimension D> extends D {
   final private Real _primaryConversion;

  Unit(Real primaryConversion) {
     if (primaryConversion == 0) {
        throw new RuntimeException(
           "Attempt to define a unit with a 0 conversion."
        );
     }
     _primaryConversion = primaryConversion;
  }

  Real getPrimaryConversion() { return _primaryConversion; }

   <Unit<D> U> Real conversionFactor() {
      return _primaryConversion / U.getPrimaryConversion();
   }

    <Unit<D> U> Measurement<D,U> inUnit() {
       return new Measurement<D,U>(conversionFactor<U>());
   }
   ...
}
```

The introduction of primary units also allows us to avoid cyclic dependencies in unit definitions.

Note that the constructor of class `Unit<D>` ensures that no instances of `Unit<D>` will have a `_primaryConversion` of 0. Because this constructor is called in static blocks of each instance class, we catch such violations when classes are loaded.

We still need to reify primary units with separate classes so we can refer to them. Given a dimension `D`, we want to be able to easily obtain its primary unit. We also need to ensure that each dimension has exactly one primary unit. We meet both of these requirements by defining a new class `PrimaryUnit`, parameterized by a dimension. Each instantiation of this class with a particular dimension `D` is the primary unit of `D`. We enforce that class `PrimaryUnit` has no subclasses or instances by introducing a new modifier, `final*`:

```
final* Unit<D> PrimaryUnit<Dimension D> {
  static { container(1); }
}
```

The `final*` keyword indicates that `PrimaryUnit` has no subclasses or instances. Conceptually, we view `final*` classes as *typed terminal values*; they act like ordinary values with unique types that could be represented in a more clumsy fashion with the Singleton Pattern.

This solution gives a syntactically enforced bijection between dimensions and primary units: the primary unit of dimension `D` is `PrimaryUnit<D>`. For convenience and readability, we use aliases to identify primary units with familiar units. For example,

```
alias Meter = PrimaryUnit<Length>;
```

Using the conversion factors defined by the units, we can easily convert between measurements in different units:

```
class Measurement<Dimension D, Unit<D> U> extends D {
   Real magnitude;
   ...
   <Unit<D> V> Measurement<D,V> inUnit() {
      return new Measurement<D,V> (
        magnitude * U.conversionFactor<V>()
      );
   }

   Measurement<D,PrimaryUnit<D>> asPrimary() {
      return new Measurement<D,PrimaryUnit<D>> (
        magnitude * U.getPrimaryConversion()
      );
   }
   ...
}
```

The methods `inUnit` and `asPrimary` require calling a static method on the type parameter `U`. Static checking ensures that each instance class of `Unit<D>` defines `conversionFactor` and `getPrimaryConversion`.

We can also name commonly used, algebraically formed dimensions using type aliases. For example, we use `Area` to refer to the dimension $\texttt{Length}^2$ and `Speed` to refer to $\frac{\texttt{Length}}{\texttt{Time}}$. We now turn turn to the question of how to handle algebraically formed dimensions such as these.

3.3 Dimensional algebra

Like quantities, dimensions and units can be combined via multiplication and division operators. We might (naïvely) try to model these operators as classes parameterized by their operands:

```
Dimension DimProd<Dimension D, Dimension E> {...}
Dimension DimQuot<Dimension D, Dimension E> {...}
```

This model is unsatisfactory because the algebraic structure of dimensions requires that we equate certain terms. For example, $\texttt{Length}\times\texttt{Time} = \texttt{Time}\times\texttt{Length}$. Furthermore, the units of a given dimension can be combined as well. Combined units obey algebraic rules, and the subtyping relationships between units and their dimensions should be respected during algebraic manipulation.

Another complication is that combined forms of dimensions and units may contain integers. For example, consider the parameter type of a `sqrt` method over quantities (where we part with the Java Programming Language specification by using `^` to refer to exponentiation rather than exclusive or):

```
<Dimension D> D sqrt(D^2 x) {
  ...
}
```

Notice that the parameter type contains a numeric exponent (namely `2`). We want to allow combined dimensions that can be raised to (constant) integer exponents.

3.3.1 Abelian classes

Rather than support algebraic manipulation in an ad hoc fashion, we introduce a more general language extension that gives us the expressiveness we need: a new (special) metaclass `abelian class`. As with `class*`, a class of kind `abelian class` must have only instance classes. The explicitly declared instance classes of an abelian class form the identity and basis of a free abelian group.[4] Metaclass `abelian class` is parameterized by an *identity* element that designates the instance class that serves as the identity for the group. For notational convenience, we also parameterize `abelian class` with two symbols, included in parentheses, that denote (*i*) the binary operator of the group and (*ii*) a *repetition operator* of the group, which takes an element of the abelian group and an integer n, and yields the result of repeated application of the binary operator to the element n times (where, if n is negative, application to an element n times is defined to denote the inverse of the result of application to that element $|n|$ times). We write an instantiation of type `abelian class` as follows:

```
abelian class (bin_op,rep_op) <identity> name {...}
```

The repetition operator is given higher precedence than the binary operator. Because we are modeling a multiplicative semantics in this section, we use `*` as the binary operator and `^` as the repetition operator. For readability, we represent applications of `^` with superscripts.

We refer to the set of explicitly defined instance classes of an abelian class G as the *base types* of G. The set of instances of G is the set generated by the following rules:

- All base types of G are elements of G.
- If T and U are elements of G then $T{*}U$ is an element of G.
- If T is an element of G then for all $n \in \mathbb{Z}$, $T^{\mathtt{n}}$ is an element of G.

Additionally, generated elements are equated according to the following rules:

- $T{*}U = U{*}T$
- $T{*}(U{*}V) = (T{*}U){*}V$
- $T^0 = \mathit{identity}$
- $T^1 = T$
- $T^{\mathtt{m}}{*}T^{\mathtt{n}} = T^{\mathtt{m}+\mathtt{n}}$

These rules imply all of the ordinary rules for manipulating integer exponents that we make use of in this paper.

Now we can redefine class `Dimension` as an abelian class, with the special dimension `Unity` as its identity element:

```
abelian class (*,^) <Unity> Dimension {...}
Dimension Unity {...}
...
Dimension Length {...}
Dimension Time {...}
...
```

Then in addition to `Length`, `Time`, etc., class `Dimension` also contains $\texttt{Length}{*}\texttt{Time}^{-2}$, etc.

[4] An *abelian group* is a set A together with an operator, often denoted $+ : A, A \rightarrow A$, that satisfies the following properties:

- There exists $i \in A$ such that $a + i = a$ for all $a \in A$; i is called the *identity* of A.
- For any $a \in A$, there exists $a' \in A$ such that $a + a' = i$; a' is called the *inverse* of a in A.
- For all $a, a', a'' \in A$, $(a + a') + a'' = a + (a' + a'')$.
- For all $a, a' \in A$, $a + a' = a' + a$.

3.3.2 Abelian constructors

For instance classes `B` and `C` of an abelian class `A`, the class `B*C` includes (as static fields) all instance fields in the definition of `A`. It does not include any of the fields defined in `B` or `C`. Initialization of the fields of `B*C` is defined by a special constructor in `A` labeled by the binary operator of `A`. The arguments to this constructor must be exactly two elements of type `A`, denoting the two instance classes being combined to form the new instance. Similarly, initialization of $\texttt{B}^n$ is defined by a special constructor in `A` labeled by the repetition operator of `A`. This constructor takes an argument of type `A` and a `final int n`, denoting the numeric argument to the repetition operator. The argument `n` is constrained by static checking to be a constant (we discuss a generalization to nonconstant exponents in Section 4.1). We refer to both of these constructors as *abelian constructors*. Notice that an abelian constructor is shared by all algebraically formed elements of `A`.

If `A` is a generic class, we need to declare type parameters on the abelian constructors to denote the instantiations of the type parameters for a particular instance class. For example, we revise our definition of class `Unit` to be an abelian class, with `PrimaryUnit<Unity>` as its identity element, and abelian constructors as follows:

```
abelian class (*,^) <PrimaryUnit<Unity>> Unit<Dimension D>
  extends D
{
  final private Real _primaryConversion;

  <Dimension E, Dimension F> *(Unit<E> V, Unit<F> W) {
      this(V.getPrimaryConversion().multiply(
             W.getPrimaryConversion()
      ));
  }

  <Dimension E> ^(Unit<E> V, final int n) {
      this(V.getPrimaryConversion().power(n));
  }
  ...
}
```

For generic abelian classes, we impose restrictions on the respective instantiations of type parameters for the elements of a combination. The instantiations of a type parameter whose kind is not an abelian class must match; the type parameter of the combination is instantiated with the same type. The instantiations of a type parameter whose kind is an abelian class are combined pointwise in the new instantiation. For example, because `Meter` is an instance class of `Unit<Length>` and `Second` is an instance class of `Unit<Time>`, `Meter*Second` is an instance of `Unit<Length*Time>`. In general, the type `Unit<E>*Unit<F>` equals the type `Unit<E*F>`. Likewise, `Unit<E>^n` equals `Unit<E^n>`.

The identity element of a generic abelian class is also constrained: every type parameter of an identity element whose kind is an abelian class must be instantiated with the identity element of its kind. For example, the identity of `Unit<D>` must be an instance of `Unit<Unity>` because `Unity` is the identity of `Dimension`.

An algebraically formed instance of an abelian class `A` includes (as static methods) all instance methods defined in the definition of `A`. But an abelian class may also have abstract methods. For example, class `Dimension` includes the abstract method `shortName`. Abstract methods of an abelian class `A` may be defined for algebraically formed instance classes by defining them in the abelian class itself. Methods defined this way must include definitions of behavior under combination by both the binary operator and the repetition operator of `A`, and these definitions must be prefixed by the binary and repetition operators. The definition prefixed by the binary operator includes two type parameters of kind `A`. The definition prefixed by the repetition operator includes a type parameter of kind `A` and a `final int` parameter. For example, abstract method `shortName` is defined for algebraically formed instance classes of `Dimension` by adding the following method definitions to class `Dimension`:

```
<Dimension E, Dimension F>
   * String shortName() {
      return E.shortName() + "*" + F.shortName();
}

<Dimension E>
   ^ String shortName(final int n) {
      return E.shortName() + "^" + n;
}
```

In the example above, does `Time*Length.shortName()` evaluate to `"L*T"` or `"T*L"`? The answer would appear to depend on the order of the arguments. However, by the algebraic properties of abelian classes, `Time*Length` is the same as `Length*Time`. To allow methods such as `shortName` to be well-defined, we need to canonicalize all combined abelian types. More importantly, canonicalization is a natural way to statically determine type equivalence. We discuss a procedure for canonicalization in the next section.

Before any constructors are called on an algebraically formed instance of an abelian class, the instance is canonicalized. After canonicalization, the terms $\texttt{B}^n$ of the canonicalized product are initialized from left to right via the abelian constructor prefixed by the repetition operator. Then combinations of these terms are initialized from left to right via the abelian constructor prefixed by the binary operator.

3.4 Canonicalization of abelian type references

To statically check type equivalence between members of an `abelian class` G, we identify members of G and put them in canonical form so they can be compared directly.

Ignoring for the moment the complication of type parameters, elements of G can be identified syntactically: they are simply type expressions whose base types are all elements of G that are combined solely via the two operators of G. To put elements of G into canonical form, we perform the following steps:

1. Replace all occurrences of a base type T (no exponent) with T^1.
2. Distribute exponents so we are left with a product of base types that are raised to various powers.
3. Eliminate all parentheses.
4. Put base types into lexicographic order.

5. For each occurrence of base type T, combine all tokens T^{n_i} into a single token $T^{\sum n_i}$.
6. Eliminate all base type tokens of exponent `0`.
7. Eliminate all occurrences of the identity element of G.
8. If nothing is left but the empty product, add a single occurrence of the identity element.

For example, the (improbable) dimension

$$(\texttt{Length}^2 * \texttt{Time}^{-2} * \texttt{Unity})^4 * \texttt{Length}^{-8} * \texttt{Time}^8$$

is canonicalized as follows:

$$\begin{aligned}
&\mapsto (\texttt{Length}^2 * \texttt{Time}^{-2} * \texttt{Unity}^1)^4 * \texttt{Length}^{-8} * \texttt{Time}^8 \\
&\mapsto (\texttt{Length}^8 * \texttt{Time}^{-8} * \texttt{Unity}^4) * \texttt{Length}^{-8} * \texttt{Time}^8 \\
&\mapsto \texttt{Length}^8 * \texttt{Time}^{-8} * \texttt{Unity}^4 * \texttt{Length}^{-8} * \texttt{Time}^8 \\
&\mapsto \texttt{Length}^8 * \texttt{Length}^{-8} * \texttt{Time}^{-8} * \texttt{Time}^8 * \texttt{Unity}^4 \\
&\mapsto \texttt{Length}^0 * \texttt{Time}^0 * \texttt{Unity}^4 \\
&\mapsto \texttt{Unity} \\
&\mapsto \epsilon \\
&\mapsto \texttt{Unity}
\end{aligned}$$

Once elements of G are in canonical form, type equivalence can be determined via syntactic equivalence.

Generalizing to allow type parameters of kind G is straightforward: In a lexical scope binding new type parameters, we identify type parameters of kind G as elements of G, just as if they were new base types of G. Elements of G can be built with these type parameters just as they can with true base types. Of course, these type parameters may be instantiated with elements of G that are not base types; we merely treat them syntactically as base types within the scope that binds them. Within that scope, canonicalized elements of G are equated if and only if they are syntactically equivalent.

References to instance classes of abelian classes that are syntactically contained in generic type instantiations are also canonicalized. For example, because `PrimaryUnit` is parameterized by a dimension, our solution for algebra over `Dimension` automatically allows us to form and compare primary units of compound dimension. Thus,

```
PrimaryUnit<Length*Time>
```

equals

```
PrimaryUnit<Time*Length>
```

For abelian classes with type parameters, an additional step is necessary for canonicalization: instantiations of common generic abelian classes are combined pointwise. For example,

```
Unit<Length>^2*Unit<Length*Time>
```

is canonicalized to:

```
Unit<Length^3*Time>
```

3.5 Arithmetic operations on quantities

To use an arbitrary instance of a dimension `D` in arithmetic contexts, we need to declare that instances of `D` define arithmetic operations; that is, all instances of `D` must include a set of arithmetic methods. To express such a restriction, we add a new class modifier to our language. To motivate this new modifier, let us reconsider our metaphor of classes as sets. We can understand conventional class extension declarations `class C extends D` as an assertion of the following form:

$$\forall x \in C \,.\, x \in D$$

When the only relationship between classes is extension, this simple form of assertion is sufficient to state the most essential relationships. Once we generalize our language to include metaclasses, several new assertions suggest themselves. For example, consider a class hierarchy for biological applications with classes `Species` and `LivingThing`. We want to declare that all instances of `Species` are subtypes of `LivingThing`.

In general, we may want to make assertions of the following form, relating classes to their subclasses and instances:

$$\forall x \in C \,.\, x \subseteq D$$

To express this new relation, we add an optional `extends*` clause to the header of a class definition. For example, we can represent the constraint we would like on instances of `Species` as follows:

```
abstract class LivingThing {}
abstract class Species extends* LivingThing {}
Species Human extends LivingThing {}
Species Lion extends LivingThing {}
```

Then `Human` and `Lion` would be invalid class definitions if they did not extend `LivingThing`. We also allow for an `implements*` clause that constrains instance classes to implement designated interfaces.[5]

As with instances of a class of kind `class*`, all instances of a class that includes either an `extends*` or an `implements*` clause in its definition must be instance classes.

We add an `implements*` clause to class `Dimension`, requiring that each instance class `D` of class `Dimension` implements an interface `DimensionI` that contains a set of arithmetic operations. In this way, we ensure that each instance of `D` will define those methods.

The signatures of the arithmetic methods of an instance of dimension `D` must refer to `D` itself:

```
D add(D x);
D subtract(D x);
<Dimension E> D*E multiply(E x);
<Dimension E> D*E^-1 divide(E x);
<Unit<D> U> Measurement<D,U> inUnit();
```

The method `inUnit` allows us to convert an arbitrary quantity of `D` into a measurement in a given unit `U` of `D` so we can add and subtract it to another measurement in `U`.

[5] Of the two remaining variations, $\forall x \subseteq C \,.\, x \subseteq D$ is logically equivalent to the constraint for ordinary subtyping, and we know of no motivation in practice for stipulating constraints of the form $\forall x \subseteq C \,.\, x \in D$.

How can we put these declarations in an interface in such a way that the interface has a handle on the type of an instance class of `Dimension`? We need to include a type parameter on the interface to represent the type of an instance class of `Dimension`. The bound on this type parameter will therefore be our interface instantiated with the type parameter itself:

```
interface DimensionI<DimensionI<This> This> {
   <Unit<This> U> Measurement<This,U> inUnit();
   This add(This x);
   This subtract(This x);
   <Dimension That> This*That multiply(That x);
   <Dimension That> This*That^-1 divide(That x);
   <Unit<This> U> Measurement<This,U> inUnit();
}
```

We now need to redefine `Dimension` to declare that each of its instance classes implements an instantiation of this interface with itself. We do that as follows:

```
abelian class (*,^) <Unity> Dimension<Dimension<This> This>
  implements* DimensionI<This> {...}
```

Our instance classes of `Dimension` now need to instantiate this parameter with themselves:

```
Dimension<Length> Length {...}
Dimension<Time> Time {...}
```

The tight bound on parameter `This` ensures that each instance class of `Dimension` must instantiate `This` with itself. Therefore, it is straightforward to introduce syntactic sugar that implicitly includes a type parameter `This` on every class of kind `class*` or `abelian class` and on every class that includes an `extends*` or `implements*` clause. We need only reserve the identifier `This` as a new keyword.

With this syntactic sugar, our class definitions revert almost to their earlier form:

```
abelian class (*,^) <Unity> Dimension
   implements* DimensionI<This> {...}
Dimension Length {...}
Dimension Time {...}
...
```

3.6 Zero quantities

When we introduced the notion of dimension, we said that if two quantities Q and Q' exist in the same dimension, we can answer the question, "how much of Q does it take to equal Q'?" If it takes x of Q to equal Q' then we write $Q' = xQ$. We now discuss a minor complication with this naïve view.

Consider the ability of two quantities to be related in this way as a relation R. Because we naturally think of dimensions as dividing quantities into disjoint sets, we might expect that R is an equivalence relation. Unfortunately, our expectation is not quite right: although R is reflexive, it is neither symmetric or transitive. The problem lies with *zero quantities*: it takes 0 of any quantity to equal a zero quantity, but no amount of a zero quantity equals a nonzero quantity. Furthermore, the symmetric closure of R does not satisfy transitivity. For example, suppose QRQ' and $Q'RQ''$. Then it may be the case that $Q' = 0Q$ and $Q' = 0Q''$, but that QRQ'' does not hold. Strictly speaking, we should define the notion of dimension for nonzero quantities, and then add in the notion of the zero quantity as special.

Given this special status of the zero quantity, it is natural to ask whether there are any software engineering repercussions as to how we treat it. In fact, there are [21, 27]. We would like to allow programmers to write down zero quantities without the tedium of having to add "dummy" units. For example, we want the following code to be legal:

```
Length x = <some expression>;
if (x == 0) {...}
```

Treating the zero quantity as polymorphic in this way is harmless: If we can statically determine that a quantity is zero, then no harm can come from adding it to, or multiplying it by, an arbitrary quantity; in the former case, the result is simply the quantity it is added to; in the latter case the result is simply the zero quantity. On the other hand, if we cannot statically determine that a quantity is zero, then static checking will prevent it from being used polymorphically.

To include a special zero quantity, we might consider restricting magnitudes of measurements to be nonzero and then adding a new typed element `Zero` of class `Measurement`:

```
Measurement<D, U> Zero<Dimension D, Unit<D> U> {}
```

However, we do not want a separate `Zero` for each dimension and unit. Rather, we want a single polymorphic zero quantity that is an instance of all instantiations of `Measurement`. We express this relationship by using *wildcard instantiations* of type parameters of generic classes.

If `T<C>` is a generic type, then we can instantiate `C` with a wildcard, written `T<*>`. An instance of `T<*>` is an instance of all instantiations of `T`. Normally we cannot say such things in object-oriented languages with generic types because the notions of pointwise extension of a generic type and of extension of all instantiations of a generic type are conflated (e.g., `Zero<D,U> extends Measurement<D,U>`). But it is helpful to distinguish these notions. Because the programmer has no handle on the type argument of a wildcard instantiation of a parent class, it is syntactically impossible for the definition of such a class to depend on those arguments.

Using wildcards, we define the zero quantity to be a wildcard instantiation of `Measurement`:

```
Measurement<*,*> Zero {...}
```

Although adding wildcards may seem excessive just to get a polymorphic zero quantity, wildcards are useful in other contexts. For example, wildcard instantiations are similar to a standard interpretation of many type hierarchies as they are written in a structurally typed languages such as ML and Haskell, where parametric types are erased after type checking and do not affect run-time behavior. For example, lists are typically modeled in ML as follows:

```
'a List = Empty | 'a Cons
```

In this definition, there is one `Empty` list shared by all instantiations of `List`. In contrast, Generic Java without wildcard

instantiations forces us to include a separate `Empty` for each instantiation of `List` [6]:

```
class Empty<T> extends List<T> {...}
```

Using wildcards, we can define class `Empty` as:

```
List<*> Empty {...}
```

4. EXTENSIONS

We have presented our framework in the context of a general-purpose object-oriented type system, adding only metaclasses, abelian classes, and wildcards. With just these three extensions, we are able to express the most common uses of physical quantities in computations. In particular, we can express user-defined dimensions and units, automatic conversion between units of a given dimension, static types corresponding to dimensions and units, parametric polymorphism with respect to these types, dimensional algebra, and zero quantities of polymorphic dimension. Furthermore, our formulation is general and robust enough to extend easily in many ways to support additional static checking. In this section, we discuss some of these ways.

4.1 Nonconstant exponents

In general, we prefer that exponents on abelian classes not be limited to be constants. For example, we would like to define a `power` function over quantities:

```
/** @precondition n >= 0 */
<Dimension D> D^n power(D x, int n) {
  if (n == 0) return 1;
  else return x.multiply(power(x, n-1));
}
```

where the value `1` is autoboxed to a value of type `Unity`. Employing pure recursion is of course undesirable in a math library function, but the specific implementation is irrelevant for our present purposes; what is relevant is that we are able to type check even recursive definitions.

Parameterizing dimensions by run-time integer values is a form of dependent typing. Because we cannot statically determine precisely which integer computations will result in equal values when the program is run, we must, as with all static checking, settle for a conservative approximation.

One approximation that allows us to check many practical examples *and* leverage the machinery we have already developed is to treat the set of `int`s as an abelian class![6] The binary operator of this class is addition; the repetition operator is multiplication. Each `int` constant is treated as a distinct type.

A `final int` bound in a lexical scope is treated as a type parameter ranging over this abelian class. Thus, the set of valid `int` types consists of all expressions of `int` variables and constants formed via multiplication and addition. We identify and canonicalize elements of this type in the same way as explained for general abelian classes. However, special properties of the `int`s allow us to equate even more terms than we can for an ordinary abelian class. Specifically, because the set of `int`s is the only abelian class with the property that the second, numeric, argument to the repetition operator is itself a element of the class, and the repetition operator represents multiplication, we allow additional canonicalization for this special class.[7] Automatic simplification of arithmetic expressions is well understood, and is incorporated in many production systems, including proof checkers such as Isabelle and symbolic math tools such as Mathematica.

To support nonconstant exponents in type expressions in other abelian classes (e.g., `Dimension`), we need to add one step to the canonicalization process in Section 3.3, between steps 5 and 6:

5.5. Canonicalize all exponents.

We can now type check our `power` method, provided the parameter `n` is declared to be `final` (allowing it to be used as a type):

```
/** @precondition n >= 0 */
<Dimension D> D^n power(D x, final int n) {
  if (n == 0) return 1;
  else return x.multiply(power(x, n-1));
}
```

4.2 Generalizing magnitudes to arbitrary fields

Note that a magnitude need not be a real number. For example, the magnitude of an impedance is often represented as a complex number.[8]

We want to generalize our treatment of class `Measurement` so that magnitudes may be elements of various numeric classes (e.g., `Float`, `Double`, `Complex`, etc.). We define a new class `AlgebraicField` that contains classes representing algebraic fields. We want to be able to perform arithmetic operations on any two elements of an `AlgebraicField`. But we also want to require that elements of an `AlgebraicField` representing magnitudes can be combined via arithmetic operations with elements of any other `AlgebraicField` representing magnitudes (e.g., we want to allow a complex magnitude to be multiplied by a real magnitude). We also want to be able to combine dimensioned quantities with arbitrary magnitudes via multiplication and division.

We can satisfy all of these requirements by extending class `AlgebraicField` with a class `MagnitudeField` and requiring every instance class of `MagnitudeField` to extend dimension `Unity`.

```
class* MagnitudeField extends AlgebraicField
   extends* Unity {}
```

[6]We assume that autoboxing of primitives is built into the language, as it is in the Java™ 2 SDK1.5.

[7]Alternatively, we could define another special metaclass `commutative ring` and define `int` as an instance class of that kind. But this approach requires adding additional complexity to our language with little reward. Furthermore, even the notion of a commutative ring does not capture all of the algebraic simplification possible with the integers.

[8]We use the term "magnitude" as it is employed in most contexts concerning dimension checking. However, this use differs from its typical use with regard to impedance, to refer to the square root of the sum of the squares of the real and imaginary parts of what we call the magnitude.

Because class `Unity` implements `DimensionI<Unity>`, all instance classes of `MagnitudeField` must implement it as well. Thus, an element of an instance class of `MagnitudeField` can be added to or subtracted from any other element of an instance class of `MagnitudeField`. It may also be multiplied by and divided by any other quantity. When an element e of an instance class of `MagnitudeField` is multiplied by another element e' of an instance class of `MagnitudeField`, the dimension of the result is $\texttt{Unity} * \texttt{Unity} = \texttt{Unity}$. When e is divided by e', the dimension of the result is $\texttt{Unity} * \texttt{Unity}^{-1} = \texttt{Unity}$. For quantities of `Dimension D`, the dimension of a multiplication is $\texttt{Unity} * \texttt{D} = \texttt{D}$ and of a division is $\texttt{Unity} * \texttt{D}^{-1} = \texttt{D}^{-1}$.

`DimensionI<Unity>` also includes method `inUnit`, but the only unit of dimension `Unity` is `PrimaryUnit<Unity>`. Therefore, we define this method as follows: if it is called on an instance of `Unity` and the given unit to convert to is `PrimaryUnit<Unity>`, then the method simply returns its receiver. Otherwise, it throws an exception.

We redefine existing classes `Float` and `Double` as instance classes of kind `MagnitudeField` and provide them with appropriate definitions of the arithmetic operations required to satisfy the `extends*` constraint on `MagnitudeField`:

```
MagnitudeField Float extends Unity {...}
MagnitudeField Double extends Unity {...}
```

Now we need only modify field `_magnitude` in class `Measurement` to be of type `Unity`. In this way, we allow the magnitude of a measurement to be any dimensionless quantity. Note that dimensionless products of measurements are both measurements and instances of `Unity`. A definition of a `Measurement` class with methods for binary operations and for unit conversion is presented in Figure 1. Class `Measurement` includes several polymorphic methods with type parameters that cannot be inferred from method arguments alone. As in MixGen, polymorphic methods in MetaGen employ explicit polymorphism.

We can also define new instance classes `Complex`, `Rational`, `Real`, `Interval`, etc. with kind `MagnitudeField`, along with appropriate subclassing relationships between them. A proper encoding of these classes and their relation to one another is beyond the scope of this paper.

4.3 Discovered dimensions

Sometimes the combination of measurements results in dimensionless quantities (i.e., quantities of dimension unity). For example, we can think of an angle as being determined by measuring two lengths (an arc and a radius) and dividing one by the other. Similarly, a relative density is obtained by dividing one density by another. Although angle and relative density are both of dimension unity, it is nonsense to compare an angle to a relative density.

Therefore, a programmer may want to view (nonzero) measurements as measurements of quantities in newly discovered primary dimensions (e.g., angle, relative density, etc.). It is often possible to measure such quantities directly (e.g., with a protractor).

We leave the decision of when to treat a dimensionless quantity in this way to the programmer. Our formulation allows a programmer to easily convert dimensionless quantities to dimensioned quantities when necessary. Because dimensionless quantities are identified with elements of class `Unity`, which are also used to represent magnitudes of `Measurement`s, a programmer wishing to treat such a quantity as having a dimension other than unity can simply wrap it in a new `Measurement`. For example, to treat a dimensionless quotient of two `Length`s `l1` and `l2` as an angle, we can write:

```
new Measurement<Angle,Radian>(l1.divide(l2))
```

4.4 Scales of measurement

Along with multiplicative conversions between units of the same dimension, we often want to define scales with explicit zeros that are based on a unit of measurement (e.g., the Celsius, Kelvin, and Fahrenheit scales). Scale conversions may involve more than just multiplication by conversion factors; they can involve addition and subtraction as well (e.g., the conversion of a Celsius scale reading to a Fahrenheit scale reading involves multiplication by 9/5 and addition of 32 F°, where we follow the convention that n°C means a Celsius scale reading of n but nC° means a distance of n degrees between two readings on the scale). How are we to understand scales and scale conversions in the context of our analysis? In particular, can we explain scale conversion solely in terms of manipulation of quantities?

The key to understanding the relation between quantities and scale readings is to realize that a measurement of a quantity is really a measurement of the difference between two points in a continuum: an *origin* (or *zero point*) and an *external point*. For example, a measurement of time is really a measurement between an origin (e.g., the point at which a stopwatch is started) and an external point (the point at which the watch is stopped). Scale readings are measurements for which the origin is fixed by the scale itself. Thus, even when two scales are applied to the same external point, the quantities represented by the readings may not be the same—the scales may have different origins. When we convert a reading R of external point M in scale S with zero point Z_S to a reading R' in another scale S' with zero point $Z_{S'}$ we are really deducing the size of a measurement from $Z_{S'}$ to M given both a measurement from Z_S to M and a measurement from $Z_{S'}$ to Z_S. This deduction involves a simple addition. Because scales may also represent quantities in different units, unit conversions of the initial measurements may be necessary as well.

For example, a reading of a temperature in degrees Celsius is really a measurement of difference of an external "temperature point" with a reference point (0°C) in unit C°. If we write the external point as A and 0°C as Z then we can express this difference with the following equation:

$$A - Z = x\,\mathrm{C}^\circ$$

Now suppose we want to convert the measurement $x\,$C° to a measurement on the Fahrenheit scale. We represent 0°F as Z_F. Then we have:

$$Z - Z_F = 32\,\mathrm{F}^\circ$$

We also know the conversion factor of C° to F°:

$$\mathrm{C}^\circ = \frac{9}{5}\mathrm{F}^\circ$$

```
class Measurement<Dimension D, Unit<D> U> extends D {
   private final Unity _magnitude;

   Measurement(Unity magnitude) {  _magnitude = magnitude; }

   Unity getMagnitude() { return _magnitude; }

   Unity getPrimaryConversion() { return _magnitude.multiply(U.getPrimaryConversion()); }

   <Unit<D> V> Measurement<D,V> inUnit() {
      if (U == V) return this;
      else return new Measurement<D,V>(getMagnitude().multiply(V.conversionFactor<U>()));
   }

   // Arithmetic operations are overloaded for cases where more specific arguments can be statically determined.

   // add
   D add(D x) { return new Measurement<D,U>(getMagnitude().add(x.inUnit<U>().getMagnitude())); }

   Measurement<D,U> add(Measurement<D,U> x) { return new Measurement<D,U>(getMagnitude().add(x.getMagnitude())); }

   // subtract
   D subtract(D x) { return new Measurement<D,U>(getMagnitude().subtract(x.inUnit<U>().getMagnitude())); }

   Measurement<D,U> subtract(Measurement<D,U>) { return new Measurement<D,U>(getMagnitude().subtract(x.getMagnitude())); }

   // multiply
   <Dimension E> D*E multiply(E x) {
      return new Measurement<D*E,U*PrimaryUnit<E>>(getMagnitude().multiply(x.inUnit<PrimaryUnit<E>>().getMagnitude()));
   }
   <Dimension E, Unit<E> V> Measurement<D*E,U*V> multiply(Measurement<E,V> x) {
      return new Measurement<D*E,U*V>(getMagnitude().multiply(x.getMagnitude()));
   }
   <Dimension E, Unit<E> V> Measurement<D*E,U*V> multiply(V v) { return new Measurement<D*E,U*V>(getMagnitude()); }

   Zero multiply(Zero x) { return Zero; }

   // divide
   <Dimension E> D*E^-1 divide(E x) {
      return new Measurement<D*E^-1,U*PrimaryUnit<E^-1>>(getMagnitude().divide(x.inUnit<PrimaryUnit<E>>().getMagnitude()));
   }
   <Dimension E, Unit<E> V> Measurement<D*E^-1,U*V^-1> divide(Measurement<E,V> x)  {
      return new Measurement<D*E^-1,U*V^-1>(getMagnitude().divide(x.getMagnitude()));
   }
   <Dimension E, Unit<E> V> Measurement<D*E^-1,U*V^-1> divide(V v)  {
      return new Measurement<D*E^-1,U*V^-1>(getMagnitude());
   }
   Unity divide(D x) { return getMagnitude().divide(x.inUnit<U>().getMagnitude()); }

   Unity divide(Measurement<D,U> x) { return getMagnitude().divide(x.getMagnitude()); }
}
```

Figure 1: Class `Measurement`

To derive an equation of the form $A - Z_F = y\mathrm{F}^\circ$, we add the two difference equations and perform unit conversion:

$$\begin{aligned} A - Z_F &= x\,\mathrm{C}^\circ + 32\,\mathrm{F}^\circ \\ &= x\frac{9}{5}\mathrm{F}^\circ + 32\,\mathrm{F}^\circ \\ &= (\frac{9}{5}x + 32)\,\mathrm{F}^\circ \end{aligned}$$

yielding the standard formula. By viewing scale readings as measurements from fixed reference points, we can convert between scales while manipulating only quantities.

We can model scales and points similarly to how we model measurements and units. Just as every dimension has a primary unit (which prevents cyclic dependencies), so too it has a primary origin. Every point in a given dimension is defined with respect to the primary origin of that dimension. Just as measurements are defined with respect to a particular unit, scales are defined with respect to a unit and origin. Therefore, we define a class `Point` as follows:

```
class Point<Dimension D> {
   private final D _distanceToOrigin;

   Point(D distanceToOrigin) {
      _distanceToOrigin = distanceToOrigin;
   }

   D getDistanceToOrigin() {
      return _distanceToOrigin;
   }
   D distanceTo(Point<D> p) {
      return _distanceToOrigin - p.distanceToOrigin();
   }
}
```

Unlike `Units`, we do not require all points to be instance classes, so we do not define `Point` as having kind `class*`. We also do not define `Point` as an abelian class.

As with `PrimaryUnits`, we define an instance class `PrimaryOrigin` that is parametric in a dimension:

```
final* Point<D> PrimaryOrigin<Dimension D> {
   static { container(0); }
}
```

We can now define classes `Scale` and `ScaleReading` that are parametric in a dimension, a unit, and a point:

```
class Scale<Dimension D, Unit<D> U, Point<D> P> {
   ScaleReading<D,U,P> newReading(Point<D> Q) {
      return new ScaleReading<D,U,P>(Q.distanceTo(P));
   }
}
class ScaleReading<Dimension D, Unit<D> U, Point<D> P> {
   private final Measurement<D,U> _value;

   ScaleReading(Measurement<D,U> value) { _value = value; }

   Measurement<D,U> getValue() { return _value; }

   <Unit<D> V, Point<D> Q> ScaleReading<D,V,Q> inScale() {
      return new ScaleReading<D,V,Q> (
         _value.subtract(P.distanceTo(Q)).inUnit<U>()
      );
   }
}
```

A `ScaleReading` can be converted to any other `ScaleReading` of the same dimension via method `inScale`. Class `Scale` is merely a factory for new `ScaleReading`s. We may want to alias particular instantiations of `Scale` with their common names, e.g.,

```
alias KelvinScale =
  Scale<Temperature,CelsiusDegrees,PrimaryOrigin<Temperature>>;

final* Point<Temperature> ZeroCelsius {
   static {
     container (
       new Measurement<Temperature, CelsiusDegrees>(273)
     );
   }
}
alias CelsiusScale =
   Scale<Temperature,CelsiusDegrees, ZeroCelsius>;
```

4.5 Nonlinear scales

More generally, zeroed "scales" are also defined in which readings are nonlinear, e.g., the bel scale, which is a measure of the logarithm of the ratio of two measurements of `Power`, and the Gregorian calendar, where the length of a year depends on which year we're talking about. A particularly unusual nonlinear scale from science fiction is the warp scale from *Star Trek: The Next Generation*,[9] which is asymptotic at Warp 10.

Although such "scales" appear superficially to be distinct from linear scales such as Celsius and Fahrenheit, they really are the same kind of entity. Any scale can be thought of as linear along a particular dimension. In fact, temperature itself is equal to the reciprocal of the rate of increase of entropy with respect to energy of a given system. In the case of decibels, "absolute zero" corresponds to 0dB, and it is qualitatively different from other decibel readings. Differences can be taken between two readings in the dB scale; the result is expressed in a special unit (let us call it "decibel units") that is not typically written down.[10] Another example is the musical scale, which can be thought of as based on the unit "halftone". High C and middle C differ by 12 halftones. If we start from middle C, and then add 4 halftones, we get E, and if we add 8 more, we get high C. We can arbitrarily pick middle C as the 0 point, and count halftones from there. However, we can also associate a reading on the musical scale with the frequency of the note. A musical scale based on halftone is logarithmic in the frequency. But we might as easily say that the frequency scale is exponential in the halftone scale.

Because scale readings are often logarithmic with respect to other scales of more fundamental quantities (based on units that the designers of the original scale may not have been aware of), it is natural that we find readings on many scales to be restricted to proper subintervals of a field (e.g., the positive reals, the reals between 0 and 10, etc.).

[9] The warp scale from the original *Star Trek* series was based on a less interesting formula.

[10] Because a decibel reading is measured as the logarithm of the ratio of two quantities of `Power`, a decibel unit is like an angle in that it can be viewed either as dimensionless or as a unit in a discovered dimension.

5. PRAGMATICS

We now consider some pragmatic aspects of our system with respect to syntax and performance.

5.1 Syntax

Many of the syntactic constructs in our finished design are long-winded. However, this is an artifact of C++-style syntax that is easily addressed with syntactic sugar. For one thing, many of the instantiations of generic types involved instantiations of multiple type parameters for `Measurement`, `Dimension`, and `Unit` when instantiation of the `Unit` is all that is needed to infer the others. Also, we have selected long names for our classes to help communicate ideas; in a production system, we may want to use shorter names in some cases. Finally, support for operator overloading in our language would allow us to specify arithmetic operations on `Measurement`s more concisely. If we had autoboxing of numeric types and we could also overload arithmetic operations on numeric types to take a unit and return a measurement in that unit, many expressions could be written more succinctly. For example, we could replace expressions such as:

```
new Measurement<Double,Length,Meter>(5)
```

with `5 * Meter`.

5.2 Efficiency of our representation

Our formulation of dimensioned quantities is powerful, and it allows for implementation with reasonable performance: Our representation of measurements involves only a single field for each measurement. Elements of this field are elements of classes associated with primitive values such as `Float` and `Double`. Because the `_magnitude` field is `private` and `final`, rudimentary escape analysis should allow us to inline these field values in many cases. Thus, run-time representation of quantities should require overhead about equal to that of boxing primitive values. Additionally, various reductions in expressive power allow for even better performance, giving library designers the freedom to choose the power/performance tradeoff for their target applications. When run-time representation of quantities is not required, the `Measurement`s themselves could be represented as primitive values. Even when run-time representation of quantities is important, it should be possible to simply pass this information as type parameters to methods that require it and represent the measurement values themselves as primitives.

All the arithmetic methods are simple nonrecursive one-line methods that are easily inlined by modern just-in-time compilers. Through static overloading, calls to the arithmetic operations are more efficient when additional information can be statically inferred. For example, when the static types of the operands are `Measurement`s with the same `Unit`, no unit conversion is necessary. Additionally, field `_magnitude` is `final`; a savvy just-in-time compiler could calculate values for methods such as `getPrimaryConversion` on the first call for a value and cache the result for subsequent calls. If performance were still unsatisfactory, we could require all measurements to be kept in the primary unit of their dimension. Although we may lose precision for some programs, we would be able to reduce each of the four arithmetic methods to a single arithmetic operation.

6. RELATED WORK

6.1 Dimension and unit checking

Most formulations of dimension checking in programming languages encode dimensions as an ad hoc language feature rather than integrating them into a more general type system. Also, most formulations incorporate dimension checking into functional and procedural languages rather than object-oriented languages, where the design issues are significantly different. Few formulations support parametric polymorphism over dimensions. No previous formulation presents dimension checking as a generalization of static checking for metaclasses.

Andrew Kennedy presents an extension of a core calculus for ML with support for dimension checking [20, 21]. Because ML is a structurally typed functional language rather than a nominally typed object-oriented language, many of the key issues that have driven our design do not apply to this language. Units are not given separate types and each dimension is associated with a single unit. Other units are encoded by the programmer as constant values defined in terms of this single unit. Because dimensions are added as an ad hoc language extension, the issues involved in encoding them via metaclasses do not arise, nor do those involved in general support for kinds whose elements satisfy the properties of an abelian group. Nevertheless, Kennedy's approach shares similarities with ours. Dimension exponents are constrained to be integers. The zero quantity is treated as having polymorphic dimension. Also, function types may be polymorphic with respect to dimensions. The problem of typing functions that are polymorphic with respect to a dimension exponent (such as `power`) is discussed but no solution is presented.

A significant portion of Kennedy's proposal involves the formulation of type inference over dimension types. In this respect, it builds on the work of Wand and O'Keefe [28] and Goubault [16], which present formulations of dimension inference for ML-like languages. Incorporating an inference mechanism over metaclasses and abelian classes based on these inference systems is an important direction for future work.

Kennedy extends his own work by relating it to the parametricity result of System F [22]. By proving a "dimensional invariance" principle over a core calculus based on ML, Kennedy is able to establish several surprising properties of this calculus. For example, it is possible to show that functions of the type $\forall u\,.\,\texttt{num}\ u^2 \mapsto u$ cannot return a nonzero result for any argument value, and therefore that an approximating square root function with this type cannot be expressed in the language! It is not at all obvious how to relate this work, which is based on F-bounded parametric polymorphism over structural types, to an object-oriented language with typed metaclasses and nominal types. But it would be interesting to determine such a relationship.

Other formulations of dimension checking do not support parametric polymorphism over dimensions. Formulations of dimensions for object-oriented languages have typically encoded a fixed set of dimensions. A popular formulation of dimension checking in C++ (via the Standard Template Library) is `SIUnits` [5], which provides for the checking of quan-

tities in the SI system by encoding the seven dimensions addressed by that system as integer parameters of a template class `Quantity`. Parametric dimensions, class relationships between units dimensions and quantities, and programmer extensions of this hardwired template are not supported. In addition, the underlying type system is unsafe and does not directly represent the abstraction of generic types, greatly reducing the value of static checking.

Fowler discusses a modeling of units of measurement used in the Cosmos project for the U.K. National Health Service [10]. In addition to units and quantities, Fowler includes a notion of "measurement" that includes the notion of an observation and an observed "patient". Dimensions of quantities and the relationships of algebraically formed units and dimensions are not modeled. Static checking and dimensional parametricity are not discussed.

The formulation closest to our own is by van Delft [27], who describes a mechanism for implementing dimensions in Java class files as annotated classes. As with our approach, van Delft treats the zero quantity as having polymorphic dimension. However, van Delft does not provide a specification for the semantics of dimension types, does not encode dimensions as part of a class hierarchy, does not allow for the definition of new methods specific to a particular dimension, and does not include parametric polymorphism over dimension types or separate types for units. As a result, the issues in object-oriented modeling that we have addressed in our proposal do not arise. Van Delft does includes a construct `dimension(exp)`, which evaluates to the dimension of `exp`. Although this feature is useful for linking dimensions of method parameters (the term "dimension" can be used to take an argument of arbitrary dimension), and provides some of the benefits of parametric dimensions, this limited form of parametricity, combined with the restriction that dimension exponents be natural numbers, prevents definition of a polymorphic `sqrt` function:

```
double dimension(v)^1/2 sqrt(double*dimension v)
```

van Delft proposes the following alternative:

```
double*dimension sqrt(double*dimension(return*return) v)
```

Where `return` is the type of the return value. This notation provides an ad hoc mechanism to allow for equating the return type of a method with its parameter types. But this mechanism is really just a workaround for the lack of general support for parametricity, and it prevents expression of parametric dependencies between multiple method parameters, e.g.,

```
<Dimension D>  boolean lessThan(D x, D y)
```

Other published formulations of dimension checking have been over procedural languages. Gehani contrasts the use of derived types in Ada with the use of a facility for unit checking, showing how derived types are an inadequate substitute for a special language feature for units [15]. He also presents an extension of Pascal with support for units [13], in which he allows for polymorphic use of units via a `UNITS(*)` declaration, and a `UNITSOF` operator. However, his discussion often conflates the notion of units and dimensions: he does not include a notion of dimension in his language extension; instead, units of a common dimension are declared to be related via a `counits` operator. As a result, unit conversion can occur along multiple paths, adding significant complexity to an implementation.

Gehani's approach is criticized by House [18], who notes that units are not actually declared in Gehani's proposal. Instead, new units are simply used when needed by a variable declaration. As a result, it is possible to form pathological definitions where the name of a quantity is identical to the name of a (distinct) unit. House also notes that Gehani's system is not statically checkable without severely restricting the use of the `UNITS(*)` operator. House goes on to cite a litany of other criticisms of Gehani's proposal and then proposes his own extension of Pascal supporting dimension checking.

In House's proposal, each dimension is associated with a unique primary unit. Exponents on dimensions are arbitrary rational numbers to allow the definition of methods such as `sqrt`; the syntax for polymorphic dimensions prohibits the solution we have presented, where the input dimension is defined to be the square of a dimension variable. House also restricts exponents on dimensions to be constants.[11] Generic type declarations are supported for records and procedures. However, procedure return types are precluded from being generic dimensions independent of the parameter dimensions. House proposes to solve the problem of scale conversion by allowing for "units with different zeros"; unit conversion involves both an additive and a multiplicative conversion. This approach fails to recognize that scale readings have distinct properties from other quantities and should be modeled as separate entities.

Gehani also discusses the application of units of measurement to databases [14]. In this context, conversion factors can change dynamically (e.g., between monetary units), suggesting that the best way to store data is not with a standard unit per dimension, but instead in the units in which the data was originally entered.

In the context of database manipulation, Gehani recommends the following operators:

- `counits`: specifies that two or more units are of the same dimension
- `remove counits`: removes unit relationships
- `ccc`: changes the conversion constant between two units (useful when dealing with rates of exchange, improving precision of scientific constants, etc.)
- `in`: evaluates an expression in a given unit
- `unitsof`: returns the units of a given value

In our formulation, where dimensions of quantities are denoted explicitly, the `counits` operator is unnecessary, and

[11] In fact, because Pascal syntax does not include an exponentiation operator, compound dimensions are expressed via multiplication and division, which is expressively equivalent to requiring dimension exponents to be constant.

the `remove counits` operator is nonsensical (`remove counits` is apparently motivated by the need to maintain possibly corrupted, persistent data in a database). In a programming language, units either belong to a dimension or they do not. Programmers may introduce and correct erroneous declarations of unit classes in their programs, but dynamically altering the dimension of a quantity is not needed. The ability to support dynamically changing conversion constants between two units could be supported by making the conversion factor in class `Unit` mutable and allowing select classes to support a `ccc` method. The `in` and `unitsof` operator are easily supported as methods in class `Measurement`. The fact that we are able to easily incorporate Gehani's operators into our formulation is a demonstration of the extensibility we have attained by unifying our formulation into a general object-oriented type system.

Manner describes a syntactic extension to Pascal and Ada in which unit declarations are supported [23]. Each dimension is associated with a unique primary unit and all quantities of that dimension are kept in the primary unit. Polymorphism over quantities is not supported. Dreiheller et al. expand upon Manner's syntax to provide a language extension and implementation (via a preprocessor) for Pascal called PHYSCAL [8]. PHYSCAL supports scale factors on units as well as input/output facilities for quantities. PHYSCAL is limited to supporting SI units; the programmer cannot define new units or dimensions. Scales of measurement are not handled by PHYSCAL, nor is polymorphism over program constructs involving dimensions.

Baldwin proposes another extension of Pascal supporting units inspired by Manner [3]: a fixed set of dimensions is added to Pascal as siblings of type `REAL`. Each dimension is associated with a unique primary unit, and quantities of that dimension are always kept in the primary unit. Multiplication and division of dimensioned quantities by dimensionless quantities is not supported despite recognition that such support is needed for correctness.

Hilfinger proposed an encoding of support for dimension checking in Ada in [17]. Hilfinger's approach shares some of the flavor of ours in that he attempts to use existing language abstraction mechanisms to encode dimensions rather than introducing an ad hoc language feature. He relies on the subtyping mechanism of Ada to encode dimensions as subtypes of records of constant integers. Each field in this record supertype corresponds to a primary dimension. Because the fields in this record are fixed, the programmer cannot define new dimensions. Also, quantities of a dimension must be represented in a primary unit. Limited polymorphism is allowed because a programmer can refer to the record supertype of all dimensions when defining methods. However, general parametric polymorphism including dependencies between the dimensions of multiple variables is not supported.

Karr and Loveman discuss mechanically incorporating units into a procedural programming language [19]. Like us, they are concerned not with simply adding unit checking to an existing language, but rather with understanding more general issues in adding unit checking to a class of languages. Karr and Loveman do not include a separate notion of dimension. As a result, detecting commensurability of units becomes a central aspect of their approach. However, there is an upshot to their focus on commensurability: their mechanism for manipulating unit relationships is more general than ours: Rather than defining the magnitudes of all units for a given dimension in terms of a primary unit, each unit may be related to any other unit. Commensurability is determined by a interesting computational trick where equations relating units are represented in logarithmic form, presented as matrices, and put into row echelon form. Cycles and inconsistencies in the unit relationships can be determined by straightforward inspection of the reduced matrix.

Because we allow for dimension types in our approach, commensurability detection is not necessary for static checking. However, intelligent unit conversion at run time would allow for improved precision. For example, when converting feet to inches, it may be advantageous to relate them directly rather than to relate them each to meters and then divide. An interesting direction for future work is to try to embed the row-echelon technique of Karr and Loveman into the definition of method `inUnit` on measurements. Ideally, the equations composing this matrix would be checked statically.

Parametric units are not discussed in Karr and Loveman's proposal. Raising quantities to a (possibly nonconstant) power is allowed, so long as the unit of the exponent is commensurate with 1. If the exponent is nonconstant, the units of the expression are assumed to be 1, breaking soundness of static checking.[12] Scale measurements and input/output of quantities are not addressed by their proposal.

The earliest mention of dimension checking for programming languages that we are aware of occurs in J.C. Cleaveland's discussion of "pouches" [7]. Pouches are described as a general way to allow programmers to organize numeric variables into natural classes or categories. It is shown how a set of pouches can be organized as the base elements of a freely generated abelian group, and therefore as unit annotations. No polymorphism is provided over pouches, but it is mentioned in passing that a notion of "hidden" pouches would be useful when defining arithmetic functions that are naturally polymorphic. Exponents on generated elements of pouches are restricted to be constants. No subtyping is provided over pouches. Pouches are used to describe units rather than dimensions, but no facility is provided for converting between commensurable units.

6.2 Generalized class relationships in other languages

The use of class relationships beyond simple class extension is essential for modeling many real-world ontologies [9]. Many non-statically-typed object-oriented languages, such

[12]Because parametric polymorphism is not supported, this design decision is reasonable as long as all occurrences of these exponent expressions are either accompanied by a static warning or required to be annotated by the programmer. Provided that enough information is kept to determine the actual units of these expressions at run time and to check their use, exponents with nonconstant exponents in this proposal are akin to downcasts in the Java Programming Language.

as Smalltalk, Python, and Self, allow for more flexible relationships. 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. Self, on the other hand, is a prototype-based object-oriented language, 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.

Our notion of metaclass differs from that in languages such as Smalltalk and Python, where each class has a unique metaclass that defines its class members. Metaclasses in these languages are typically used to define customized properties of select classes. In contrast, we use metaclasses to augment traditional static checking of properties that are typically encoded indirectly. Not all classes have a distinct metaclass.

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 MetaGen is more expressive in the sense that we include classes and subclassing relationships. Also, unlike typical prototype-based languages, MetaGen is statically typed.

Because our formulation of metaclasses includes nominal generic types, we can handle certain (conceptually parametric) metaclasses more gracefully than they are handled in dynamically typed languages like Smalltalk. For example, in Smalltalk (pre Smalltalk 80),[13] the upper ontology of classes and metaclasses is as follows (using MetaGen syntax) [29]:

```
ObjectClass Object{...}
ClassDescriptionClass ClassDescription extends Object {...}
MetaclassClass Metaclass  extends ClassDescription {...}
ClassClass class extends ClassDescription {...}
Metaclass ObjectClass extends class {...}
Metaclass classDescriptionClass extends ObjectClass {...}
Metaclass MetaclassClass extends ClassDescriptionClass {...}
Metaclass ClassClass extends ClassDescriptionClass {...}
```

User-defined classes are instances of new metaclasses. For example, class `Dimension` would be represented as:

```
Metaclass DimensionClass extends ObjectClass {...}
DimensionClass Dimension extends Object {...}
```

The metaclasses hold the static members of their instances.

Notice that in Smalltalk we are prevented from modeling the structure we really want over these classes because we do not have generic types. An object-oriented generic type system with nominal subtyping such as in NextGen and MixGen [6, 2, 1] have the important property that each instantiation of a generic type has a distinct run-time representation (this is not the case for a structural type system).[14] We can leverage this fact to encode a metaclass hierarchy much more concisely than can be done in Smalltalk. Instead of writing classes such as `ClassClass`, `ObjectClass`, etc., we can define a metaclass `Class` that is parameterized by its contained class:

```
Class<Class<C>> Class<Class<C> C> {...}
```

Note that the bound on type parameter `C` is itself `Class<C>`. This bound allows us to define instance classes such as

```
Class<Object> Object {...}
Class<Dimension> Dimension {...}
```

So instead of having `ObjectClass` and `DimensionClass`, we have `Class<Object>` and `Class<Dimension>`. If we want to include extra class members in the metaclass of `Dimension` (for instance, if we want to model static members of `Dimension` as members of the metaclass), we can subclass `Class<Dimension>` and define `Dimension` to be a member of the subclass. The important point is that we do not need an explicit `Metaclass` class because we can represent metaclasses as instantiations of a single generic type. Likewise, there is no need for a `ClassDescription` class, significantly simplifying the class hierarchy.

Dynamically typed languages such as Smalltalk, Self, and Python are inappropriate for modeling dimension checking. The reflection facility in the Java Programming Language can also be thought of as providing some of the power of metaclasses. For example, each class is regarded by the reflection facility as an instance of class `Class`. However, use of the reflection facility is not statically checked in the Java Programming Language, effectively placing it in the same category as dynamically typed languages in this respect.

Nominal subtyping is an essential property of our extended language. Many of the instance classes we define may themselves contain no internal structure, but establishing subtype relationships that are independent of structure is central to our approach. A static type inference system is provided for Smalltalk-like metaclasses in *Strongtalk* in [4]. However, this type system is based on structural subtyping with no run-time representation of types. Therefore, it is inadequate for encoding our formulation of dimensions and units, which relies fundamentally on first-class genericity and run-time representation of types.

7. FUTURE DIRECTIONS

In this paper, we introduce a generic static-checking system over dimensions and units. There are several interesting directions to extend this work. For example, our system employs explicit polymorphism, which is strictly more expressive in a nominal generic type system with support for type-dependent operations (consider, for example, a polymorphic method that takes no arguments and returns an instance of a type parameter). Nevertheless, it would be natural to extend this system with limited forms of implicit polymorphism where type instantiations can be inferred.

Another important extension to our system would be to provide support for programmers to define new classes with equational constraints such as those we have built into abelian classes. A natural approach to this problem would be to

[13] Smalltalk 80 includes a more complex hierarchy with software engineering benefits orthogonal to the issues under discussion.

[14] In Java 2 SDK v.1.5, generic types are erased to their upper bounds after type checking. Consequently, type-dependent operations such as casts and `instanceof` tests cannot be performed safely on generic types in that language.

transfer and extend results from constrained type systems such as the HM(X) extension to Hindley-Milner type inference, presented in [26] and implemented in the Chameleon extension of Haskell [25], from a structural type system to a nominally typed object-oriented type system.

We would also like to examine to what degree we can unbox dimensioned quantities in real programs to achieve performance acceptable for high-performance computation.

Finally, formalizing the MetaGen semantics over a core calculus (e.g., an extension of Core MixGen) is an important next step in helping us to better analyze its properties and interactions with other language extensions.

Acknowledgements

We are gratetful to Matthias Felleisen, Sam Tobin-Hochstadt, Cheryl McCosh, Miriam Kadansky, and the anonymous referees for their comments on earlier drafts of this work.

8. REFERENCES

[1] E. Allen, J. Bannet, and R. Cartwright. A First-Class Approach to Genericity. In *Proceedings of the Eighteenth Annual SIGPLAN Conference on Object-Oriented Systems, Languages, and Applications*, Anaheim CA 2003.

[2] E. Allen, R. Cartwright, and B. Stoler. Efficient implementation of run-time generic types for Java. In *IFIP WG2.1 Working Conference on Generic Programming*, 2002.

[3] G. Baldwin. Implementation of Physical Units. SIGPLAN Notices, V. 22(8). August 1987.

[4] G. Bracha, D. Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings of the Eighth Annual SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 215-230, 1993.

[5] W. Brown. Applied Template Metaprogramming in SIUnits: the Library of Unit-Based Computation. In `Proceedings of the Second Workshop on C++ Template Programming`, October 2001. Available at `http://www.oonumerics.org/tmpw01/brown.pdf`.

[6] R. Cartwright and G. Steele. Compatible genericity with run-time types for the Java programming language. In *Proceedings of the Thirteenth Annual SIGPLAN Conference on Object-Oriented Systems, Languages, and Applications*, Vancouver BC 1998.

[7] J.C. Cleaveland. Redundant Specification in Programming Languages Through Pouches. UCLA Technical Report. 1975.

[8] A. Dreiheller, M. Moerschbacher, B. Mohr. PHYSCAL: Programming Pascal with Physical Units. SIGPLAN Notices, V. 21(12), December 1986.

[9] D. Ferrucci, C. Welty. What's in an Instance? RPI Computer Science Technical Report. 1994.

[10] M. Fowler. Analysis Patterns: Reusable Object Models. Addison-Wesley 1997.

[11] D. Isbell, D. Savage. Mars Climate Orbiter Failure Board Releases Report, Numerous NASA Actions Underway in Response. NASA Press Release 99-134. `http://nssdc.gsfc.nasa.gov/planetary/text/mco_pr_19991110.txt`. Nov. 10, 1999.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[13] N. Gehani. Units of Measure as a Data Attribute. Computer Languages, Vol. 2. pp. 93-111. Pergamon Press, 1977. Printed in Great Britain.

[14] N. Gehani. Databases and Units of Measure. IEEE Transactions on Software Engineering. Vol. SE-8, No. 6, November 1982.

[15] N. Gehani. Ada's Derived Types and Units of Measure. Software – Practice and Experience, Vol. 15(6), 555-569. June 1985.

[16] J. Goubault. Inference d'unites physiques en ML. In P. Cointe, C. Queinnec, and B. Serpette, editors, Journees Francophones des Langages Applicatifs, Noirmoutier, p.3-20. INRIA, Collection didactique, 1994.

[17] P. Hilfinger. An Ada Package for Dimensional Analysis. ACM Transactions on Programming Languages and Systems, Vol. 10, No. 2, April 1988. p. 189-203.

[18] R. T. House. A Proposal for an Extended Form of Type Checking of Expressions. The Computer Journal, Vol. 26, No. 4. 1983.

[19] M. Karr, D. B. Loveman III. Incorporation of units into programming languages. Communications of the ACM, 21(5):385-391, May 1978.

[20] A. Kennedy. Dimension Types. In Proceedings of the 5th European Symposium on Programming Languages and Systems. Edinburgh, U.K. 1994.

[21] A. Kennedy. Programming Languages and Dimensions. PhD Thesis. St. Catharine's College. November 1995.

[22] A. Kennedy. Relational Parametricity and Units and Measure. In *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages*, Paris, France. January 1997.

[23] R. Manner. Strong Typing and Physical Units. SIGPLAN Notices V. 21(3), March 1986.

[24] P. Neumann. Risks to the public from the use of computers. ACM Software Engineering Notes, 10(3) July 1985.

[25] M. Sulzmann, The Chameleon website. http://www.comp.nus.edu.sg/ sulzmann/chameleon/

[26] M. Sulzmann, M. Odersky, M. Wehr, Type Inference with Constrained Types. Theory and Practice of Object Systems, 5(1), 1999.

[27] A. van Delft. A Java Extension with Support for Dimensions. Software – Practice and Experience, 29(7), 605-616. 1999.

[28] M. Wand, P. M. O'Keefe. Automatic dimensional inference. In J.-L. Lassez and G. Plotkin, editors, Computational Logic: Essays in Honor of Alan Robinson. MIT Press, 1991.

[29] A. Goldberg, D. Robson. *Smalltalk-80: The Language.* Addison-Wesley, 1989.