Quantities in OWL

Bijan Parsia and Michael Smith

Clark & Parsia LLC, Washington, DC, USA
{bijan,msmith}@clarkparsia.com

Abstract. It is essential to be able to incorporate representation of quantities into models of many domains. Many ontologies attempt to axiomatize a theory of quantities in OWL so that they may incorporate quantities in their ontologies. However, OWL based axiomatizations suffer from a number of problems including the impossibility of representing (and computing) conversions in OWL 1. While there are proposed extensions to OWL that would allow for linear equations (and thus, in principle, handle a large class of conversions), we argue that OWL based axiomatizations are, nevertheless, suboptimal from several perspectives. Instead, we propose a new datatype system for handling quantities and show that it is superior from both a modeling and an implementation point of view.

1 Introduction

It is essential to be able to incorporate *quantities* into models of many domains. The Health Care and Life Science (HCLS) domains are, perhaps, the most intensive area of ontology development and have been for decades. These domains are shot through with a need to represent physical quantities. Many HCLS (and other) ontologies attempt to axiomatize a theory of quantities in OWL[1] so that they may incorporate quantities in their modeling. Axiomatizing in OWL has some advantages:

1. No changes are required to OWL itself or to the toolchain to use such quantity representations.
2. The theory is transparent and inspectable.
3. The theory is extensible and customizable by OWL modelers.

However, there are considerable disadvantages:

1. The theory is non-standard, thus tools have a much harder time being sensitive to quantities (e.g., in order to display measurements using different units). While this can, in part, be overcome by standardizing on a particular ontology, the fact that it is *in* OWL makes it difficult to optimize for, especially in the presence of customizations (intended or inadvertent).

[1] We use OWL primarily to mean OWL 2 and somewhat indistinguishably with OWL 1 and, in all cases, the description logic (DL) variant.

2. The axiomatization of quantities contaminates the axiomatization of the domain. The most obvious issue is how it affects reasoner performance, but it also can interfere with, for example, the modularization of the ontology. (Since quantities are cross-cutting, they can force parts of the ontology to be interdependent, logically speaking, that otherwise would not be).
3. The axiomatization is rather difficult to produce and ensure correctness. (Consider all the problems with three distinct international standards described in [12].) Furthermore, some aspects may not be possible to handle in OWL 1 (unit conversions) or even OWL 2 (unit comparability).
4. Axiomizations of quantities in OWL are typically difficult to use, even from a simple syntactic perspective.

Consider a very simple example which highlights some of these issues (adapted from [10], example 1, and used to illustrate the utility of n-ary data predicates):

Example 1 (Miles and Kilometers). Suppose we are trying to reconcile two geography data sets: one where river lengths are described in kilometers and a second where they are described in miles. We have data on the length of the Yangtze river as follows:

```
ClassAssertion(Yangtze River)
PropertyAssertion(length-in-miles Yangtze "3937.5"^^xsd:decimal)
PropertyAssertion(length-in-kilometers Yangtze "6300"^^xsd:decimal)
```

We would prefer these to be consistent, and we would prefer assertions like

```
ClassAssertion(Yangtze River)
PropertyAssertion(length-in-miles Yangtze "6300"^^xsd:decimal)
PropertyAssertion(length-in-kilometers Yangtze "6300"^^xsd:decimal)
```

to be inconsistent. To get this effect, we need to add a constraint on the instances of River such that the value of `length-in-miles` $= 1.6*$`length-in-kilometers`:

```
SubClassOf(River
   AllValuesFrom(length-in-miles length-in-kilometers
      LinearExpression(Arguments(m k) eq m times(1.6 k))))
```

This “simple” example, while achieving the key functionality of conversion, illustrates wretched usability. While, in the current proposal for linear equations[2], we could name the conversion and thus reuse it, the fact of having to incorporate the unit name into the property leads to an absurd proliferation of properties. Not only do we need distinct properties for each unit, but for each aspect of the object which has a length (consider, `maximum-depth-in-miles` and `maximum-depth-in-kilometers`). (Imagine the chaos if we include derived units!) It is clear that we need a careful design in order to have usable quantities in OWL.

In general, there are two basic aims for an OWL ontology of quantities: to explore the representation of quantities (quantities as domain) and to use a

[2] `http://www.w3.org/2007/OWL/wiki/Data_Range_Extension:_Linear_Equations`

representation of quantities in representations of other domains. The exemplar of the first aim is [5] (but see also [11] which is discuss in some detail in section 2.1), which axiomatizes a very nice theory of physical quantities in first order logic (specifically, in KIF). In this paper, we are concerned with the second aim. We argue that the overall best way to incorporate quantities into OWL is by a basic extension to the language – in particular, quantities should be represented by an extension to the datatype system. This choice, while hardly dominant, is not unheard of; several programming languages (e.g., Frink[3] or Fortress[4]) incorporate quantities into the language itself. The idea of using a language's type system to enforce dimensional consistency recurs often, e.g., in [2–4, 6, 7].

In this paper we focus on physical quantities such as length, mass, and time, though our approach can be applied to other quantities without difficulty.[5]

2 Background

First we introduce some terminology, loosely following [2]. First we have distinct *dimensions*, which are used to type quantities. A dimension is either a *base* dimension or a *derived* dimension. Base dimensions (given a certain system) are primitive and disjoint. Derived dimensions are combinations of base dimensions using multiplication and division operations. `Length` and `Time` are examples of base dimensions in SI. `Mass` × `Temperature` and `Length` ÷ `Time` are examples of derived dimensions.[6] A *unit* is a specially designated quantity, as such associated with a particular dimension, which is defined specifically for use in the description of other quantities. For example, [1] defines the unit `Meter` in the `Length` dimension. Finally, the focus of this paper, a *quantity*, is a unit paired with a magnitude. For example, 3.7 `Seconds` is a quantity.

This conceptualization is constructed such that all and only quantities with the same dimension are comparable. More specifically, for any two units of the same dimension, a scaling relationship exists to map quantities from the first unit to the second. For example,

$$m\ \texttt{lightseconds} = m \times 299792458\ \texttt{meters}$$

[3] `http://futureboy.homeip.net/frinkdocs/`
Frink has inspiring usability. It exemplifies the advantages of building quantities directly into the language.

[4] `http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf`, although dimensions have been removed from the final 1.0 spec. This was to accurately reflect the current implementation, not to express a design decision.

[5] That is, aside from difficulties brought in by the quantity itself. Money is a good example since exchange rates alone are ever fluctuating and quite complex even before one tries to account for inflation. Frink has a interesting support of money which includes working with historical buying power.

[6] There is flexibility in which dimensions you chose as base and which are derived. For example, you could chose speed and either time or length as base and have length or time (respectively) be derived.

is a unit conversion in the `Length` dimension. By converting quantities to the same unit, their magnitudes are directly comparable. Given such relationships among quantities within a dimension, the ability to check the equality of dimensions, the simplest kind of *dimensional analysis*, becomes particularly relevant. Dimensional analysis is an area to which considerable study has been devoted and computationally efficient algorithms have been defined, as in [4].

2.1 Axiomatizations in OWL

While we did not do an extensive survey, a simple perusal of the first page of a Swoogle search[7] for "unit" revealed five distinct axiomatizations of units. There were individuals (`resolutionUnit`, `time-measure-second`), classes (`Unit-Of-Measure`, `Time-Measure`, `measuring-unit`, `speed-unit`), and properties (`has-unit-of-measure`, `Unit`, `angularMeasure`). Some obviously were meant to work together (`mass` (a property) and `kilogram` (a class)).

Clearly this is a horrid mess. Most of the examined ontologies were in RDFS, thus could not, in principle, support dimensional disjointness. Furthermore, since there are no equations included, there is no unit conversion either. Some of the axiomatizations were intentionally trivial, e.g., the range of a property `Unit` was `xsd:string`. In others, however, there are enough separate terms arranged in fairly complex relations to indicate that the authors did want some joy from all that.

This shows that *some* sort of standard way of handling quantities would be very helpful if only to prevent repeated, flawed effort. But if we examine an ontology explicitly proposed to provide standard quantity handling in OWL, UnitDim[11], we find a number of problems.[8] First, we note that it is, necessarily, very heavyweight with 302 classes, 500 individuals, 476 subclass axioms, 880 data property assertions, and so on. It uses nominals, inverses, and functional properties (specifically, it is in $\mathcal{ALCOIF}(D)$). It is also inconsistent (due to, we believe, a datatype error, though our attempted fix did not resolve matters). It is rather daunting to have to import such an ontology that may well be larger and more complex than one's own! There is a wealth of detail including an interesting attempt to model derived dimensions by a series of data properties. It seems reasonable to expect that the modeling would trigger inconsistencies if an individual quantity were made an instance of two dimensionally incompatible quantities. But consider a simple example,

Example 2 (UnitDim definition of `Frequency`*).* Frequency is defined to be a quantity which has two possible units, 1/*second* and *hertz* and as having a certain dimension:

```
SubClassOf(Frequency
    IntersectionOf(
```

[7] `http://swoogle.umbc.edu/index.php?service=search&queryType=search_swd_ontology&searchString=unit&searchStart=1`

[8] We retrieved a copy from `http://www.atoapps.nl/ProjectSite2/data/documents/217618013-20041216155627/UnitDim.owl`.

```
Quantity
UnionOf(
   HasValue(possible_unit reciprocal_second)
   HasValue(possible_unit hertz))
HasValue(SIdimension time_to_the_power_-1_SI_dimension)))
```

The dimension is connected to a dimension vector by a series of (functional) data properties (zero valued assertions removed for space reasons, but are essential in the ontology):

```
ClassAssertion(time_to_the_power_-1_SI_dimension SI_dimension)
PropertyAssertion(time_exponent
   time_to_the_power_-1_SI_dimension "-1"^^xsd:int)
```

We see that quantities can have multiple sorts of units (though it is unclear to us that the right units are distinct; for example, it seems possible that hertz and meters could be consistently equal), but the relationship between these units cannot be modeled effectively. Multiplication and division are represented as objects, thus are not computed. Thus, it seems possible for a reasoner to find that 1 centimeter is equal to 1 meter. If such an expression is not detectably inconsistent, then the unit representation loses much of its value. Indeed, from a practical perspective, it would be better to manually normalize all unit expressions into a directly comparable form, though the naturalness of the modeling would be lost.

Clearly there is a lot of representational redundancy explicit in UnitDim. For example, not only is the dimension vector explicit in the ontology, but the name of the corresponding individual is a long franken-name that *looks* compositional and meaningful, but, to the tools, it is not. Furthermore, the effective representation (the vector) is a rather strange representation from the user point of view. We typically find such representations as the internal data structure of a program, not as a notation for people.

We have examined two general strategies for representing quantities in OWL: profilerating properties and building structured objects. Unfortunately, each has a key strength that the other lacks. Proliferating properties allows us to use feasible equation systems for unit conversion and, to some extent, in dimenional analysis of equations. (For example, it would be nice if we could detect that a class with a property constrained by an equation was a subclass of a class that has a property constrained to be $< 2cm$.) However, the modeling is in principle unworkable (since we would have to generate the cross product of our properites and the units we want to use with them). On the other hand, we can at least imagine a structured object representation (such as in UnitDim) that did not have these issues (though would still be unwieldy at best). But these depend on *chains* of properties to relate objects to measurements. That is, instead of writing that a person's height is *2 meters* (where height is represented as a data property) we have to make height an object property with a range `Length` which has a property like `hasUnit` and another `measuredValue`. But now there is no hope of using these values in equation systems for the forseeable future, since the known implementable equation systems require that all inputs to the equation be taken from direct data properties. This is a serious compromise of the usability of OWL based quantity systems.

What we would like from quantity support in OWL is the ability to work simply and naturally with quantities described using different unit systems wherein comparison, range restrictions, etc. are unit insensitive (i.e., the system converts behind the scenes instead of required manual casting), dimensional analysis is performed (so comparison of lengths with times causes inconsistency), and we have the ability to easily create arbitrary derived units whether directly or implicitly via some equation over data properties. For example, we would like (given that `height` is functional):

```
PropertyAssertion(height sheevah "2 meters"^^owl:quantity)
PropertyAssertion(height sheevah "200cm"^^owl:quantity)
```

to be consistent while:

```
PropertyAssertion(height sheevah "2m"^^owl:quantity)
PropertyAssertion(height sheevah "2centimeters"^^owl:quantity)
```

to be inconsistent. Similarly,

```
EquivalentClasses(Tall
   SomeValuesFrom(height
      DatatypeRestriction(xsd:integer minExclusive "6 feet"^^owl:quantity)))
```

Should have `sheevah` as an inferred instance. We should get disjoint classes if they have properties which are dimensionally inconsistent, subsumptions when they are dimensional consistent and the ranges are appropriately related, and so on. It should be very hard to mess things up and the tools should be able to give back a lot of information when we do mess things up.

3 Datatype approaches

All these considerations point to adding quantities as a new sort of datatype to OWL. Datatypes represent domains for which there is a worked out theory such that it makes sense for the system to be sensitive to it. For example, even if it were possible, it is very undesirable to require people to axiomatize a theory of the integers in order to model with integers. Similarly, it is pretty clear that quantities have a worked out theory that would benefit from special syntactic and semantic support. A new datatype would allow us to introduce very nice syntax and to enforce all our the semantic desiderata. The main price is that, without rather more work to define a quantity description system, our set of base units and unit names are fixed.

4 Implementation

In this section we present two alternative approaches to adding support for quantities, as a concrete domain, to existing OWL applications. The first approach is implemented by translation to plain OWL 2 thus is compatible with all OWL 2 reasoners. The second approach describes modification to an existing OWL 2 reasoner to directly support quantities.

We introduce two elements of syntax as extensions to the OWL structural syntax defined in [9]. This syntactic extension is introduced only for purposes of exposition and demonstration, final syntax choices have not been made. First, for the remainder of the document we will use `owl:quantity` as the datatype of quantities being represented as literals. We adopt the lexical form defined for the Unified Code for Units of Measure (UCUM)[9][12]. Second, we introduce `quantity:dimension` as a facet on the quantity datatype used to constrain the value space to a single dimension. Facet values are unit atoms, also defined by UCUM. Examples in the next section illustrate the intended use of this syntax.

4.1 External

The translation to plain OWL 2 requires two maps. $\Phi(\cdot)$ is a map from dimension to a class. $\Omega(\cdot)$ is a map from data property to object property. Each map begins empty. During algorithm execution, if $\Phi(\cdot)$ is undefined for a dimension, it is defined with a mapping to a fresh class. Similarly, if $\Omega(\cdot)$ is undefined for a data property, it is defined with a mapping to a fresh object property. We present the algorithm in two parts. First we present the translation of ABox facts, then we describe the translation of TBox concept descriptions using data ranges. For each, an illustrative example is included. For implementation we must choose a canonical set of base units to which all quantities can be normalized. We use the notation $\mu(q)$ to refer to the magnitude of quantity q, when normalized to the base units of our implementation.

The fact translation part of the algorithm begins by iterating over all data property assertions in which the literal is typed as an `owl:quantity`. For each such assertion, we'll refer to the subject individual as s, data property as p, quantity literal as q, and the dimension of the quantity as $\delta(q)$. To complete the translation, the axiom is retracted from the ontology and three facts are added. The first is a class assertion between a fresh individual i and class $\Phi(\delta(q))$. The second is an object property assertion from s to i using the object property $\Omega(p)$. The third is a data property assertion relating i and $\mu(q)$, using the special functional data property `quantity:magnitude`. The following example demonstrates the translation of a single quantity data property assertion.

Example 3 (Converting a Quantity Fact).
The fact that follows:

```
PropertyAssertion( a:height a:JohnDoe "6.0 ft"^^owl:quantity )
```

is replaced with the following set of axioms, where $\Omega(\mathtt{a:height}) = \mathtt{mint:height}$ and $\Phi(\mathtt{ft}) = \mathtt{mint:DimLength}$:

[9] `http://www.regenstrief.org/medinformatics/ucum`

```
PropertyAssertion( mint:height a:JohnDoe _:d1 )
ClassAssertion( mint:DimLength _:d1 )
PropertyAssertion( quantity:magnitude _:d1 "1.8288"^^owl:real )
```

Informally, the quantity dimension is captured using an object representation and the magnitude is mapped onto the real number line after conversion into the base unit (in this example, meters is the base unit for the length dimension).

Quantities may appear in concept descriptions which include value restrictions on datatype properties. We proceed by considering concept descriptions which include a data range defined as a datatype restriction on `owl:units`. For each such description, we'll refer to the data property as p. The concept description is modified as follows. If the `quantity:dimension` facet is present, with value d, it is removed and a restriction is added constraining fillers of the object property $\Omega(p)$ to class $\Phi(d)$. Each range constraining facet present (e.g., `xsd:minInclusive`) is removed and a restriction is added constraining fillers of the object property $\Omega(p)$ to individuals satisfying a corresponding range facet on the `quantity:magnitude` data property, with the facet values base unit normalized. The following example demonstrates translation of a single concept description. We have built a prototype implementation in XSLT.

Example 4 (Converting a Quantity Data Range).
The concept description that follows:

```
AllValuesFrom( a:height
   DatatypeRestriction( owl:quantity quantity:dimension "m"
      xsd:minInclusive "1.65m"^^owl:quantity
      xsd:maxInclusive "1.8m"^^owl:quantity ))
```

is replaced with the following concept expression, reusing $\Omega(\cdot)$ and $\Phi(\cdot)$ definitions from Example 3:

```
AllValuesFrom( mint:height
   IntersectionOf ( mint:DimLength
      SomeValuesFrom( quantity:magnitude
         xsd:minInclusive "1.65"^^owl:real
         xsd:maxInclusive "1.8"^^owl:real ) )
```

The translation algorithm for other quantity data ranges is straightforward.

4.2 Native

We describe the direct implementation in an existing OWL DL reasoner more succinctly. We begin by assuming a datatype implementation similar to the approach in [8]. Implementation thus requires extending the datatype map to include the datatype `owl:quantity`. The value space for this datatype can be represented internally as a two-tuple consisting of a dimension (a discrete infinite space) and base unit normalized magnitude (the real number line). The reasoner thus must include facilities for dimensional analysis, perhaps based on [4], and base unit normalization. Given these facilities, satisfying the requirements of a datatype handler for `owl:quantity` is straightforward as one can proceed by extending the datatype handler for `owl:real` defined in [8] to included dimensional analysis in containment and equality considerations.

5 Syntax and Other Usability Considerations

We stress that, thus far, we have focused entirely on supporting *modeling* with quantities and have deliberately neglected acquisition and presentation issues. We strongly believe that range axioms and restrictions should *not* be used to constrain the form of an input. That is, we do not think that it makes sense to require that a property's range is *some unit*, such as centimeters, rather than that it is of *some dimension*, such as length. For convenience, it is reasonable to allow a range restriction to centimeters to be sugar for a restriction to length (after all, we want to be able to say meters per second squared, not length per time squared though the latter should be allowed and be presented upon demand), but this should not force an inconsistency if an assertion uses meters instead. A length is a length regardless of the units used to describe it.

In other words, while we always *give* lengths in terms of some unit, we should not thereby force units into the domain. What we describe (via units) are quantities and the same quantity can be described using various units. In this sense, units are much closer to the logical vocabulary of the language and we should expect the reasoner to be *appropriately* indifferent to the choice of unit (just as it is indifferent between using an existential or a min cardinality of 1).

However, we also believe that input constraints, sensible presentation , and helpful queries are important. We advocate tools being very flexible on this matter. For example, it is not difficult to display constants under different comprehensive unit systems. It's also not difficult to display constants using the appropriate scaling prefix (e.g., centi-, kilo-) for various ranges of numbers. All this should be configurable. We are designing a set of annotations and configuration that allow ontology authors to indicate this sort of preference in a standard way. In this manner we hope to pre-empt the overloading of modeling axioms with such information.

6 Conclusion

The representation of quantities, especially those with widespread unit systems, should be standardized. Aside from the considerable interoperability issues, it relieves an unnecessary cognitive load from users: No one should have to decide how to handle meters in their ontology. No one should have to master large amounts of detailed knowledge of OWL in order to say that a field is 100 meters long. Saying such should be syntactically easy and semantically correct.

We believe there is no reasonable substitute for building quantity support into OWL. While a structured object approach (whether chained or branchy) can handle a large subset of interesting cases (while remaining extensible) it fails on ease of use and is not compatible with forseeable equation support in OWL. To address these issues, we propose an extension of OWL in the form of a new concrete domain for quantities. We show how to reduce that support to OWL (with datatype ranges on reals), thus providing a reasoner independent implementation for datatypes involving units. We observe that native reasoner support is very straightforward. Finally, our approach is extendable to supporting unit sensitive equations in OWL.

The main part lacking from our proposal, as it stands, is extensibility. The translation approach shows that extensibility is not too difficult to support. Of course, derived units are already in, but we could easily allow new base units for existing dimensions. Furthermore, as long as new dimensions are relevantly similar to existing ones, adding additional dimensions is not difficult either (though there could be some performance impact). For more unusual dimensions, we think a revision to implementations is appropriate. Reasoners could have a plug in architecture for such, if they so desired.

In any case, we think that the utility of just the standard physical quantities is worth the candle. Hypothetical extensibility should not block actual use.

References

1. NIST Special Publication 330: The International System of Units (SI). 2008.
2. Eric Allen, David Chase, Victor Luchangco, et al. Object-oriented units of measurement. *SIGPLAN Not.*, 39(10):384–403, 2004.
3. Narain H. Gehani. Units of measure as a data attribute. *Comput. Lang.*, 2(3):93–111, 1977.
4. Jr. Gordon S. Novak. Conversion of units of measurement. *IEEE Trans. Softw. Eng.*, 21(8):651–661, 1995.
5. Thomas R. Gruber and Gregory R. Olsen. An ontology for engineering mathematics. In *KR*, pages 258–269, 1994.
6. Michael Karr and III David B. Loveman. Incorporation of units into programming languages. *Commun. ACM*, 21(5):385–391, 1978.
7. Andrew Kennedy. Dimension types. In Donald Sannella, editor, *ESOP*, volume 788 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 1994.
8. Boris Motik and Ian Horrocks. OWL datatypes: Design and implementation. In *The 7th International Semantic Web Conference (ISWC2008)*, 2008.
9. Boris Motik, Peter F. Patel-Schneider, and Ian Horrocks. OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax. 2008.
10. Jeff Z. Pan and Ian Horrocks. Web ontology reasoning with datatype groups. In D. Fensel, K. Sycara, and J. Mylopoulos, editors, *International Semantic Web Conference*, volume 2870 of *Lect. Notes in Comp. Sci.*, pages 47–63. Springer, 2003.
11. Hajo Rijgersberg and Jan Top. Unitdim: an ontology of physical units and quantities. 2004.
12. G Schadow, C J McDonald, J G Suico, U Fohring, and T Tolxdorff. Units of measure in clinical information systems. *J Am Med Inform Assoc*, 6(2):151–62, 1999.