Facets, Tiers and Gems: Ontology Patterns for Hypernormalisation

Phillip Lord [1]*and Robert Stevens[2]

[1]School of Computing Science,
Newcastle University,
Newcastle-upon-Tyne
[2]School of Computer Science,
University of Manchester,
Manchester

ABSTRACT

There are many methodologies and techniques for easing the task of ontology building. Here we describe the intersection of two of these: ontology normalisation and fully programmatic ontology development. The first of these describes a standardized organisation for an ontology, with singly inherited *self-standing* entities, and a number of small taxonomies of *refining* entities. The former are described and defined in terms of the latter and used to manage the polyhierarchy of the self-standing entities. Fully programmatic development is a technique where an ontology is developed using a domain-specific language within a programming language, meaning that as well defining ontological entities, it is possible to add arbitrary patterns or new syntax within the same environment. We describe how new patterns can be used to enable a new style of ontology development that we call *hypernormalisation*.

1 INTRODUCTION

Building ontologies is a difficult and time-consuming business for a number of reasons: from an abstract point-of-view knowledge about the domain can be difficult to gather, to understand and to represent ontologically; more, immediately, ontologies, especially those with a complex representation, can be taxing to describe and define consistently, to update, expand or change when that representation needs to change.

There have been numerous attempts to simplify and clarify this process including: the development of methodologies such as OntoClean that defines a set of meta-properties that can inform ontological modelling (Guarino and Welty, 2002); upper ontologies such as DOLCE or BFO (Grenon *et al.*, 2004) that provide a pre-made upper classification.

Another approach that can leverage both of these techniques is ontology normalisation (Rector, 2002). Originally intended as a mechanism for "untangling" existing hierarchies or classifications being reused as the basis for an ontology, it also has significant use as a pattern for building ontologies *de novo*.

Broadly, a normalised ontology is defined using a skeleton that is a strict tree (i.e. not a acyclic graph) of concepts differentiated using an inheritance (i.e. not a partonomy) relationship. These are further split into: a set of *self-standing entities* in which children are disjoint from each other, but do not cover the parent, and *partitioning* or *refining* concepts that form closed, covering and disjoint hierarchies. Building an ontology in this way, allows the ontology developer to exploit the reasoner to build a polyhierarchy by using classes that define the self-standing entity in terms of the refining partitions. Polyhierarchies are difficult to build manually, as human ontology developers, no matter how good their domain knowledge, find it hard to ensure all possible parents of an entity are taken into account. The normalisation approach uses defined classes and reasoning to remove this chore. Creating the tree of self-standing entities still, however, remains as a task for the developer. The normalisation approach can significantly increase the robustness and reduce the work of manual maintenance (Wroe *et al.*, 2003). In this latter form, ontological normalisation has been widely, if implicitly, used.

While the term "ontology normalisation" has been borrowed, somewhat metaphorically, from database engineering, the process of building ontologies using a set of standard design patterns has a rather more direct relationship to the software engineering equivalent. By reusing a standard set of patterns, it is possible to build an ontology both rapidly, and consistently. This has manifested itself in a number of different ways, with a number of different tools, such as TermGenie (Dietze *et al.*, 2014), or Populus (Jupp *et al.*, 2011) which can generate ontologies according to a pattern.

We have previously described a fully programmatic methodology for ontology development (Lord, 2013), using the Tawny-OWL environment. This is built around the programming language Clojure and enables the ontology to take advantage of all the features of a programming language and its environment, including unit testing (Warrender and Lord, 2015), build, evaluation and, of course, pattern-driven development by simple use of functions (Warrender and Lord, 2013). With respect to patterns, this environment has several advantages. First, and unlike tools such as Populous and OPPL (Egana Aranguren *et al.*, 2009), patterns are developed in the same environment and syntax as simple ontology concepts; it is, therefore, as easy to define a pattern as it is to define a class. Second, being based on Clojure, a language which is homoiconic and has very little syntax of its own, it is possible to build arbitrary syntactic constructions to represent patterns in a way that is both convenient and attractive to the developer.

In this paper, we describe an extension of the normalisation technique that we call *hypernormalisation*. This technique is typified by the (near or complete) absence of asserted hierarchy among the self-standing entities. We describe how this allows construction of an exemplar ontology of amino-acids (Stevens and Lord, 2012). We then move on to describe recent developments

*To whom correspondence should be addressed: phillip.lord@newcastle.ac.uk

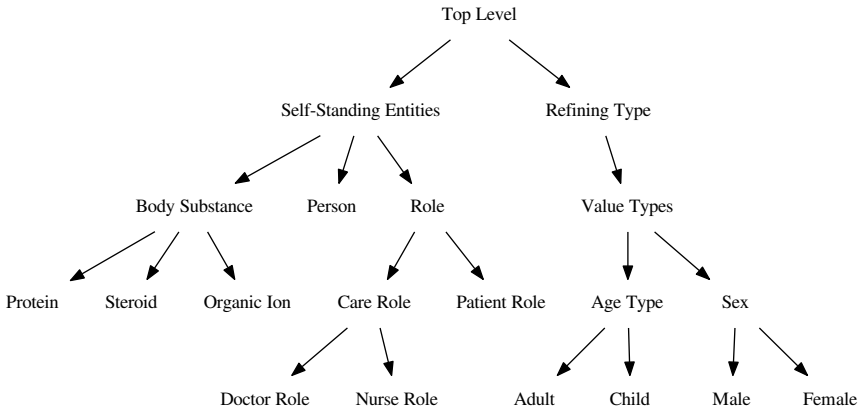

Fig. 1. A normalised ontology slightly modified from Rector (2002). The graph does not necessarily reflect subsumption, see text for details.

in the Tawny-OWL environment, including the definition of two new design patterns, the tier and the facet, and one syntactic abstraction, the gem, can be used to enable hypernormalised ontology development. Finally, we discuss the application of this approach to other ontologies.

2 HYPERNORMALISATION AND AMINO ACIDS

Normalisation is a methodology that aims to disentangle an ontological structure, in the process managing its maintainability, utility and expressivity of the ontology generated. To achieve this, the ontology is split into two main hierarchies: self-standing entities and refining types, see Figure 2 for an example. The self-standing hierarchy contains entities with a central hierarchy or *skeleton*. In this part of the ontology, we would expect that hierarchy contains levels that are not-exhaustive – that is the children do not cover the parents, and parents are not closed to new children. This is contrasted by the refining hierarchy that consists of classes that are exhaustive; in many cases, children will be non-overlapping and, therefore, disjoint. This is not to say that the refining types hierarchy are necessarily complete: in Figure 2, for example, the representation of `Sex` is too simple for many medical uses, but might be sufficient for a customer relations system. In general, the self-standing entities will be defined in terms of the refining types, while polyhierarchical relationships between the self-standing entities will be determined through use of a reasoner.

This form of ontology development is quite different from an upper ontology and agnostic to the choice of upper ontology or none. While Rector (2002) suggests only that self-standing entities and refining types should be "made clear by some mechanism"; in OWL, it could be an upper ontological term, or an annotation.

We next introduce the amino-acid, used here as an exemplar, which defines the biological amino-acids in terms of the physiochemical properties most relevant to their biological role. It is a structurally interesting ontology because it is normalised, with a clear and clean separation between the self-standing entities and the five refining concepts. It is rather more than this, though; the self-standing entities are split into only three sets: the amino-acids themselves (e.g. Alanine); a (very large) set of defined classes describing the refined types of amino-acid (e.g. Small Neutral Amino Acid); and, finally, the single class Amino Acid. Or, stated alternatively, it contains no skeleton hierarchy at all, and all relationships between the self-standing classes are arrived at through reasoning. This is particularly relevant for the amino acid ontology as it contains over 500 defined classes, with subsumption relationships to the amino acids and between themselves. Maintaining this form of ontology by hand would be impractical.

We call this style of ontology development *hypernormalised*. We believe that it is a natural extension of normalisation. Rector notes, for example, that the choice of aspect to form the skeleton is "to some degree arbitrary", but that they should be rigid (after OntoClean (Guarino and Welty, 2002)) and pragmatically stable (i.e. unlikely to change during the evolution of the ontology). Both of these are, however, true for all the refining concepts in the amino-acids. In short, not only is the choice of skeleton arbitrary it is actually unnecessary and brings no further utility to the ontology than that which can be achieved by use of reasoning.

We note that the distinction between normalisation and hypernormalisation is not absolute, but one of degree; we are simply describing the tendency toward an ontology with an flat asserted hierarchy.

Having introduced the notions of a hypernormalised ontology, we next consider a set of new patterns in Tawny-OWL that enable this style of ontology development.

3 PATTERNISING AND TAWNY-OWL

The Tawny-OWL environment (Lord, 2013) and its ability to support patterns (Warrender and Lord, 2013) has been described elsewhere in detail; here, we provide a quick overview, so that the rest of the paper is clear. Tawny-OWL is implemented as a DSL (domain-specific language) in Clojure, which is a Lisp-like language implemented in Java, and running on the Java virtual machine.

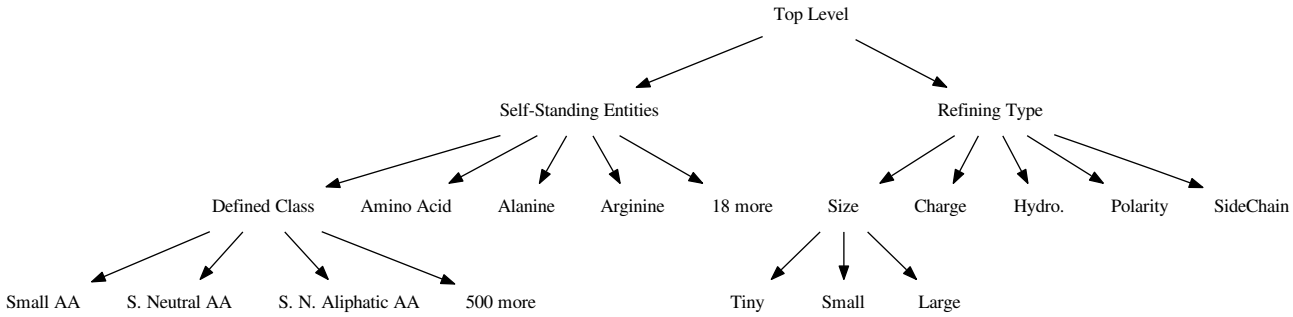

Fig. 2. A hypernormalised ontology representing the amino-acids using the same terminology as Figure 2. Some labels have been abbreviated.

Tawny-OWL itself wraps the OWL API (Horridge and Bechhofer, 2011); this is the same library that underpins Protg, and from it, Tawny-OWL gains much of its functionality. Simple sections of the ontology can be generated using a syntax based on a "lispified" version of Manchester OWL Notation; for example, the following code:

```
(defclass A
  :super B)
```

This declares a new class `A` that has the pre-existing class `B` as a superclass [1] which in Manchester OWL notation would be expressed as:

```
Class: o:A
   SubClassOf:
        o:B
```

This code is entirely valid Clojure and can be evaluated in any Clojure environment, such as CIDER/Emacs or Cursive/IntelliJ. It is also possible to define new patterns: for example the following pattern definition:

```
(defn some-only [property & clazzes]
  (list (some property clazzes)
        (only property (or clazzes))))
```

defines the `some-only` pattern which generates a set of existential restrictions and one universal with the union of the existential fillers as its filler, which implements the ontological *closure* pattern. This is a function definition in Clojure terms: `defn` introduces the function, `property & clazzes` is the argument list, `some`, `only` and `or` are functions provided by Tawny-OWL and `list` returns, prosaically, a list[2]. Critically, it is possible to define this pattern in the same environment, or side-by-side in the same file as a simple class definition; with Tawny-OWL it is as easy to define a class, as to define and use a new pattern. Ontologies such as the Karyotype ontology make extensive use of this facility moving freely between ontology and pattern definitions, as well as literal data structures, utility functions and unit tests (Warrender and Lord, 2013).

Tawny-OWL is now a mature and used software product; the first alpha release of Tawny-OWL was in Nov 2012, first full release, Nov 2013, followed by four point releases to 2016. This paper describes mostly the upcoming v2.0 release, although some of the features described were available in earlier versions.

4 THE VALUE PARTITION

A common pattern for building a normalised ontology is called the *value partition*. This pattern (Rector, 2005) addresses the problem of the ontological modelling of a continuous range. For example, in modelling the amino-acids, we can consider the concept of `Size`; this could be described directly using the molecular weight of the amino-acid. However, for the purpose of the amino-acids, it is both easy and general practice to split size into three categories: tiny, small and large. In Tawny-OWL, this can be achieved straightforwardly using the `defpartition` function[3].

```
(defpartition Size
  [Tiny Small Large]
  :domain AminoAcid
  :super PhysioChemicalProperty)
```

Axiomatically, this expands into: a class `Size`; three subclasses, `Tiny`, `Small` and `Large`; and, a property `hasSize`. The

[1] See Lord (2014) an explanation of why `:super` is used rather than `:subclass`.

[2] The function shown here is a slightly simplified version of one provided in Tawny-OWL.

[3] For those with knowledge of Lisp, this is actually a macro; the main implementation is in the `value-partition` function. Tawny-OWL provides support for implementing syntactic macros whose function is simply to allow the use of bare symbols. For those without knowledge of Lisp, the distinction is not important!

property is functional, has range of `Size` and domain of `AminoAcid`. Expanded, this would be expressed as follows[4]:

```
Class: o:Large
    SubClassOf:
        o:Size

Class: o:Size
    EquivalentTo:
        o:Large or o:Small or o:Tiny

    SubClassOf:
        o:PhysioChemicalProperty

Class: o:Small
    SubClassOf:
        o:Size

Class: o:Tiny
    SubClassOf:
        o:Size

DisjointClasses:
    o:Large,o:Small,o:Tiny
```

The subclasses are disjoint and cover the parent. Following the terminology from Rector (2002), the value partition is useful for defining partitioning or refining concepts.

5 THE TIER

The value partition is a pattern aimed at a specific purpose – segmenting a continuous range. In practice, though, we have found that the axiomatization of this pattern is more generally useful. For example, considering the amino-acid ontology, it is natural to model the chemistry of the side-chain as such:

```
(defpartition SideChainStructure
  [Aromatic Aliphatic]
  :domain AminoAcid
  :super PhysicoChemicalProperty)
```

While this is intuitive, ontologically, `SideChainStructure` is actually of a very different form from `Size`, as it does not reflect a spectrum. Either the side-chain contains a benzene ring, making it aromatic, or it does not. This form of partition was also noted in Rector (2002) which includes the classes `Male` and `Female` which is not a spectrum, at least in this simplified representation. We introduce here, therefore, the more general notion of the *tier*: a small set of concepts in a one-deep hierarchy. The tier function supports a range of options:

```
(deftier Charge
   [Positive Neutral Negative]
   :domain AminoAcid
   :super PhysioChemicalProperty
   :suffix true)
```

[4] Tawny-OWL also adds annotations which have been elided

The use of `:suffix true` causes a simple change to the naming of the entities: `Positive` will become `PositiveCharge` which would be expanded as follows:

```
Class: o:PositiveCharge
    SubClassOf:
        o:Charge
```

Other names are modified equivalently. By default, this will manifest both when referring to the class in the Tawny-OWL environment, in the IRI of the concept when serialized as OWL, and in the value of an annotation on the concepts[5]. In addition to naming, it is also possible to optionalise: whether or not the subclasses are disjoint, covering, whether the property is functional or whether it is created at all.

The tier is a more general pattern than the value-partition; in fact, in the current version of Tawny-OWL, the latter is defined in terms of the former.

6 THE FACET

Both the value partition and tier introduce a new object property named after the tier, and with a range limited to the classes defined within the tier. The converse is also true; where we use one of the tier classes, such as `PositiveCharge` it is most likely that we wish to use it with the `hasCharge` property defined as part. Taken together, we describe the combination of classes and a property as a *facet*. Facets are a well known technique, first proposed in a library classification (the Colon Classification (Ranganathan, 1933), named after the use of ":" as a separator). They are now common-place as seen with facetted browsers used by many websites for navigation of complex product catalogues.

Tawny-OWL provides explicit support for facets, allowing the association of a property and a set of classes, as demonstrated by the following code:

```
(as-facet
 hasCharge

 Positive Neutral Negative)
```

The practical implication of this is that we can now use the `facet` function to return an existential restriction providing just a class. We can express this programmatically; for example, we might use the `assert` function provided by Clojure's unit test framework.

```
(assert
 (= (some hasCharge Positive)
    (facet Positive)))
```

By itself, this ability is only slightly more succinct. However, when used with multiple facetted classes, the advantages become considerably clearer, as can be shown by the following assertion.

```
(assert
 (= (list (some hasCharge
```

[5] The duplication between the annotation and the IRI fragment is there because IRI schemes such as numeric style OBO IDs; annotations have been elided for brevity

```
            Neutral)
        (some hasHydrophobicity
            Hydrophobic)
        (some hasPolarity
            NonPolar)
        (some hasSideChainStructure
            Aliphatic)
        (some hasSize
            Tiny))
    (facet Neutral Hydrophobic NonPolar
        Aliphatic Tiny)))
```

In addition to succinctness, this pattern also reduces the risk of errors; a class such as `Tiny` will always be used with its correct property. Without the use of facets, the ontology developer must achieve this by hand. It would also be possible to detect the error using reasoning, although this will only succeed if appropriate range and disjoint restrictions are in the ontology. The `defpartition` and `deftier` functions, of course, both add these range and disjoint restrictions and declare their classes as facets of their properties.

7 THE GEM

Finally, we define the *gem* that provides a syntactic abstraction for a class composed entirely or mainly from facets. Following the terminology from Rector (2002), this abstraction would be useful mostly for self-standing concepts. For example, we could define the amino acid alanine using the following `defgem` statement.

```
(defgem Alanine
  :comment "An amino acid with a single
methyl group as a side-chain."
  :facet Neutral Hydrophobic NonPolar
  Aliphatic Tiny)
```

The other amino-acids can be likewise defined as a series of gems. In fact, the amino acids are so regular, all having the same five facets, that we use a further syntactic abstract specific to the amino-acid ontology – a form of pattern that we describe as *localized* (Warrender, 2015). The gem represents generalised syntax useful for developing any ontology.

8 ON ANNOTATION

We have previously discussed the relationship between a design methodology such as normalisation and the use of an upper ontology. The Tawny-OWL patterns described here are all orthogonal and agnostic to the choice of an upper ontology or to none. They do not place their entities in any particular part of the class hierarchy nor define classes outside of those required for the domain ontology, although they could be easily extended to do so should the ontology developer require.

However, we agree with Rector (2002) that the use of patterns should "made clear" and be explicit within the ontology. For this reason, all of the patterns described here also make use of annotations, using annotation properties defined using its own internal annotation ontology. For example, all entities generated as a result of a pattern such as `deftier` are explicitly annotated as such. This means that the use of these patterns is (informally) explicit in the OWL serialization. Tawny-OWL actually uses these annotations internally, for example, to enable the `facet` functionality by providing a relationship between the classes and the appropriate object property. This is a strictly an implementation detail and could have been achieved without annotations; however, we believe that it shows the value of having this knowledge explicit in OWL.

9 DISCUSSION

In this paper, we describe how we have used Tawny-OWL to provide higher-level patterns which can be applied to ontology development. The patterns provide both functionality and syntactic abstraction over the underlying OWL implementation. In the process, they enable the easy and accurate construction of ontologies.

More specifically, we demonstrate two new patterns: the tier and the facet. The tier is an extension of the existing value partition pattern and can be used for the generation of many small hierarchies that can be used as refining properties. The facet borrows from the library sciences notion of a facetted classification, and is used to associate a set of classes with a specific set of values. This form of classification is very common in the web; the majority of web stores, for example, offer facetted browsing, often with the facets changing for different subsections of the catalogue.

Taken together, these two patterns enable a new form of ontology development, hypernormalisation, which is an extreme form of normalisation. In this form of normalisation, we do away with the creation of a tree of self-standing entities and instead rely on the reasoner to build all the hierarchy. As well as making the ontologist's task easier, it makes the characteristic that would have been used to create the tree of self-standing entities explicit in the form of a refining characteristic. Here, we have described the application of this methodology to the exemplar amino-acid ontology. Of course, it is dangerous to extrapolate to generality from an exemplar, but we have also started to apply hypernormalisation to ontologies of other, more real, domains including clouds (in the meterological sense), cell lines and a reworking of the Gene Ontology. The tier has been made generic; it does not require, for example, that all refining types are closed (i.e. all possibilities are known in advance) nor disjoint.

Clearly, not all forms of ontology will naturally be represented in a hypernormalised form. For example, the Karyotype ontology (Warrender and Lord, 2013) is far from this form; here, we define the self-standing concepts and then use reasoning over a set of defined classes which effectively operate as facets (Warrender and Lord, 2015). However, the popularity of the facetted browsers shows that is possible to use this form of classification in many areas. We believe that the introduction of the concept of hypernormalisation and the implementation of it in Tawny-OWL could have significant implications for the future development of ontologies.

REFERENCES

Dietze, H., Berardini, T. Z., Foulger, R. E., Hill, D. P., Lomax, J., OsumiSutherland, D., Roncaglia, P., and Mungall, C. J. (2014). Termgenie - a web application for pattern-based ontology class generation. *Journal of Biomedical Semantics*, **5**(1), 48.

Egana Aranguren, M., Stevens, R., and Antezana, E. (2009). Transforming the axiomisation of ontologies: The ontology pre-processor language. *Nature Precedings*.

Grenon, P., Smith, B., and Goldberg, L. (2004). Biodynamic ontology: applying BFO in the biomedical domain. *Stud Health Technol Inform*, **102**, 20–38.

Guarino, N. and Welty, C. (2002). Evaluating ontological decisions with ontoclean. *Commun. ACM*, **45**(2), 61–65.

Horridge, M. and Bechhofer, S. (2011). The OWL API: A Java API for OWL Ontologies. *Semantic Web Journal*, **2**.

Jupp, S., Horridge, M., Iannone, L., Klein, J., Owen, S., Schanstra, J., Wolstencroft, K., and Stevens, R. (2011). Populous: a tool for building owl ontologies from templates. *BMC Bioinformatics*, **13**(Suppl 1), S5.

Lord, P. (2013). The Semantic Web takes Wing: Programming Ontologies with Tawny-OWL. *OWLED 2013*.

Lord, P. (2014). Manchester syntax is a bit backward. `http://www.russet.org.uk/blog/2985`.

Ranganathan, S. (1933). *Colon Classification*.

Rector, A. (2005). Representing specified values in owl: "value partitions" and "value sets". W3C Working Group Note.

Rector, A. L. (2002). Normalisation of ontology implementations: Towards modularity, re-use, and maintainability. *Proceedings Workshop on Ontologies for Multiagent Systems (OMAS) in conjunction with European Knowledge Acquisition Workshops*. Siguenza, Spain.

Stevens, R. and Lord, P. (2012). Semantic publishing of knowledge about amino acids. `http://ceur-ws.org/Vol-903/paper-06.pdf`.

Warrender, J. (2015). *The Consistent Representation of Scientific Knowledge: Investigations into the Ontology of Karyotypes and Mitochondria.* Ph.D. thesis, School of Computing Science, Newcastle University.

Warrender, J. and Lord, P. (2013). A pattern-driven approach to biomedical ontology engineering. *SWAT4LS 2013*.

Warrender, J. D. and Lord, P. (2013). The Karyotype Ontology: a computational representation for human cytogenetic patterns. *Bio-Ontologies 2013*.

Warrender, J. D. and Lord, P. (2015). How, What and Why to test an ontology.

Wroe, C., Stevens, R., Goble, C., and Ashburner, M. (2003). A methodology to migrate the gene ontology to a description logic environment using daml+oil. Pacific Symposium on Biocomputing.