Chapter 13

Ontologies and Knowledge-Based Systems

> *The most serious problems standing in the way of developing an adequate theory of computation are as much ontological as they are semantical. It is not that the semantic problems go away; they remain as challenging as ever. It is just that they are joined – on center stage, as it were – by even more demanding problems of ontology.*
>
> – Smith [1996, p. 14]

How do you go about representing knowledge about a world so it is easy to acquire, debug, maintain, communicate, share, and reason with? This chapter explores how to specify the meaning of symbols in intelligent agents, how to use the meaning for knowledge-based debugging and explanation, and, finally, how an agent can represent its own reasoning and how this may be used to build knowledge-based systems. As Smith points out in the quote above, the problems of ontology are central for building intelligent computational agents.

13.1 Knowledge Sharing

Having an appropriate representation is only part of the story of building a knowledge-based agent. We also should be able to ensure that the knowledge can be acquired, particularly when the knowledge comes from diverse sources and at multiple points in time and should interoperate with other knowledge. We should also ensure that the knowledge can be reasoned about effectively.

Recall (page 61) that an **ontology** is a specification of the meanings of the symbols in an information system. Here an information system is a knowledge

base or some source of information, such as a thermometer. The meaning is sometimes just in the mind of the knowledge-base designer or in comments with the knowledge base. Increasingly, the specification of the meaning is in machine-interpretable form. This formal specification is important for **semantic interoperability** – the ability of different knowledge bases to work together.

Example 13.1 A purchasing agent has to know, when a web site claims it has a good price on "chips," whether these are potato chips, computer chips, wood chips, or poker chips. An ontology would specify meaning of the terminology used by the web site. Instead of using the symbol "chip", a web site that adheres to ontologies may use the symbol "WoodChipMixed" as defined by some particular organization that has published an ontology. By using this symbol and declaring which ontology it is from, it should be unambiguous as to which use of the word *chip* is meant. A formal representation of the web page would use "WoodChipMixed", which may get translated into English simply as "chip". If another information source uses the symbol "ChipOfWood", some third party may declare that the use of the term "ChipOfWood" in that information source corresponds to "WoodChipMixed" and therefore enable the information sources to be combined.

Before discussing how ontologies are specified, we first discuss how the logic of the previous chapter (with variables, terms, and relations) can be used to build flexible representations. These flexible representations allow for the modular addition of knowledge, including adding arguments to relations.

Given a specification of the meaning of the symbols, an agent can use that meaning for knowledge acquisition, explanation, and debugging at the knowledge level.

13.2 Flexible Representations

The first part of this chapter considers a way to build flexible representations using the tools of logic. These flexible representations are the basis of modern ontologies.

13.2.1 Choosing Individuals and Relations

Given a logical representation language, such as the one developed in the previous chapter, and a world to reason about, the people designing knowledge bases have to choose what, in the world, to refer to. That is, they have to choose what individuals and relations there are. It may seem that they can just refer to the individuals and relations that exist in the world. However, the world does not determine what individuals there are. How the world is divided into individuals is invented by whomever is modeling the world. The modeler divides up the world up into things so that the agent can refer to parts of the world that make sense for the task at hand.

Example 13.2 It may seem as though "*red*" is a reasonable property to ascribe to things in the world. You may do this because you want to tell the delivery robot to go and get the red parcel. In the world, there are surfaces absorbing some frequencies and reflecting other frequencies of light. Some user may have decided that, for some application, some particular set of reflectance properties should be called "red." Some other modeler of the domain might decide on another mapping of the spectrum and use the terms *pink*, *scarlet*, *ruby*, and *crimson*, and yet another modeler may divide the spectrum into regions that do not correspond to words in any language but are those regions most useful to distinguish different categories of individuals.

Just as modelers choose what individuals to represent, they also choose what relations to use. There are, however, some guiding principles that are useful for choosing relations and individuals. These will be demonstrated through a sequence of examples.

Example 13.3 Suppose you decide that "red" is an appropriate category for classifying individuals. You can treat the name *red* as a unary relation and write that parcel a is red:

$$red(a).$$

If you represent the color information in this way, then you can easily ask what is red:

ask $red(X)$.

The X returned are the red individuals.

With this representation, it is hard to ask the question, "What color is parcel a?" In the syntax of definite clauses, you cannot ask

ask $X(a)$.

because, in languages based on first-order logic (page 519), predicate names cannot be variables. In second-order or higher-order logic, this would return any property of a, not just its color.

There are alternative representations that allow you to ask about the color of parcel a. There is nothing in the world that forces you to make *red* a predicate. You could just as easily say that colors are individuals too, and you could use the constant *red* to denote the color red. Given that *red* is a constant, you can use the predicate *color* where $color(Ind, Val)$ means that physical individual *Ind* has color *Val*. "Parcel a is red" can now be written as

$$color(a, red).$$

What you have done is reconceive the world: the world now consists of colors as individuals that you can name. There is now a new binary relation *color* between physical individuals and colors. Under this new representation you can ask, "What color is block a?" with the query

ask $color(a, C)$.

To make an abstract concept into an object is to **reify** it. In the preceding example, we reified the color *red*.

Example 13.4 It seems as though there is no disadvantage to the new representation of colors in the previous example. Everything that could be done before can be done now. It is not much more difficult to write $color(X, red)$ than $red(X)$, but you can now ask about the color of things. So the question arises of whether you can do this to every relation, and what do you end up with?

You can do a similar analysis for the *color* predicate as for the *red* predicate in Example 13.3. The representation with *color* as a predicate does not allow you to ask the question, "Which property of parcel *a* has value *red*?," where the appropriate answer is "color." Carrying out a similar transformation to that of Example 13.3, you can view properties such as *color* as individuals, and you can invent a relation *prop* and write "individual *a* has the *color* of *red*" as

$$prop(a, color, red).$$

This representation allows for all of the queries of this and the previous example. You do not have to do this again, because you can write all relations in terms of the *prop* relation.

The **individual-property-value** representation is in terms of a single relation *prop* where

$$prop(Ind, Prop, Val)$$

means that individual *Ind* has value *Val* for property *Prop*. This is also called the **triple representation** because all of the relations are represented as **triples**. The first element of the triple is called the **subject**, the second is the **verb**, and the third is the **object**, using the analogy that a triple is a simple three-word sentence.

The verb of a triple is a **property**. The **domain** of property p is the set of individuals that can appear as the subject of a triple when p is the verb. The **range** of a property p is the set of values that can appear as the object of a triple that has p as the verb.

An **attribute** is a property–value pair. For example, an attribute of a parcel may be that its color is red. Two parcels may be identical if they have the same attributes – the same values for their properties.

There are some predicates that may seem to be too simple for the triple representation:

Example 13.5 To transform $parcel(a)$, which means that a is a parcel, there do not seem to be appropriate properties or values. There are two ways to transform this into the triple representation. The first is to reify the concept parcel and to say that a is a parcel:

$$prop(a, type, parcel).$$

Here *type* is a special property that relates an individual to a class. The constant *parcel* denotes the class that is the set of all, real or potential, things that are parcels. This triple specifies that the individual *a* is in the class *parcel*.

The second is to make parcel a property and write "*a* is a parcel" as

$$prop(a, parcel, true).$$

In this representation, *parcel* is a Boolean property which is true of things that are parcels.

A **Boolean property** is a property whose range is $\{true, false\}$, where *true* and *false* are constant symbols in the language.

Some predicates may seem to be too complicated for the triple representation:

Example 13.6 Suppose you want to represent the relation

$$scheduled(C, S, T, R),$$

which is to mean that section S of course C is scheduled to start at time T in room R. For example, "section 2 of course *cs*422 is scheduled to start at 10:30 in room *cc*208" is written as

$$scheduled(cs422, 2, 1030, cc208).$$

To represent this in the triple representation, you can invent a new individual, a *booking*. Thus, the *scheduled* relationship is reified into a booking individual.

A booking has a number of properties, namely a course, a section, a start time, and a room. To represent "section 2 of course *cs*422 is scheduled at 10:30 in room *cc*208," you name the booking, say, the constant *b*123, and write

$$prop(b123, course, cs422).$$
$$prop(b123, section, 2).$$
$$prop(b123, start\_time, 1030).$$
$$prop(b123, room, cc208).$$

This new representation has a number of advantages. The most important is that it is modular; which values go with which properties can easily be seen. It is easy to add new properties such as the instructor or the duration. With the new representation, it is easy to add that "Fran is teaching section 2 of course *cs*422, scheduled at 10:30 in room *cc*208" or that the duration is 50 minutes:

$$prop(b123, instructor, fran).$$
$$prop(b123, duration, 50).$$

With *scheduled* as a predicate, it was very difficult to add the instructor or duration because it required adding extra arguments to every instance of the predicate.

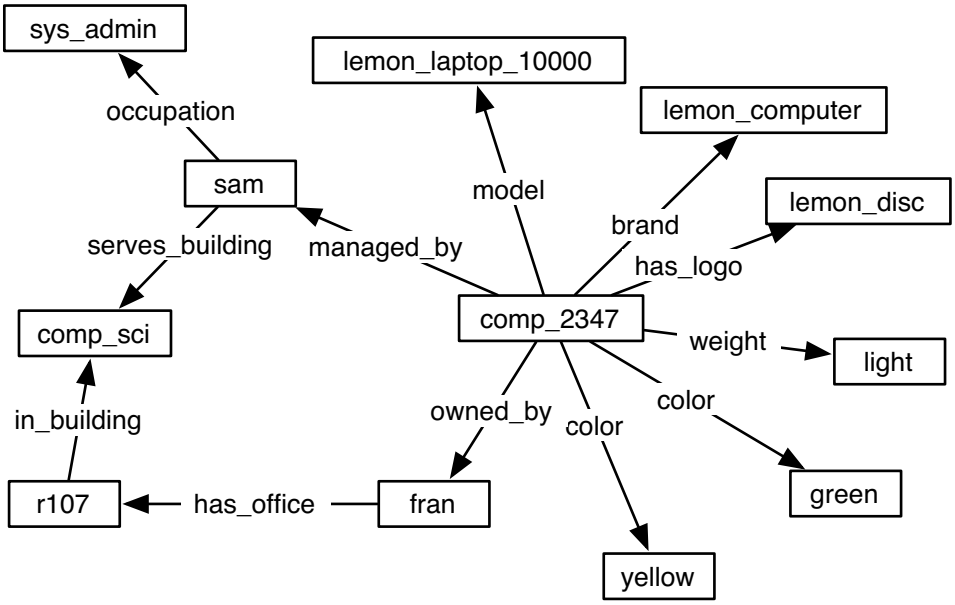

Figure 13.1: A semantic network

13.2.2 Graphical Representations

You can interpret the *prop* relation in terms of a graph, where the relation

$$prop(Ind, Prop, Val)$$

is depicted with *Ind* and *Val* as nodes with an arc labeled with *Prop* between them. Such a graph is called a **semantic network**. Given such a graphical representation, there is a straightforward mapping into a knowledge base using the *prop* relation.

Example 13.7 Figure 13.1 shows a semantic network for the delivery robot showing the sort of knowledge that the robot may have about a particular computer. Some of the knowledge represented in the network is

$prop(comp\_2347, owned\_by, fran).$
$prop(comp\_2347, managed\_by, sam).$
$prop(comp\_2347, model, lemon\_laptop\_10000).$
$prop(comp\_2347, brand, lemon\_computer).$
$prop(comp\_2347, has\_logo, lemon\_disc).$
$prop(comp\_2347, color, green).$
$prop(comp\_2347, color, yellow).$
$prop(comp\_2347, weight, light).$
$prop(fran, has\_office, r107).$
$prop(r107, in\_building, comp\_sci).$

The network also shows how the knowledge is structured. For example, it is easy to see that computer number 2347 is owned by someone (Fran) whose office (r107) is in the *comp_sci* building. The direct indexing evident in the graph can be used by humans and machines.

This graphical notation has a number of advantages:

- It is easy for a human to see the relationships without being required to learn the syntax of a particular logic. The graphical notation helps the builders of knowledge bases to organize their knowledge.
- You can ignore the labels of nodes that just have meaningless names – for example, the name *b*123 in Example 13.6 (page 555), or *comp*_2347 in Figure 13.1. You can just leave these nodes blank and make up an arbitrary name if you must map to the logical form.

Terse Language for Triples

Turtle is a simple language for representing **triples**. It is one of the languages invented for the **semantic web** (page 566). It is one of the syntaxes used for the **Resource Description Framework**, or **RDF**, growing out of a similar language called **Notation 3** or **N3**.

In Turtle and RDF everything – including individuals, classes, and properties – is a **resource**. A **Uniform Resource Identifier** (**URI**) is a unique name that can be used to identify anything. A URI is written within angle brackets, and it often has the form of a URL because URLs are unique. For example, ⟨http://aispace.org⟩ can be a URI. A "#" in a URI denotes an individual that is referred to in a web page. For example, ⟨http://cs.ubc.ca/~poole/foaf.rdf#david⟩ denotes the individual *david* referred to in http://cs.ubc.ca/~poole/foaf.rdf. The URI ⟨⟩ refers to the current document, so the URI ⟨#comp_2347⟩ denotes an individual defined in the current document.

A triple is written simply as

Subject Verb Object.

where *Subject* and *Verb* are URIs, and *Object* is either a URI or a literal (string or number). *Verb* denotes a property. *Object* is the value of the property *Verb* for *Subject*.

Example 13.8 The triples of Example 13.7 are written in Turtle as follows:

⟨#comp_2347⟩ ⟨#owned_by⟩ ⟨#fran⟩ .
⟨#comp_2347⟩ ⟨#managed_by⟩ ⟨#sam⟩ .
⟨#comp_2347⟩ ⟨#model⟩ ⟨#lemon_laptop_10000⟩ .
⟨#comp_2347⟩ ⟨#brand⟩ ⟨#lemon_computer⟩ .
⟨#comp_2347⟩ ⟨#has_logo⟩ ⟨#lemon_disc⟩ .
⟨#comp_2347⟩ ⟨#color⟩ ⟨#green⟩ .
⟨#comp_2347⟩ ⟨#color⟩ ⟨#yellow⟩ .
⟨#fran⟩ ⟨#has_office⟩ ⟨#r107⟩ .
⟨#r107⟩ ⟨#serves_building⟩ ⟨#comp_sci⟩ .

The identifier "fran" does not tell us the name of the individual. If we wanted to say that the person's name is Fran, we would write

⟨#fran⟩ ⟨#name⟩ "Fran".

There are some useful abbreviations used in Turtle. A comma is used to group objects with the same subject and verb. That is,

$S\ V\ O_1, O_2$.

is an abbreviation for

$S\ V\ O_1$.
$S\ V\ O_2$.

A semicolon is used to group verb–object pairs for the same subject. That is,

$S\ V_1\ O_1; V_2\ O_2$.

is an abbreviation for

$S\ V_1\ O_1$.
$S\ V_2\ O_2$.

Square brackets are to define an individual that is not given an identifier. This unnamed resource is used as the object of some triple, but otherwise cannot be referred to. Both commas and semicolons can be used to give this resource properties. Thus,

$[V_1\ O_1; V_2\ O_2]$

is an individual that has value O_1 on property V_1 and has value O_2 on property V_2. Such descriptions of unnamed individuals are sometimes called **frames**. The verbs are sometimes called **slots** and the objects are **fillers**.

Example 13.9 The Turtle sentence

⟨comp_3645⟩ ⟨#owned_by⟩ ⟨#fran⟩ ;
⟨#color⟩ ⟨#green⟩ , ⟨#yellow⟩ ;
⟨#managed_by⟩ [⟨#occupation⟩ ⟨#sys_admin⟩ ;
⟨#serves_building⟩ ⟨#comp_sci⟩].

says that ⟨comp_3645⟩ is owned by ⟨#fran⟩, its color is green and yellow, and it is managed by a resource whose occupation is system administration and who serves the *comp_sci* building.

This is an abbreviation for the triples

⟨comp_3645⟩ ⟨#owned_by⟩ ⟨#fran⟩ .
⟨comp_3645⟩ ⟨#color⟩ ⟨#green⟩ .
⟨comp_3645⟩ ⟨#color⟩ ⟨#yellow⟩ .
⟨comp_3645⟩ ⟨#managed_by⟩ ⟨i2134⟩ .
⟨i2134⟩ ⟨#occupation⟩ ⟨#sys_admin⟩ .
⟨i2134⟩ ⟨#serves_building⟩ ⟨#comp_sci⟩ .

but where the made-up URI, ⟨i2134⟩, cannot be referred to elsewhere.

It is difficult for a reader to know what the authors mean by a particular URI such as ⟨#name⟩ and how the use of this term relates to other people's use of the same term. There are, however, people who have agreed on certain meaning for specific terms. For example, the property ⟨http://xmlns.com/foaf/0.1/#name⟩ has a standard definition as the name of an object. Thus, if we write

⟨#fran⟩ ⟨http://xmlns.com/foaf/0.1/#name⟩ "Fran".

we mean the particular *name* property having that agreed-on definition.

It does not matter what is at the URL http://xmlns.com/foaf/0.1/, as long as those who use the URI ⟨http://xmlns.com/foaf/0.1/#name⟩ all mean the same property. That URL, at the time of writing, just redirects to a web page. However, the "friend of a friend" project (which is what "foaf" stands for) uses that name space to mean something. This works simply because people use it that way.

In Turtle, URIs can be abbreviated using a "name:" to replace a URL and the angle brackets, using an "@prefix" declaration. For example,

@prefix foaf: ⟨http://xmlns.com/foaf/0.1/#⟩

lets "foaf:name" be an abbreviation for ⟨http://xmlns.com/foaf/0.1/#name⟩. Similarly,

@prefix : ⟨#⟩

lets us write ⟨#color⟩ as :color.

Turtle also allows for parentheses for arguments to functions that are not reified. It also uses the abbreviation "a" for "rdf:type", but we do not follow that convention.

Classes in Knowledge Bases and Object-Oriented Programming

The use of "individuals" and "classes" in knowledge-based systems is very similar to the use of "objects" and "classes" in **object-oriented programming (OOP) languages** such as **Smalltalk** or **Java**. This should not be too surprising because they have an interrelated history. There are important differences that tend to make the direct analogy often more confusing than helpful:

- Objects in OOP are computational objects; they are data structures and associated programs. A "person" object in Java is not a person. However, individuals in a knowledge base (KB) are (typically) things in the real world. A "person" individual in a KB can be a real person. A "chair" individual can be a real chair you can actually sit in; it can hurt you if you bump into it. You can send a message to, and get answers from, a "chair" object in Java, whereas a chair in the real world tends to ignore what you tell it. A KB is not typically used to interact with a chair, but to reason *about* a chair. A real chair stays where it is unless it is moved by a physical agent.
- In a KB, a representation of an object is only an approximation at one (or a few) levels of abstraction. Real objects tend to be much more complicated than what is represented. We typically do not represent the individual fibers in a chair. In an OOP system, there are only the represented properties of an object. The system can know everything about a Java object, but not about a real individual.
- The class structure of Java is intended to represent designed objects. A systems analyst or a programmer gets to create a design. For example, in Java, an object is only a member of one lowest-level class. There is no multiple inheritance. Real objects are not so well behaved. The same person could be a football coach, a mathematician, and a mother.
- A computer program cannot be uncertain about its data structures; it has to select particular data structures to use. However, we can be uncertain about the types of things in the world.
- The representations in a KB do not actually do anything. In an OOP system, objects do computational work. In a KB, they just represent – that is, they just refer to objects in the world.
- Whereas an object-oriented modeling language, like **UML**, may be used for representing KBs, it may not be the best choice. A good OO modeling tool has facilities to help build good designs. However, the world being modeled may not have a good design at all. Trying to force a good design paradigm on a messy world may not be productive.

13.2.3 Primitive Versus Derived Relations

Typically, you know more about a domain than a database of facts; you know general rules from which other facts can be derived. Which facts are explicitly given and which are derived is a choice to be made when designing and building a knowledge base.

Primitive knowledge is knowledge that is defined explicitly by facts. **Derived knowledge** is knowledge that can be inferred from other knowledge. Derived knowledge is usually specified in terms of rules.

The use of rules allows for a more compact representation of knowledge. Derived relations allow for conclusions to be drawn from observations of the domain. This is important because you do not directly observe everything about a domain. Much of what is known about a domain is inferred from the observations and more general knowledge.

A standard way to use derived knowledge is to put individuals into classes. We give general properties to classes so that individuals inherit the properties of classes. The reason we group individuals into classes is because the members of a class have attributes in common, or they have common properties that make sense for them (see the box on page 569).

A **class** is the set of those actual and potential individuals that would be members of the class. In logic, this is an **intensional** set, defined by a **characteristic function** that is true of members of the set and false of other individuals. The alternative is an **extensional** set, which is defined by listing its elements.

For example, the class *chair* is the set of all things that would be chairs. We do not want the definition to be the set of things that *are* chairs, because chairs that have not yet been built also fall into the class of chairs. We do not want two classes to be equivalent just because they have the same members. For example, the class of green unicorns and the class of chairs that are exactly 124 meters high are different classes, even though they contain the same elements; they are both empty.

The definition of class allows any set that can be described to be a class. For example, the set consisting of the number 17, the Tower of London, and Julius Caesar's left foot may be a class, but it is not very useful. A **natural kind** is a class where describing objects using the class is more succinct than describing objects without the class. For example, "mammal" is a natural kind, because describing the common attributes of mammals makes a knowledge base that uses "mammal" more succinct than one that does not use "mammal" and repeats the attributes for every individual.

We use the property *type* to mean "is a member of class." Thus, in the language of definite clauses,

$$prop(X, type, C)$$

means that individual X is a member of class C.

The people who created **RDF** and **RDF Schema** used exactly the property we want to use here for membership in a class. In the language Turtle, we can

define the abbreviation

@prefix rdf: ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#⟩ .

@prefix rdfs: ⟨http://www.w3.org/2000/01/rdf-schema#⟩ .

Given these declarations, **rdf:type** means the type property that relates an individual to a class of which it is a member. By referring to the definition of **type** at that URI, this becomes a standard definition that can be used by others and can be distinguished from other meanings of the word "type."

The property **rdfs:subClassOf** between classes specifies that one class is a subset of another. In Turtle,

S rdfs:subClassOf *C*.

means that class *S* is a subclass of class *C*. In terms of sets, this means that *S* is a subset of *C*. That is, every individual of type *S* is of type *C*.

Example 13.10 Example 13.7 explicitly specified that the logo for computer *comp_2347* was a lemon disc. You may, however, know that all Lemon-brand computers have this logo. An alternative representation is to associate the logo with *lemon_computer* and derive the logo of *comp_2347*. The advantage of this representation is that if you find another Lemon-brand computer, you can infer its logo.

In Turtle,

```
:lemon_computer rdfs:subClassOf :computer.
:lemon_laptop_10000 rdfs:subClassOf :lemon_computer.
:comp_2347 rdf:type :lemon_laptop_10000.
```

says that a lemon computer is a computer, a lemon laptop 10000 is a lemon computer, and that *comp_2347* is a lemon laptop 10000. An extended example is shown in Figure 13.2, where the shaded rectangles are classes, and arcs from classes are not the properties of the class but properties of the members of the class.

The relationship between types and subclasses can be written as a definite clause:

$$prop(X, type, C) \leftarrow prop(X, type, S) \wedge prop(S, subClassOf, C)$$

You can treat *type* and *subClassOf* as special properties that allow **property inheritance**. Property inheritance is when a value for a property is specified at the class level and inherited by the members of the class. If all members of class *c* have value *v* for property *p*, this can be written in Datalog as

$$prop(Ind, p, v) \leftarrow prop(Ind, type, c).$$

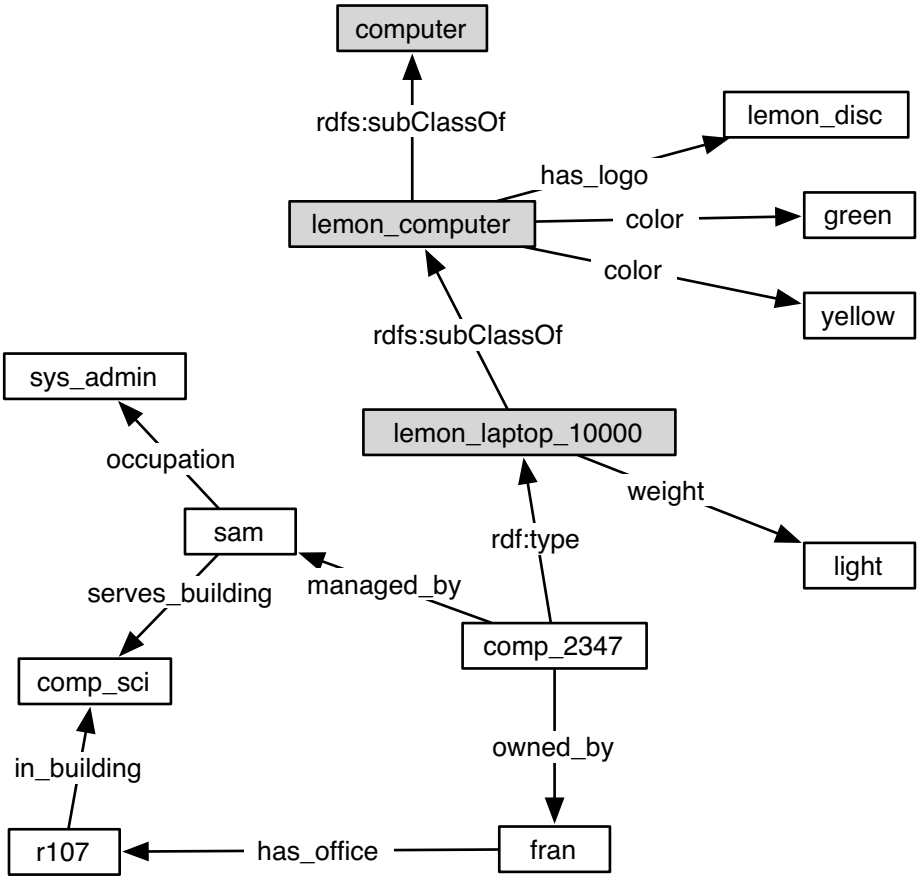

Figure 13.2: A semantic network allowing inheritance

which, together with the aforementioned rule that relates types and subclasses, can be used for property inheritance.

Example 13.11 All lemon computers have a lemon disc as a logo and have color yellow and color green (see the *logo* and *color* arcs in Figure 13.2). This can be represented by the following Datalog program:

$$prop(X, has\_logo, lemon\_disc) \leftarrow$$
$$\quad prop(X, type, lemon\_computer).$$
$$prop(X, color, green) \leftarrow$$
$$\quad prop(X, type, lemon\_computer).$$
$$prop(X, color, yellow) \leftarrow$$
$$\quad prop(X, type, lemon\_computer).$$

The *prop* relations that can be derived from these clauses are essentially the same as that which can be derived from the flat semantic network of Figure 13.1 (page 556). With the structured representation, to incorporate a new Lemon

```
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#>.
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix owl:   <http://www.w3.org/2002/07/owl#>.
@prefix :      <#>.

:computer    rdf:type  rdfs:Class.
:logo        rdf:type  rdfs:Class.
:lemon_disc  rdf:type  :logo.
:has_logo
      rdf:type    rdf:Property ;
      rdfs:domain :computer ;
      rdfs:range  :logo.
:lemon_computer
      rdf:type        rdfs:Class ;
      rdfs:subClassOf :computer ;
      rdfs:subClassOf
           owl:ObjectHasValue(:has_logo :lemon_disc).
```

Figure 13.3: Turtle representation of Example 13.12

Laptop 10000, you only declare that it is a Lemon Laptop 10000 and the color and logo properties can be derived through inheritance.

RDF and Turtle do not have definite clauses. In these languages, instead of treating the membership in a class as a predicate, classes are sets. To say that all of the elements of a set S have value v for a predicate p, we say that S is a subset of the set of all things with value v for predicate p.

Example 13.12 To state that all lemon computers have a lemon disc as a logo, we say that the set of lemon computers is a subset of the set of all things for which the property *has_logo* value *lemon_disc*.

A representation of this is shown in Figure 13.3. :computer and :logo are both classes. :lemon_disc is member of the class :logo. :has_logo is a property, with domain :computer and range :logo. :lemon_computer is a subclass of :computer. It is also a subclass of the set of all individuals that have value :lemon_disc for the property :has_logo.

owl:ObjectHasValue is a class constructor for OWL (see below), such that owl:ObjectHasValue(:has_logo :lemon_disc) is the class of all individuals that have the value :lemon_disc for the property :has_logo.

Some general guidelines are useful for deciding what should be primitive and what should be derived:

- When associating an attribute with an individual, select the most general class C, such that the individual is in C and all members of C have that attribute, and associate the attribute with class C. Inheritance can be used

to derive the attribute for the individual and all members of class C. This representation methodology tends to make knowledge bases more concise, and it means that it is easier to incorporate new individuals because they automatically inherit the attribute if they are a member of class C.

- Do not associate a contingent attribute of a class with the class. A **contingent attribute** is one whose value changes when circumstances change. For example, it may be true of the current computer environment that all of the computers come in brown boxes. However, it may not be a good idea to put that as an attribute of the *computer* class, because it would not be expected to be true as other computers are bought.
- Axiomatize in the causal direction (page 204). If a choice exists between making the cause primitive or the effect primitive, make the cause primitive. The information is then more likely to be stable when the domain changes.

13.3 Ontologies and Knowledge Sharing

Building large knowledge-based systems is complex because

- Knowledge often comes from multiple sources and must be integrated. Moreover, these sources may not have the same division of the world. Often knowledge comes from different fields that have their own distinctive terminology and divide the world according to their own needs.
- Systems evolve over time and it is difficult to anticipate all future distinctions that should be made.
- The people involved in designing a knowledge base must choose what individuals and relationships to represent. The world is not divided into individuals; that is something done by intelligent agents to understand the world. Different people involved in a knowledge-based system should agree on this division of the world.
- It is difficult to remember what your own notation means, let alone to discover what someone else's notation means. This has two aspects:
  - given a symbol used in the computer, determining what it means;
  - given a concept in someone's mind, determining what symbol they should use; that is, determining whether the concept has been used before and, if it has, discovering what notation has been used for it.

To share and communicate knowledge, it is important to be able to come up with a common vocabulary and an agreed-on meaning for that vocabulary.

A **conceptualization** is a mapping between symbols used in the computer (i.e., the vocabulary) and the individuals and relations in the world. It provides a particular abstraction of the world and notation for that abstraction. A conceptualization for small knowledge bases can be in the head of the designer or specified in natural language in the documentation. This informal specification of a conceptualization does not scale to larger systems where the conceptualization must be shared.

The Semantic Web

The **semantic web** is a way to allow machine-interpretable knowledge to be distributed on the World Wide Web. Instead of just serving HTML pages that are meant to be read by humans, web sites will also provide information that can be used by computers.

At the most basic level, **XML** (the Extensible Markup Language) provides a syntax designed to be machine readable, but which is possible for humans to read. It is a text-based language, where items are tagged in a hierarchical manner. The syntax for XML can be quite complicated, but at the simplest level, the scope of a tag is either in the form $\langle tag \ldots / \rangle$, or in the form $\langle tag \ldots \rangle \ldots \langle / tag \rangle$.

A **URI** (a **Uniform Resource Identifier**) is used to uniquely identify a resource. A **resource** is anything that can be uniquely identified. A URI is a string that refers to a resource, such as a web page, a person, or a corporation. Often URIs use the syntax of web addresses.

RDF (the **Resource Description Framework**) is a language built on XML, providing individual-property-value triples.

RDF-S (RDF Schema) lets you define resources (and so also properties) in terms of other resources (e.g., using *subClassOf*). RDF-S also lets you restrict the domain and range of properties and provides containers (sets, sequences, and alternatives – one of which must be true).

RDF allows sentences in its own language to be reified. This means that it can represent arbitrary logical formulas and so is not decidable in general. Undecidability is not necessarily a bad thing; it just means that you cannot put a bound on the time a computation may take. Simple logic programs with function symbols and virtually all programming languages are undecidable.

OWL (the **Web Ontology Language**) is an ontology language for the World Wide Web. It defines some classes and properties with a fixed interpretation that can be used for describing classes, properties, and individuals. It has built-in mechanisms for equality of individuals, classes, and properties, in addition to restricting domains and ranges of properties and other restrictions on properties (e.g., transitivity, cardinality).

There have been some efforts to build large universal ontologies, such as **cyc** (www.cyc.com), but the idea of the semantic web is to allow communities to converge on ontologies. Anyone can build an ontology. People who want to develop a knowledge base can use an existing ontology or develop their own ontology, usually built on existing ontologies. Because it is in their interest to have semantic interoperability, companies and individuals should tend to converge on standard ontologies for their domain or to develop mappings from their ontologies to others' ontologies.

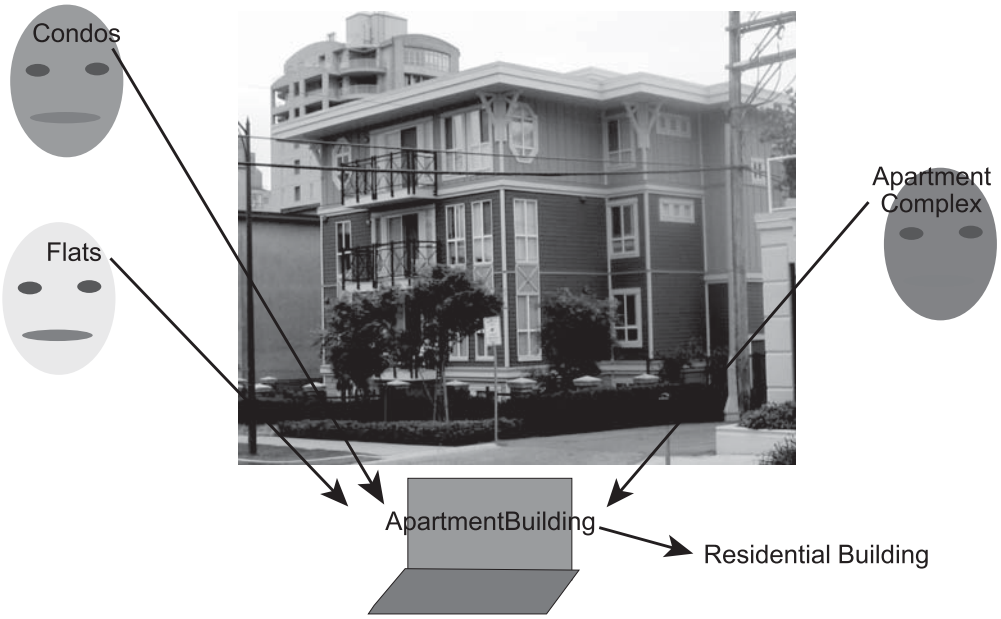

Figure 13.4: Mapping from a conceptualization to a symbol

In philosophy, **ontology** is the study of what exists. In AI, an **ontology** is a specification of the meanings of the symbols in an information system. That is, it is a specification of a conceptualization. It is a specification of what individuals and relationships are assumed to exist and what terminology is used for them. Typically, it specifies what types of individuals will be modeled, specifies what properties will be used, and gives some axioms that restrict the use of that vocabulary.

Example 13.13 An ontology of individuals that could appear on a map could specify that the symbol "ApartmentBuilding" will represent apartment buildings. The ontology will not *define* an apartment building, but it will describe it well enough so that others can understand the definition. We want other people, who may be inclined to use different symbols, to be able to use the ontology to find the appropriate symbol to use (see Figure 13.4). Multiple people are able to use the symbol consistently. An ontology should also enable a person to verify what a symbol means. That is, given a concept, they want to be able to find the symbol, and, given the symbol, they want to be able to determine what it means.

An ontology may give axioms to restrict the use of some symbol. For example, it may specify that apartment buildings are buildings, which are human-constructed artifacts. It may give some restriction on the size of buildings so that shoeboxes cannot be buildings or that cities cannot be buildings. It may state that a building cannot be at two geographically dispersed locations at the same time (so if you take off some part of the building and move it to a differ-

ent location, it is no longer a single building). Because apartment buildings are buildings, these restrictions also apply to apartment buildings.

Ontologies are usually written independently of a particular application and often involve a community to agree on the meanings of symbols. An ontology consists of

- a vocabulary of the categories of the things (both classes and properties) that a knowledge base may want to represent;
- an organization of the categories, for example into an inheritance hierarchy using *subClassOf* or *subPropertyOf*, or using Aristotelian definitions; and
- a set of axioms restricting the meanings of some of the symbols to better reflect their meaning – for example, that some property is transitive, or that the domain and range are restricted, or that there are some restriction on the number of values a property can take for each individual. Sometimes relationships are defined in terms of more primitive relationships but, ultimately, the relationships are grounded out into **primitive** relationships that are not actually defined.

An ontology does not specify the individuals not known at design time. For example, an ontology of buildings would typically not include actual buildings. An ontology would specify those individuals that are fixed and should be shared, such as the days of the week, or colors.

Example 13.14 Consider a trading agent that is designed to find accommodations. Users could use such an agent to describe what accommodation they want. The trading agent could search multiple knowledge bases to find suitable accommodations or to notify users when some appropriate accommodation becomes available. An ontology is required to specify the meaning of the symbols for the user and to allow the knowledge bases to interoperate. It provides the semantic glue to tie together the users' needs with the knowledge bases.

In such a domain, houses and apartment buildings may both be residential buildings. Although it may be sensible to suggest renting a house or an apartment in an apartment building, it may not be sensible to suggest renting an apartment building to someone who does not actually specify that they want to rent the whole building. A "living unit" could be defined to be the collection of rooms that some people, who are living together, live in. A living unit may be what a rental agency offers to rent. At some stage, the designer may have to decide whether a room for rent in a house is a living unit, or even whether part of a shared room that is rented separately is a living unit. Often the boundary cases – cases that may not be initially anticipated – are not clearly delineated but become better defined as the ontology evolves.

The ontology would not contain descriptions of actual houses or apartments because the actual available accommodation would change over time and would not change the meaning of the vocabulary.

The primary purpose of an ontology is to document what the symbols mean – the mapping between symbols (in a computer) and concepts (in someone's

Aristotelian Definitions

Categorizing objects, the basis for modern ontologies, has a long history. Aristotle [350 B.C.] suggested the definition of a class C in terms of

- **Genus**: a superclass of C. The plural of genus is genera.
- **Differentia**: the properties that make members of the class C different from other members of the superclass of C.

He anticipated many of the issues that arise in definitions:

> *If genera are different and co-ordinate, their differentiae are themselves different in kind. Take as an instance the genus "animal" and the genus "knowledge". "With feet", "two-footed", "winged", "aquatic", are differentiae of "animal"; the species of knowledge are not distinguished by the same differentiae. One species of knowledge does not differ from another in being "two-footed".* [Aristotle, 350 B.C.]

Note that "co-ordinate" here means neither is subordinate to the other.

In the style of modern ontologies, we would say that "animal" is a class, and "knowledge" is a class. The property "two-footed" has domain "animal". If something is an instance of knowledge, it does not have a value for the property "two-footed".

To build an ontology based on **Aristotelian definitions**:

- For each class you may want to define, determine a relevant superclass, and then select those attributes that distinguish the class from other subclasses. Each attribute gives a property and a value.
- For each property, define the most general class for which it makes sense, and define the domain of the property to be this class. Make the range another class that makes sense (perhaps requiring this range class to be defined, either by enumerating its values or by defining it using an Aristotelian definition).

This can get quite complicated. For example, defining "luxury furniture", perhaps the superclass you want is "furniture" and the distinguishing characteristics are cost is high and luxury furniture is soft. The softness of furniture is different than the softness of rocks. You also probably want to distinguish the squishiness from the texture (both of which may be regarded as soft).

This methodology does not, in general, give a tree hierarchy of classes. Objects can be in many classes. Each class does not have a single most-specific superclass. However, it is still straightforward to check whether one class is a subclass of another, to check the meaning of a class, and to determine the class that corresponds to a concept in your head.

In rare cases, this results in a tree structure, most famously in the **Linnaean taxonomy** of living things. It seems that the reason this is a tree is because of evolution. Trying to force a tree structure in other domains has been much less successful.

head). Given a symbol, a person is able to use the ontology to determine what it means. When someone has a concept to be represented, the ontology is used to find the appropriate symbol or to determine that the concept does not exist in the ontology. The secondary purpose, achieved by the use of axioms, is to allow inference or to determine that some combination of values is inconsistent. The main challenge in building an ontology is the organization of the concepts to allow a human to map concepts into symbols in the computer, and for the computer to infer useful new knowledge from stated facts.

13.3.1 Description Logic

A **Uniform Resource Identifier** has some meaning because someone published that it has that meaning and because people use it with that meaning. This works, but we want more. We would like to have meanings that allow a computer to do some inference.

Modern ontology languages such as **OWL** (page 566) are based on **description logics**. A description logic is used to describe classes, properties, and individuals. One of the main ideas behind a description logic is to separate

- a **terminological knowledge base** that describes the terminology, which should remain constant as the domain being modeled changes, and
- an **assertional knowledge base** that describes what is true in some domain at some point in time.

Usually, the terminological knowledge base is defined at the design time of the system and defines the ontology, and it only changes as the meaning of the vocabulary changes, which should be rare. The assertional knowledge base usually contains the knowledge that is situation-specific and is only known at run time.

It is typical to use triples (page 554) to define the assertional knowledge base and a language such as OWL to define the terminological knowledge base.

OWL describes domains in terms of the following:

- **Individuals** are things in the world that is being described (e.g., a particular house or a particular booking may be individuals).
- **Classes** are sets of individuals. A class is the set of all real or potential things that would be in that class. For example, the class "House" may be the set of all things that would be classified as a house, not just those houses that exist in the domain of interest.
- **Properties** are used to describe individuals. A **datatype property** has values that are primitive data types, such as integers or strings. For example, "streetName" may be a datatype property between a street and string. An **object property** has values that are other individuals. For example, "nextTo" may be a property between two houses, and "onStreet" may be a property between a house and a street.

OWL comes in three variants that differ in restrictions imposed on the classes and properties. In OWL-DL and OWL-Lite, a class cannot be an individual or

C_k are classes, P_k are properties, I_k are individuals, and n is an integer. $\#S$ is the number of elements in set S:

Class	Class Contains
owl:Thing	all individuals
owl:Nothing	no individuals (empty set)
owl:ObjectIntersectionOf$(C_1, \ldots, C_k)$	individuals in $C_1 \cap \cdots \cap C_k$
owl:ObjectUnionOf$(C_1, \ldots, C_k)$	individuals in $C_1 \cup \cdots \cup C_k$
owl:ObjectComplementOf(C)	the individuals not in C
owl:ObjectOneOf$(I_1, \ldots, I_k)$	$I_1, \ldots, I_k$
owl:ObjectHasValue(P, I)	individuals with value I on property P, i.e., $\{x : x\ P\ I\}$
owl:ObjectAllValuesFrom(P, C)	individuals with all values in C on property P; i.e., $\{x : x\ P\ y \rightarrow y \in C\}$
owl:ObjectSomeValuesFrom(P, C)	individuals with some values in C on property P; i.e., $\{x : \exists y \in C$ such that $x\ P\ y\}$
owl:ObjectMinCardinality(n, P, C)	individuals x with at least n individuals of class C related to x by P, i.e., $\{x : \#\{y \mid xPy$ and $y \in C\} \geq n\}$
owl:ObjectMaxCardinality(n, P, C)	individuals x with at most n individuals of class C related to x by P, i.e., $\{x : \#\{y \mid xPy$ and $y \in C\} \leq n\}$

Figure 13.5: Some OWL built-in classes and class constructors

a property, and a property is not an individual. In OWL-Full, the categories of individuals, properties, and classes are not necessarily disjoint. OWL-Lite has some syntactic restrictions that do not affect the meaning but can make reasoning simpler.

OWL does not make the unique names assumption (page 536); two names do not necessarily denote different individuals or different classes. It also does not make the complete knowledge assumption (page 192); it does not assume that all relevant facts have been stated.

Figure 13.5 gives some primitive classes and some class constructors. This figure uses set notation to define the set of individuals in a class. Figure 13.6 (on the next page) gives primitive predicates of OWL. The owl: prefixes are from OWL. To use these properties and classes in an ontology, you include the appropriate URI abbreviations:

```
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#>.
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix owl:   <http://www.w3.org/2002/07/owl#>.
```

OWL has the following predicates with a fixed interpretation, where C_k are classes, P_k are properties, and I_k are individuals; x and y are universally quantified variables.

Statement	Meaning
rdf:type(I, C)	$I \in C$
rdfs:subClassOf(C_1, C_2)	$C_1 \subseteq C_2$
owl:EquivalentClasses(C_1, C_2)	$C_1 \equiv C_2$
owl:DisjointClasses(C_1, C_2)	$C_1 \cap C_2 = \{\}$
rdfs:domain(P, C)	if xPy then $x \in C$
rdfs:range(P, C)	if xPy then $y \in C$
rdfs:subPropertyOf(P_1, P_2)	xP_1y implies xP_2y
owl:EquivalentObjectProperties(P_1, P_2)	xP_1y if and only if xP_2y
owl:DisjointObjectProperties(P_1, P_2)	xP_1y implies not xP_2y
owl:InverseObjectProperties(P_1, P_2)	xP_1y if and only if yP_2x
owl:SameIndividual$(I_1, \ldots, I_n)$	$\forall j \forall k\ I_j = I_k$
owl:DifferentIndividuals$(I_1, \ldots, I_n)$	$\forall j \forall k\ j \neq k$ implies $I_j \neq I_k$
owl:FunctionalObjectProperty(P)	if xPy_1 and xPy_2 then $y_1 = y_2$
owl:InverseFunctionalObjectProperty(P)	if x_1Py and x_2Py then $x_1 = x_2$
owl:TransitiveObjectProperty(P)	if xPy and yPz then $y = z$
owl:SymmetricObjectProperty	if xPy then yPz

Figure 13.6: Some RDF, RDF-S, and OWL built-in predicates

In these figures, xPy is a triple. Note that this is meant to define the meaning of the predicates, rather than any syntax. The predicates can be used with different syntaxes, such as XML, Turtle, or traditional relations notation.

There is one property constructor: owl:ObjectInverseOf(P), which is the inverse property of P; that is, it is the property P^{-1} such that $yP^{-1}x$ iff xPy. Note that it is only applicable to object properties; datatype properties do not have inverses, because data types cannot be the subject of a triple.

The list of classes and statements in these figures is not complete. There are corresponding datatype classes for datatype properties, where appropriate. For example, owl:DataSomeValuesFrom and owl:EquivalentDataProperties have the same definitions as the corresponding object symbols, but are for datatype properties. There are also other constructs in OWL to define properties, comments, annotations, versioning, and importing other ontologies.

Example 13.15 As an example of a class constructor in Turtle notation, which uses spaces between arguments,

```
owl:MinCardinality(2 :owns :building)
```

is the class of all individuals who own two or more buildings. That is, it is the set $\{x : \exists i_1 \exists i_2\ x$:owns i_1 and x :owns i_2 and $i_1 \neq i_2\}$. This class constructor must

be used in a statement, for example, to say that some individual is a member of this class or to say that this is equivalent to some other class.

Example 13.16 Consider an Aristotelian definition (page 569) of an apartment building. We can say that an apartment building is a residential building with multiple units and the units are rented. (This is in contrast to a condominium building, where the units are individually sold, or a house, where there is only one unit). Suppose we have the class *ResidentialBuilding* that is a subclass of *Building*.

We first define the functional object property *numberOfUnits*, with domain *ResidentialBuilding* and range $\{one, two, moreThanTwo\}$. In Turtle this is written

```
:numberOfUnits rdf:type    owl:FunctionalObjectProperty;
               rdfs:domain :ResidentialBuilding;
               rdfs:range  owl:OneOf(:one :two :moreThanTwo).
```

The functional object property *ownership* with domain *ResidentialBuilding*, and range $\{rental, ownerOccupied, coop\}$ can be defined similarly.

We can define an apartment building as a *ResidentialBuilding* where the *numberOfUnits* property has the value *moreThanTwo* and the *ownership* property has the value rental. To specify this in OWL, we define the class of things that have value *moreThanTwo* for the property *numberOfUnits*, the class of things that have value *rental* for the property *ownership*, and say that *ApartmentBuilding* is equivalent to the intersection of these classes. In Turtle, this is

```
:ApartmentBuilding
   owl:EquivalentClasses
        owl:ObjectIntersectionOf (
             owl:ObjectHasValue(:numberOfUnits :moreThanTwo)
             owl:ObjectHasValue(:onwership :rental)
            :ResidentialBuilding).
```

This definition can be used to answer questions about apartment buildings, such as the ownership and the number of units.

Note that the previous example did not really define *ownership*. The system has no idea what this actually means. Hopefully, a user will know what it means. Someone who wants to adopt an ontology should ensure that they use a property and a class to mean the same thing as other users of the ontology.

A **domain ontology** is an ontology about a particular domain of interest. Most existing ontologies are in a narrow domain that people write for specific applications. There are some guidelines that have evolved for writing domain ontologies to enable knowledge sharing:

- If possible, use an existing ontology. This means that your knowledge base will be able to interact with others who use the same ontology.
- If an existing ontology does not exactly match your needs, import it and add to it. Do not start from scratch, because then others who want to use the best ontology will have to choose. If your ontology includes and improves the other, others who want to adopt an ontology will choose yours, because their application will be able to interact with adopters of either ontology.

- Make sure that your ontology integrates with neighboring ontologies. For example, an ontology about resorts will have to interact with ontologies about food, beaches, recreation activities, and so on. Try to make sure that it uses the same terminology for the same things.
- Try to fit in with higher-level ontologies (see below). This will make it much easier for others to integrate their knowledge with yours.
- If you must design a new ontology, consult widely with other potential users. This will make it most useful and most likely to be adopted.
- Follow naming conventions. For example, call a class by the singular name of its members. For example, call a class "Resort" not "Resorts". Resist the temptation to call it "ResortConcept" (thinking it is only the concept of a resort, not a resort; see the box on page 575). When naming classes and properties, think about how they will be used. It sounds better to say that "$r1$ is of type Resort" than "$r1$ is of type Resorts", which is better than "$r1$ is of type ResortConcept".
- As a last option, specify the matching between ontologies. Sometimes ontology matching has to be done when ontologies are developed independently. It is best if matching can be avoided; it makes knowledge using the ontologies much more complicated because there are multiple ways to say the same thing.

OWL, when written in Turtle, is much easier to read than when using XML. However, OWL is at a lower level than most people will want to specify or read. It is designed to be a machine-readable specification. There are many editors that let you edit OWL representation. One example is Protégé (http://protege.stanford.edu/). An ontology editor should support the following:

- It should provide a way for people to input ontologies at the level of abstraction that makes the most sense.
- Given a concept a user wants to use, an ontology editor should facilitate finding the terminology for that concept or determining that there is no corresponding term.
- It should be straightforward for someone to determine the meaning of a term.
- It should be as easy as possible to check that the ontology is correct (i.e., matches the user's intended interpretation for the terms).
- It should create an ontology that others can use. This means that it should use a standardized language as much as possible.

13.3.2 Top-Level Ontologies

Example 13.16 defines a domain ontology designed to be used by people who want to write a knowledge base that refers to apartment buildings. Each domain ontology implicitly or explicitly assumes a higher-level ontology that it can fit into. There is interest in building a coherent top-level ontology to which

other ontologies can refer and into which they can fit. Fitting the domain ontologies into a higher-level ontology should make it easier to allow them to interoperate.

One such ontology is **BFO**, the **Basic Formal Ontology**. The categories of BFO are given in Figure 13.7 (on the next page).

At the top is **entity**. OWL calls the top of the hierarchy **thing**. Essentially, everything is an entity.

Entities are either **continuants** or **occurrents**. A continuant is something existing at an instant in time, such as a person, a country, a smile, the smell of a flower, or an email. Continuants maintain their identity though time. An occurrent is something that has temporal parts such as a life, smiling, the opening of a flower, and sending an email. One way to think about the difference is to consider the entity's parts: a finger is part of a person, but is not part of a life; infancy is part of a life, but is not part of a person. Continuants participate in occurrents. Processes that last through time and events that occur at an instant

Classes and Concepts

It is tempting to call the classes **concepts**, because symbols represent concepts: mappings from the internal representation into the object or relations that the symbols represent.

For example, it may be tempting to call the class of unicorns "unicornConcept" because there are no unicorns, only the concept of a unicorn. However, unicorns and the concept of unicorns are very different; one is an animal and one is a subclass of knowledge. A unicorn has four legs and a horn coming out of its head. The concept of a unicorn does not have legs or horns. You would be very surprised if a unicorn appeared in a class about ontologies, but you should not be surprised if the concept of a unicorn appeared. There are no instances of unicorns, but there are many instances of the concept of a unicorn. If you mean a unicorn, you should use the term "unicorn". If you mean the concept of a unicorn, you should use "concept of a unicorn". You should not say that a unicorn concept has four legs, because instances of knowledge do not have legs; only animals (and furniture) have legs.

As another example, consider a tectonic plate, which is part of the Earth's crust. The plates are millions of years old. The concept of a plate is less than a hundred years old. Someone can have the concept of a tectonic plate in their head, but they cannot have a tectonic plate in their head. It should be very clear that a tectonic plate and the concept of a tectonic plate are very different things, with very different properties. You should not use "concept of a tectonic plate" when you mean "tectonic plate" or vice versa.

Calling objects concepts is a common error in building ontologies. Although you are free to call things by whatever name you want, it is only useful for knowledge sharing if other people adopt your ontology. They will not adopt it if it does not make sense to them.

- entity
  - continuant
    - independent continuant
      - site
      - object aggregate
      - object
      - fiat part of object
      - boundary of object
    - dependent continuant
      - realizable entity
        - function
        - role
        - disposition
      - quality
    - spatial region
      - volume
      - surface
      - line
      - point
  - occurrent
    - temporal region
      - connected temporal region
        - temporal interval
        - temporal instant
      - scattered temporal region
    - spatio-temporal region
      - connected spatio-temporal region
        - spatio-temporal interval
        - spatio-temporal instant
      - scattered spatio-temporal region
    - processual entity
      - process
      - process aggregate
      - processual context
      - fiat part of process
      - boundary of process

Figure 13.7: Categories of Basic Formal Ontology (BFO). The indentation shows the subclass relationship. Each category is an immediate subclass of the lowest category above it that is less indented.

in time are also both occurrents.

A continuant is an **independent continuant**, a **dependent continuant**, or a **spatial region**. An independent continuant is an entity that can exist by itself or is part of another entity. For example, a person, a face, a pen, the surface of an apple, the equator, a country, and the atmosphere are independent continuants. A dependent continuant only exists by virtue of another entity and is not a part of that entity. For example, a smile, the smell of a flower, or the ability to laugh can only exist in relation to another object. A spatial region is a region in space, for example, the space occupied by a doughnut now, the boundary of a county, or the point in a landscape that has the best view.

An **independent continuant** can further be subdivided into the following:

- A **site** is a shape that is defined by some other continuants. For example, the hole in a donut, a city, someone's mouth, or a room are all sites. Whereas sites may be at a spatial region at every instance, they move with the object that contains them.
- An **object aggregate** is made up of other objects, such as a flock of sheep, a football team, or a heap of sand.
- An **object** is a self-connected entity that maintains its identity through time even if it gains or loses parts (e.g., a person who loses some hair, a belief, or even a leg, is still the same person). Common objects are cups, people, emails, the theory of relativity, or the knowledge of how to tie shoelaces.
- A **fiat part of an object** is part of an object that does not have clear boundaries, such as the dangerous part of a city, a tissue sample, or the secluded part of a beach.
- The **boundary of an object** is a lower-dimensional part of some continuant, for example the surface of the Earth, or a cell boundary.

A **spatial region** is three-dimensional (a volume), two-dimensional (a surface), one-dimensional (a line), or zero-dimensional (a point). These are parts of space that do not depend on other objects to give them identity. They remain static, as opposed to sites and boundaries that move with the objects that define them.

A **dependent continuant** is a quality or a realizable entity. A **quality** is something that all objects of a particular type have for all of the time they exist – for example, the mass of a bag of sugar, the shape of a hand, the fragility of a cup, the beauty of a view, the brightness of a light, and the smell of the ocean. Although these can change, the bag of sugar always has a mass and the hand always has a shape. This is contrasted with a **realizable entity**, where the value does not need to exist and the existence can change though time. A realizable entity is one of the following:

- A **function** specifies the purpose of a object. For example, the function of a cup may be to hold coffee; the function of the heart is to pump blood.
- A **role** specifies a goal that is not essential to the object's design but can be carried out. Examples of roles include the role of being a judge, the role of delivering coffee, or the role of a desk to support a computer monitor.

- A **disposition** is something that can happen to an object, for example, the disposition of a cup to break if dropped, the disposition of vegetables to rot if not refrigerated, and the disposition of matches to light if they are not wet.

The other major category of entities is the occurrent. An **occurrent** is any of the following:

- A **temporal region** is a region of time. A temporal region is either connected (if two points are in the region, so is every point in between) or scattered. Connected temporal regions are either intervals or instants (time points). Tuesday, March 1, 2011, is a temporal interval; 3:31 p.m. on that day is a temporal point. Tuesdays from 3:00 to 4:00 is a scattered temporal region.
- A **spatio-temporal region** is a region of multidimensional space-time. Spatio-temporal regions are either scattered or connected. Some examples of spatio-temporal regions are the space occupied by a human life, the border between Canada and the United States in 1812, and the region occupied by the development of a cancer tumor.
- A **processual entity** is something that occurs or happens, has temporal parts (as well as, perhaps, spatial parts), and depends on a continuant. For example, Joe's life has parts such as infancy, childhood, adolescence, and adulthood and involves a continuant, Joe. A processual entity is any of the following:
  - A **process** is something that happens over time and has distinct ends, such as a life, a holiday, or a diagnostic session.
  - A **process aggregate** is a collection of processes such as the playing of the individuals in a band, or the flying of a set of planes in a day.
  - A **fiat part of process** is part of a process having no distinct ends, such as the most interesting part of a holiday, or the most serious part of an operation.
  - A **processual context** is the setting for some other occurrent, for example, relaxation as the setting for rejuvenation, or a surgery as a setting for an infection.
  - A **boundary of a process** is the instantaneous temporal boundary of a process, such as when a robot starts to clean up the lab, or a birth.

The claim is that this is a useful categorization on which to base other ontologies. Making it explicit how domain ontologies fit into an upper-level ontology promises to facilitate the integration of these ontologies. The integration of ontologies is necessary to allow applications to refer to multiple knowledge bases, each of which may each use different ontologies.

Designing a top-level ontology is difficult. It probably will not satisfy everyone who must use one. There always seem to be some problematic cases. In particular, boundary cases are often not well specified. However, using a standard top-level ontology should help in connecting ontologies together.