A Brief Introduction to the OpenCyc Ontology

Kenneth D. Forbus, Northwestern University

Version of 1/7/19

The OpenCyc ontology is a very useful formalization of foundational aspects of knowledge representation. Concepts from the OpenCyc ontology have found their way into other KRR systems, for example, the SUMO upper ontology was inspired by Cyc. An understanding of this ontology is useful for anyone who wants to understand knowledge representation more deeply, and because systems based on it are out in the field, it is relatively easy to experiment with. Northwestern's NextKB, for example, which is used in the CogSketch sketch understanding system and the Case Mapper analogy tool, incorporates the OpenCyc ontology.

Conventions about Assertions

We use Lisp-style syntax for assertions, for example, in

```
(likes-Generic Forbus Lisp)
```

the predicate is `likes-Generic`, and the arguments are `Forbus` and `Lisp`. Case sensitivity is assumed for both predicates and entities, with camel case commonly used (e.g. `relationAllExistsMany` is one of the many higher-order predicates found in OpenCyc that make it highly expressive).

Kinds of things in the Ontology

There are three basic concepts you need to understand to get started:

- *Collections* provide an engineering approximation to concepts. There are structural relationships that provide inheritance relationships between concepts, and a rich vocabulary of them to begin with.
- *Predicates* provide relations and functions to make statements with. There are structural relationships that specify inheritance and type information that supports quick constraint checking, and a rich vocabulary to draw upon.
- *Microtheories* provide a notion of context, enabling a knowledge base to embody multiple mutually incompatible perspectives, alternate hypotheses, and fictional worlds, as well as support fine-grained control over reasoning. All of the knowledge in the OpenCyc ontology is so contextualized.

The next three sections expand on these basic concepts.

Collections

Roughly, a collection can be viewed as an intentionally specified set. That is, the collection `Cat` includes all cats that ever have been, every will be, and ever could be. Almost no human concepts admit to a precise, analytic definition (e.g. "A triangle is a 2D polygon that has exactly three straight sides."). Like words in human language, meanings are constrained by the set of statements that are made with them. This can include references to percepts, e.g. an object ID in a vision system appearing in statements is a means of perceptual grounding that is available for many kinds of everyday objects in today's AI systems.

To specify that something is an instance of a collection, the predicate `isa` is used, e.g.

```
(isa NeroTheCat Cat)
```

indicates that the entity referred to by `NeroTheCat` is an instance of the collection `Cat`. The second argument to `isa` must always be a collection. Collections are also entities, which enables the meaning of cat to be further constrained by statements like

```
(isa Cat BiologicalSpecies)
(isa Cat DomesticatedAnimalType)
```

The concept of collection is represented explicitly in the ontology by `Collection`, which is why collections should not be viewed as sets, or Russell's Paradox would become relevant.

Since the 1970s, inheritance has been viewed as a fundamental property of human conceptual structure. It is convenient to store common properties like needing to eat or sleep under higher-level concepts, like `Animal`, rather than doing so for each animal type. It also supports learning, since once we hear about a new kind of animal, we may safely assume that instances of it, too, will need to eat and sleep. Such inheritance is expressed in OpenCyc via the `genls` relation, e.g.

```
(genls Cat Animal)
```

Indicates that `Cat` inherits from `Animal`. Intuitively, one can think of `genls` as like subset, and `isa` as like element of, in set theory terms. Inheritance in OpenCyc is monotonic, i.e. every statement true about members of `Animal` must also hold for `Cat`. (This is a wise choice because making correct inheritance algorithms that handle exceptions is much harder than it looks, and information about defaults can instead be put into relationships that can be explicitly reasoned about.)

The most general collection is `Thing`. In addition to collections specified directly, there are a set of logical functions in the ontology which enable the flexible specification of novel collections, by combining other collections, e.g. the left elbows of organisms.

Predicates

There are two kinds of predicates.

- *Relations* are used to make statements. `likes-Generic` and `isa` are both examples of relations. Statements can be true or false or unknown, and have various other epistemic properties.
- *Functions* are used to denote entities, e.g. `ArmyFn` is a function that can be used to refer to specific entities, e.g. `(ArmyFn NewZealand)` refers to the New Zealand Army. Non-atomic terms (aka NATs) are not statements, but indirect ways of referring to entities, perhaps very abstract entities.

All functions are instances of the collection `Function-Denotational`. All relations are an instance of the collection `Predicate`[1]. Both Relations and Functions have a set of structural predicates that

[1] In keeping with practice in mathematics, rather than logic, the collection `Relation` subsumes both `Predicate` and `Function-Denotational`.

provide type information about them. The relation `arity` indicates the number of arguments that a relation or function takes, i.e.

```
(arity likes-Generic 2)
```

There are multiple relations that provide type constraints on the arguments of a relation or function. Continuing with `likes-Generic`,

```
(arg1Isa likes-Generic Agent-Generic)
(arg2Isa likes-Generic Thing)
```

These say that the first argument must be an instance of the collection `Agent-Generic`, and the second argument must be an instance of the collection `Thing`. `Agent-Generic` is quite general, including people, but also fictional characters and some kinds of computer software. Thing, of course, is even more general. What about providing more specific relationships, when we know more about the arguments? Just like for collections, there is a separate set of inheritance relations for predicates:

```
(genlPreds likesAsFriend likes-Generic)
```

Indicates that `likesAsFriend` is a more specific relationship than `likes-Generic`. This means that if `likesAsFriend` holds between two entities, then `likes-Generic` does as well. The argument constraints on `likesAsFriend` must be subsets of the constraints on `likes-Generic`, here both arguments must be instances of `PerceptualAgent-Embodied`. The term `specPred` is used for the inverse relationship.

There are additional relations that specify constraints on what is denoted by functions. For example,

```
(resultIsa ArmyFn Army-BranchOfService)
```

Indicates that the thing denoted by any non-atomic term with `ArmyFn` as its functor must be an instance of `Army-BranchOfService`.

Microtheories

The world is a complicated place. It is useful to factor knowledge when reasoning. When playing a strategy game with a type of unit called an explorer, when someone says “explorer”, they are more likely to be referring to that than to a Ford Explorer automobile, for example. Being able to store knowledge in partitions which can be flexibly assembled as needed can help speed up reasoning by eliminating the need to search through irrelevant knowledge. Moreover, we often have to consider hypothetical worlds, as when we read a story, or compare and contrast conflicting models, as when we generate and compare alternatives during diagnosis and/or design. The OpenCyc ontology uses microtheories provide an elegant mechanism for handling contexts.

A microtheory can be viewed as a container that holds facts. Microtheories are used to represent specific concerns, e.g. `HumanActivitiesMt` contains over 2,000 facts about everyday human actions. By convention, microtheory names are capitalized and end in Mt. Microtheories, like collections and predicates, have an inheritance structure as well, specified by the relationship `genlMt`, e.g.

```
(genlMt FolkPsychologyMt HumanActivitiesMt)
```

Indicates that `FolkPsychologyMt` inherits from `HumanActivitiesMt`, i.e. every fact that holds in `HumanActivitesMt` will also hold in `FolkPsychologyMt`.

There are four special microtheories defined in the ontology:

- `BaseKB` is the most general microtheory, and fact that are true in it are true in every context.
- `UniversalVocabularyMt`, like BaseKB, contains facts that are true in any context, but these are structural facts, like arity and argument constraints.
- `EverythingPSC` is the union of all microtheory contents. It is of course strongly inconsistent, but sometimes it is useful for retrieving facts to identify useful contexts for subsequent reasoning.
- `NothingPSC` is the empty microtheory, which does not even include BaseKB or `UniversalVocabularyMt`.

All reasoning in systems built on this ontology is performed with respect to some microtheory, so that the facts in it, and all of the microtheories that it inherits from, are available for use, with conclusions being drawn stored in the microtheory used to supply context for the reasoning. This is called the *logical environment* for the computation. Reasoning systems will often construct temporary microtheories to serve as scratchpads and add in additional microtheories that contain standard assumptions for a type of analysis, background data, etc. to create an appropriate logical environment for tackling a particular problem.

We note that microtheories are first-class entities in the ontology. For example, microtheories are used as arguments in a number of predicates to express an agent's beliefs (e.g. `desires-Microtheory`'s second argument is a microtheory, which enables an arbitrarily complex set of statements to be used to express a situation that an agent desires).

Most facts in the ontology can be restricted via microtheories, but not all. For example, `genlMt` statements are held to be global, independent of any microtheory, since having them depend on what microtheory they are in could easily lead to some terrible logical tangles. Similarly, `arity` statements are also global. On the other hand, `genls` statements are contextualized by microtheory, e.g. a carbon chauvinist might not grant personhood to an intelligent robot, whereas other more liberal souls might.

Some Useful Aspects of the OpenCyc Ontology

A whole book could (and should) be written about the OpenCyc ontology. But for this brief introduction, we touch on only two interesting aspects.

Disjointness Reasoning

Mutual exclusivity is a powerful constraint for reasoning. OpenCyc has several relations that support this. The simplest is `disjointWith`, which indicates that two collections are disjoint, i.e. an instance of one cannot be an instance of the other. Another useful concept is the idea of `SiblingDisjointCollectionType`, i.e. instances of this collection are themselves collections, whose sibling subcollections must be disjoint. For example, using the OpenCyc ontology the disjointness of dogs and cats can be inferred, rather than having to be stated, because those collections are specializations of genus descriptions that are themselves an instance of `BiologicalTaxonType`, which is a member of `SiblingDisjointCollectionType`.

Rule Macro Predicates

When building large-scale knowledge bases, it makes sense that there will be patterns that appear over and over again, and that encapsulating these patterns, like macros in a programming language, can support both more concise expression of knowledge and more efficient implementation. Consider the idea that planets have atmospheres that completely cover them. In the OpenCyc ontology, this can be concisely expressed as

```
(relationAllExists covers-Generic Atmosphere Planet)
```

Which, when expanded out into a more traditional axiomatic form would read

```
(forAll (?p) (implies (isa ?p Planet)
          (Exists (?a) (and (isa ?a Atmosphere)
                            (covers-Generic ?a ?p)))))
```

Learning More

The OpenCyc ontology is one of the richest resources available in artificial intelligence, representing decades of work on a wide range of knowledge representation and reasoning problems. While Cycorp no longer supports OpenCyc, there are still multiple versions available on various web sites. Their browser and reasoning engine are excellent. If you want to experiment with massive amounts of knowledge and a very powerful reasoning engine, I recommend contacting Cycorp to get a ResearchCyc or full Cyc license, depending on what you are doing.

If you are looking to build on open resources, and are using Northwestern's NextKB, which incorporates the OpenCyc ontology, there are two straightforward ways to be able to browse it and perform simple queries:

- Download our CogSketch sketch understanding software (Windows only), which uses NextKB. The browsing tools are documented in the user manual.
- Download our Case Mapper software, a tool aimed at helping cognitive scientists do simulation experiments with analogical matching and retrieval (Windows, Linux, and Macs). Case Mapper currently uses an older open-license KB, but you can download the NextKB FIRE build from our web site and open it up from Case Mapper.

If you have questions, comments, or suggestions, please contact us at nextkb-feedback@cs.northwestern.edu.