From Knowledge Represented in Frame-Based Languages to Declarative Representation and Reasoning via ASP

Chitta Baral and Shanshan Liang
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University, Tempe, AZ

Abstract

In this paper we encode some of the reasoning methods used in frame based knowledge representation languages in answer set programming (ASP). In particular, we show how "cloning" and "unification" in frame based systems can be encoded in ASP. We then show how some of the types of queries with respect to a biological knowledge base can be encoded using our methodology. We also provide insight on how the reasoning can be done more efficiently when dealing with a huge knowledge base.

Introduction

The broader goal of our proposed research is to be able to answer various kinds of questions with respect to knowledge bases constructed by domain experts in particular domains. Towards this effort we noticed that a large body of such knowledge bases have been developed using *frame based representations* (Fikes and Kehler 1985)[1]. Examples of such knowledge bases include AURA (Chaudhri et al. 2009), EcoCyc (Karp et al. 2002) and RiboWeb (Altman et al. 1999). A University of Texas site[2] links to a large number of these knowledge bases.

While the terms such as "object" and "oriented" are reported to be first mentioned in late 1950s and early 1960's at MIT, and the terms "object" and "instance" were used by Sutherland in 1960-61, the origin of frame based knowledge representation is often attributed to Minsky's 1975 work (Minsky 1975) . In frame based KR, knowledge is organized in chunks referred to as frames. Frames have slots which can be filled with values or other frames. There are hierarchies of frames and one frame may inherit the properties of another frame based on this hierarchy. Most frame based formalisms include some procedurally expressed knowledge, such as specification regarding how frames should be used and how inheritance should be addressed. Thus, semantics of frames are often specified operationally. Because of inbuilt features, such as inheritance, frames allow concise representation of knowledge. So, despite the statement in Hayes (Hayes 1979) that said "most of frames is just a new syntax for parts of first order logic", to capture some of the semantics of frames, such as inheritance, one needs additional axiomatization. This is also true with respect to Descriptions logics (DLs) (Baader et al. 2003), which although capture many of the declarative aspects of frames do not automatically capture the procedural, operational and inbuilt aspects. They (DLs) do have much richer mechanism to define new classes. There have also been many proposals that combine frame based knowledge representation aspects with rule based knowledge representation aspects. Example of such formalisms are DLV+ (Ricca and Leone 2007) and FAS (Alviano et al. 2008).

Some of the operational and inbuilt aspects of frames have been logically formalized. The most common aspect being the formalization of inheritance, logical formalization of which has been given by many (Touretzky 1986; Horty 1994). Of particular note is the work by Brewka (Brewka 1987) that discuses the logic of inheritance in frame systems.

However, these formalisms also do not address some of the procedural reasoning mechanisms in many frame based systems. In particular, none of them address the issue of "cloning" and "unification" that we discuss in the following paragraphs.

In this paper, while working towards our goal of answering various kinds of questions with respect to knowledge bases constructed by domain experts in particular domains, we first formalize the notion of cloning and unification that were earlier presented in an operational manner, and not precisely enough, in the frame based systems KM (Clark, Porter, and Works 2004)[3] and AURA (Chaudhri et al. 2009) [4]. However, these notions make it easier for multiple domain experts to work in parallel in representing knowledge using frames. They also make the representation compact. Some basic ideas and motivation behind them is as follows. Suppose a group of domain experts were to describe the knowledge about cells in a knowledge base as they go through the chapters in a book about cells. In a frame based approach each may consider describing a prototype cell with various slots (for example, parts-of) and values. Many of those values, such as the various parts of cells (for example, nucleus,

[1] See also http://www.cs.man.ac.uk/∼stevensr/onto/node14.html.

[2] http://www.cs.utexas.edu/∼mfkb/related.html

[3] KM has been adopted by various projects, notably Project Halo (Gunning and et 2010).

[4] AURA is also developed under Project Halo.

mitochondria, etc.), may themselves be then described using prototypes. Now depending upon the particular focus in a chapter multiple domain experts may write multiple prototypes for the same class, each focusing on a different facet of knowledge about cells, with possible overlaps and conflicts. The reasoning mechanism unifies these prototypes to obtain a comprehensive view of the class. In addition the knowledge base has class-instance membership facts and class-superclass facts. Enhancement of an instance using "unification" then refers to integrating it with the other instances that it can clone from taking into account the cloning information as well as inheritance. Thus while the idea of inheritance and prototypes make it easier to encode knowledge concisely, and in parallel by multiple domain experts, the reasoning mechanism must account for inheritance, cloning and unification. In KM and AURA procedural modules are written for that. In this paper we first give a formal definition of unification using cloning and inheritance and then give an Answer Set Programming (ASP) based implementation of that. We then show how ASP can be used to answer several different kinds of questions with respect to such a knowledge base. In the process, we point out ways to make the ASP based computation efficient and mention the impact of that with respect to a large biological knowledge base.

Background

Answer Set Programs

We adopted Answer Set Programming (ASP) (Gelfond and Lifschitz 1988) as our knowledge representation language for the following reasons: (i) it is non-monotonic, has a simpler syntax but yet is expressive; (ii) it has a strong theoretical foundation with many building-block results (Baral 2003); and (iii) it has several efficient solvers (Gebser et al. 2008; Niemelä and Simons 1997; Leone et al. 2006). An ASP program is a collection of rules of the form:

$$a \leftarrow a_1, \ldots, a_m, \mathit{not}\ b_1, \ldots, \mathit{not}\ b_n$$

where a, $a_1, ..., a_m$ and $b_1, ..., b_n$ are atoms. The rule reads as "a is true if $a_1 ... a_m$ are all known to be true and $b_1 ... b_n$ can be assumed to be false". The semantics of answer set programs are defined using answer sets (earlier called stable models). An entailment relation ($\models$) with respect to answer set programs is defined as follows: A program Π entails an atom p iff p is true in all the answer sets of Π.

Frame-based KR&R

In this paper we choose KM as our reference frame-based formalism. The basic representational unit in KM is a frame that contains slots and values, in which the values can be either atomic or refer to frames. Frames are used to provide the structured representation for classes and instance via slots (Fikes and Kehler 1985). In KM classes and instances are represented using the following format:

```
(every <class> has
  (superclasses (<superclass1> ...
       <superclassN>))
  (<slot1> (<expr11> <expr12>...))
  (<slot2> (<expr21> <expr22>...))...)
```

```
(<instance> has
  (instance-of (<class1>...<classn>))
  (<slot1> (<expr11> <expr12>...))
  (<slot2> (<expr21> <expr22>...))...)
```

The slots in the frames normally represent attributes of class/instance, or specify relations between class-class, class-instance, and instance-instance. From the format we can tell that "superclasses" and "instance-of" are two special meaning slots for representing class and instance, respectively. Some of the special meaning slots that are key for representation and reasoning in KM are listed below:

- *Superclasses:* The slot "superclasses" in the $< class >$ representation specifies the superclasses of a class. The superclass information constitute a hierarchy for all the concepts defined in the knowledge base.
- *Prototype-of:* This slot defines prototype(s) of a class. A prototype is a proto-typical instance of a class, which is an instance that serves as a basis of a class. If A is a prototype of class C, then the properties of A is true in all instances of C, unless restricted otherwise via a prototype scope that is specified.
- *Instance-of:* This slot defines instances of a class. An instance can be either a proto-typical or a normal element of a class. If A is an instance of class C, then A automatically inherits all the inheritable slot values expressed in the prototype of C, meaning A gets a copy of all the slot values of Cp, where Cp is a prototype of C.
- *Cloned-From:* If a description of an instance A has instance B in its slot named cloned_from, then all the slot values (aka attributes or properties) of B also become the slot values of A. [5]

A KM knowledge base is accessible via user posed queries, and both the queries and the answers to those queries are normally based on instances. The inference step utilizes deterministic construction rules (Clark, Porter, and Works 2004) to provide answers to queries, and has several specialized reasoning modules to capture inheritance, cloning and unification. Here we illustrate the reasoning process with an example, slightly modified from examples in (Clark, Porter, and Works 2004) :

```
(Vehicle1 has
  (instance-of (Vehicle))
  (prototype-of (Vehicle))
  (has-engine ((a Engine with
      (strength (*Powerful))
      (fuel ((a Gas-type with
        (combustibility (*Hi)))))))))

(every Car has
   (superclasses (Vehicle)))
```

[5] This is the intuitive meaning, however there are cases when A need to unify its values with B's values, which is explained in the next example.

```
(Car1 has
   (instance-of (Car))
   (prototype-of (Car))
   (has-engine ((a Engine with
      (size (*Average))
      (fuel ((a Gas-type with
        (type (*Unleaded))))))))))
```

There are two instances: Vehicle1, Car1; and two classes: Vehicle, Car. The "prototype-of" slots specifies that Vehicle1 is a prototype of Vehicle, and Car1 is a prototype of Car. The "superclasses" slot specifies that Vehicle is a superclass of Car, which means Car will inherit from Vehicle, which in turn means Car1 will clone from Vehicle1. [6] So Car1 will obtain all the values of Vehicle1, ideally. However Car1 already has an engine; if it directly clones another engine from Vehicle1, then Car1 will have two engines, which is not the desired result that makes intuitive sense. The more intuitive way is for Car1 to keep its engine, but clone all the other attributes(values) from Vehicle1's engine. In this case, Car1 still has one engine, but this engine will have the new value "powerful", illustrated as follows:

```
(Car1 has
   (has-engine (a Engine with
      (strength (*Powerful))
      (size (*Average))
      (fuel (?)))
```

For the slot "fuel", we can see that both Car1's engine and Vehicle's engine has "fuel", so again we don't directly clone a second "fuel" for Car1's engine, instead we let this fuel obtain attributes from Vehicle1's engine's fuel, and we will have the following result for Car1:

```
(Car1 has
   (has-engine (a Engine with
      (strength (*Powerful))
      (size (*Average))
      (fuel (((a Gas-type with
            (type (*Unleaded))
            (combustibility (*Hi)))))))
```

This whole process is called "cloning and unification", which is an important aspect of reasoning in KM. KM defined it in a procedural way. As performing a full unification is computationally expensive, only lazy unification is allowed in KM. This means that instead of performing unification for "fuel", a symbol "&" is used to denote where the unification need to be performed, so the fuel for Car1 will look like:

```
(fuel (((a Gas-type with
            (type (*Unleaded)))
       & (a Gas-type with
            (combustibility (*Hi)))))))
```

In this paper, we defined the "clone and unify" in a declarative way, and allow the computation of full unification. In the next section we formalize the notion of a knowledge base (KB), give the necessary reasoning modules for performing "clone and unify", and give detailed explanation of both.

[6] The rule is that when a class inherits from its superclasses, all of the classes' instances will clone from the superclasses' prototypes.

Formal Definitions about a Frame Based Knowledge Base

As we mentioned earlier, some of the aspects (such as "unification") in frame based systems were not precisely defined earlier. Since one of our main goal is to develop an ASP implementation of reasoning with knowledge represented in a frame based format and prove its correctness, in this section, we first formally define various aspects of a frame based representation. Our definitions are based on various frame based systems and we made some simplifications to focus on the main aspects, namely, cloning and unification. We motivate some aspects of the definitions with examples and present the direct translation of the knowledge represented in a frame based format to ASP. (In the next section we give the ASP rules that encode inheritance, cloning and unification.)

Frame-based KR focuses on two aspects: representing classes and objects (or instances). For each class, the class properties and hierarchy information are the key aspects. Class properties will be inherited by all the objects that belong to it. Some frame-based KR formalisms provide special mechanisms for this type of inheritance: creating a prototype of class A [7], namely A_p, and letting all instances of A clone from A_p. As prototypes are special cases of objects, we define them in a similar fashion, and we only define the hierarchy information for classes themselves. We now start with the definition of a Class Hierarchy Graph.

Definition 1: *A Class Hierarchy Graph (G_{classH}) is a directed acyclic graph whose vertices are classes and whose edges encode class-superclass relationship; i.e. if (C1, C2) is a directed edge then C2 is a superclass of C1.*

The ASP encoding of G_{classH} is a collection of facts of the form: $has(X, superclass, Y)$, where X and Y are vertices in the graph, and there is an edge from X to Y.

Definition 2: *An Object Graph $G_{obj}(O)$ (also denoted simply by $G(O)$) that describes an object O consists of a core object graph which is a directed graph whose vertices are instances and whose edges are labelled by two types of labels: non-inheritable and inheritable labels. For each vertex in the core object graph, there is an edge in the object graph from that vertex to a class; this edge is labelled with $instance\_of$ and specifies which class this instance belongs to. Object Graphs are acyclic with respect to the inheritable labeled edges.*

The ASP encoding of $G_{obj}(O)$ is a collection of all the slot values of X in the form $has(X, S, V)$, in which the slot S is the label of the edge connecting X to V.

An edge (X, Y) labelled by R means X R Y. For example, an edge (X,Y) labelled by "instance-of" means that X is an instance of Y.

[7] A class can also have multiple prototypes.

Non-inheritable labels, such as {instance-of, prototype-of, cloned-from, etc}, tend to describe the "relations" rather than the "properties". These labels are not to be transferred from one object graph to another via cloning or inheritance.

Inheritable labels, such as {has-part, agent, etc} are the ones that describe the properties of an instance, and may be inherited or cloned by other instances.

The difference between an object graph describing a normal instance versus a prototype is based on whether the graph contains an edge labelled with $prototype\_of$.

Another important label is $cloned\_from$, which means that the current object obj_c clones from another object. The cloning process involves all the object graphs that obj_c transitively clones from, and the outcome will be a new object graph for obj_c. This process is defined latter in this section.

Definition 3: *A Knowledge Base is a collection of object graphs, a class hierarchy graph, and a specification of inheritable/non-inheritable labels (slot names).*

Both object graphs and class hierarchy graphs are encoded using ASP facts of the form $has(X, S, V)$, in which X can be either an object or a class, S is a slot name, and V is the value for slot S. The inheritable slots are encoded as $inheritable(S)$.

We already identified several special meaning slots, namely $instance\_of$, $prototype\_of$, $cloned\_from$, and $superclass$. We will illustrate how these slots can be used for defining other concepts that are useful for reasoning about a KB.

Definition 4: *Let KB be a knowledge base. Let x be an instance and y_1 ... y_{k+1} be classes. We say that x is a tc-instance-of y_{k+1} if*

(i) there is an object graph G_obj in KB that has (x, y_{k+1}) as an edge labeled with instance-of, or

(ii) there is an object graph G_obj in KB that has (x, y_1) an edge labeled with instance-of, and in the class hierarchy graph G_{classH} we have: for all $i \in [1, k]$, (y_i, y_{i+1}) is a superclass edge.

As a compact representation, $G_{obj}(X)$ (also denoted as $G(X)$) only contains instance-of edges connecting X to the immediate class(es) that X belongs to. However, X also recursively belongs to all the superclasses of the immediate class(es). The above definition defines a transitive-closure of the classes that an object X belongs to. Similarly, below we define the tc_cloned_from relation which is a transitive closure of the $cloned\_from$ relation.

Definition 5: *Let KB be a knowledge base. Let x_1 ... x_{k+1} be instances. We say that x_1 is tc-cloned-from x_{k+1} if*

(i) $G_1, G_2, .., G_k$ are object graphs in the KB, and for all $i \in [1, k]$, (x_i, x_{i+1}) is a cloned-from edge in G_i, or

(ii) there exists a class y_{k+1} such that x_1 is a tc-instance-of y_{k+1}, and object graph G_{k+1} contains (x_{k+1}, y_{k+1}) as an edge labeled with prototype-of.

For $G(X)$, after obtaining the set of instances $\{Y_1, ..., Y_n\}$ that X clones from, the next step will be for X to obtain all the properties from those instances. However, the properties of $\{Y_1, ..., Y_n\}$ should not be directly added to X's object graph, but need to go through a merging process which is referred to as unification (Clark, Porter, and Works 2004). Definitions 6 and 7 define the unification process, and the necessary relations needed in unification.

Definition 6: *Let x and y be instances, the sets $\{xc_1...xc_m\}$ and $\{yc_1...yc_n\}$ be all the classes that x and y are tc-instances-of, respectively. We say that x subsumes y if $\{yc_1...yc_n\} \subseteq \{xc_1...xc_m\}$, $m \geq n$.*

Definition 7: *Given an object graph $G(X)$ and a KB, the cloned and unified enhancement of $G(X)$ w.r.t the KB, denoted by $G^*_{KB}(X)$, has the following properties. Let $\{G(X_1), G(X_2), ...G(X_k)\}$ be the set of object graphs that $G(X)$ clones from.*

1. *Base case: $G^*_{KB}(X)$ has all the inheritable edges and all the vertices linked by those edges from $G(X)$. The root node for $G^*_{KB}(X)$ is still X, which is at level zero.*
2. *For each node V in $G^*_{KB}(X)$ at level N, an additional node V' is a child of V if V' satisfies one of the following conditions:*
   - *(V_i, V') is an inheritable edge in object graph $G(X_i)$ for some $i \in [1, k]$, V subsumes V_i (which is at level N), and both of the common conditions are satisfied.*
   - *V tc_cloned_from V_i, (V_i, V') is an inheritable edge in the KB, and both of the common conditions given below are satisfied.*
   - *V_j is a node at level N in the object graph $G(X_j)$ for some $j \in [1, k]$, V subsumes V_j (which is at level N), V_j tc_cloned_from V_i, (V_i, V') is an inheritable edge in the KB, and both of the common conditions below are satisfied.*

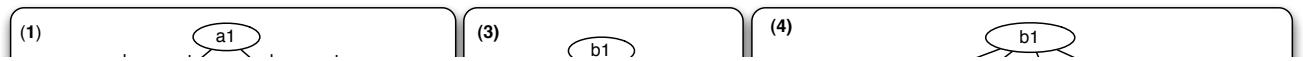
* *Common conditions:*

- *V' is not subsumed by any other nodes that are at level $N + 1$ in any of the object graphs beside $G(X_i)$, if $G(X_i)$ has (V_i, V') as an inheritable edge.*
- *The label of the added edge (V, V') in $G^*_{KB}(X)$ is the same as of the edge (V'', V') if we obtain V' via V''.*

We illustrate the above definitions using Figure 1 and Figure 2. In Figure 2, a, b, and c represent three classes, and $a1$, $b1$, $c1$ are the corresponding instances/prototypes. In Figure 1, subgraphs (1)(3)(5) represent the object graphs $G(a1)$, $G(b1)$ and $G(c1)$, and subgraphs (2)(4)(6) represent the cloned and unified enhancements $G^*_{KB}(a1)$, $G^*_{KB}(b1)$, and $G^*_{KB}(c1)$. Here the KB consist of:

1. the class hierarchy graph which specifies that a is a superclass of b, and b is a superclass of c.
2. the object graphs for a1, b1, and c1, which are the prototypes for class a, b, and c respectively. [8]

[8] In Figure 2, the relations "a1 instance-of a" and "a1 prototype-of a" are also part of the object graph for a1. We extract and put it in a different graph for a simpler view.



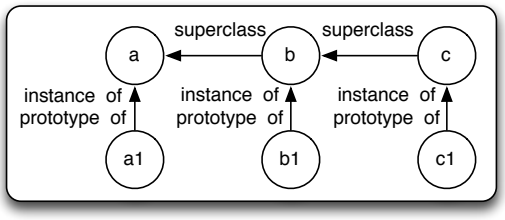


Figure 2: *The relations among a1,b1, and c1.*

3. the specification that non-inheritable slots consist of $prototype\_of$, $instance\_of$, and $cloned\_from$, and inheritable slots consist of the rest of the slots encoded in the KB.

Obtaining $G^*_{KB}(b1)$

We can conclude that b1 is a tc-instance-of class a using Definition 4, and that b1 tc-cloned-from a1 using Definition 5, hence $G(b1)$ clones from $G(a1)$. The white nodes in $G^*_{KB}(b1)$ are directly from $G(b1)$, and do not keep the non-inheritable edge (body2, cloned_from, head1). The shaded nodes in $G^*_{KB}(b1)$ come from $G(a1)$ when the nodes satisfy the conditions listed in Definition 7. Let's take body2 for example: we can infer that body2 subsumes body1 using Definition 6[9], (body1, big1) is an inheritable edge (with label has-size) in $G(a1)$, and big1 is not subsumed by any node on the same level in $G(b1)$ (see the first common condition in Definition 7), so we can conclude that (body2, big1) will be an edge in $G^*_{KB}(b1)$ with the same label "has-size" (see the second common condition in Definition 7). On the other hand, skin1 from body1 is not present in $G^*_{KB}(b1)$ because skin1 is already subsumed by skin2. The third node "hair1" of body2 is obtained via the "cloned-from" relation of body2 in $G(b1)$.

Obtaining $G^*_{KB}(c1)$

Similarly we can infer that c1 is tc-instance-of class a and b, and c1 tc-cloned-from both a1 and b1, hence $G(c1)$ clones from both $G(b1)$ and $G(a1)$. Let's take body3 for example: we can infer that body3 subsumes both body1 and body2; so ideally body3 will have big1, and neither have skin1 nor skin2. As body3 subsumes body2 and body2 also clones from head1, body3 will have hair1 from head1. If we go deeper, we notice that skin3 subsumes both skin1 and skin2, and in the next level hair3 subsumes hair1. The subsume relation goes vertically and expands along a path in the graph. This provides a hint for the declarative implementation which we'll show in the next section.

Definition 8: *We say that X has value V for an inheritable*

[9]We use the name of the instance to reflect the class that it belongs to, so the relations such as (body1, instance-of, body) is not drawn in the figures.

slot S after cloning & unification with respect to a KB if for some object Y in KB, $G^*_{KB}(Y)$ has an inheritable edge (X, V) with label S.

ASP encoding of frame based reasoning aspects

In this section we present the declarative ASP rules for reasoning over the knowledge base. From the previous section we noticed several properties of the knowledge base. In particular, the representation of object is not self-contained. An object graph describing an object X only records the most specific properties of X, and will let X inherit from the superclasses, or clone from other instances to obtain the general properties. One of the fundamental reasoning task for the KB is to retrieve the full properties for an object, which can be used to answer the most fundamental question: *What is X?*. The *property acquiring* process for an instance has the following steps:

(i) *Obtaining generalizations of the instance:* Each instance can have facts about inheritance in the following form: $has(Instance1, instance\_of, Class1)$, and each class can have facts expressing superclass information: $has(Class1, superclass, Class2)$, which indicates that Class2 is a superclass of Class1. Therefore Instance1 will inherit from Class2 as well. Transitive closure rules are needed to define this process.

(ii) *Obtaining what an instance clones from:* Besides inheritance, an instance can also obtain some properties from other instances via cloning: $has(Instance1, clone\_from, Instance2)$. Similarly, cloning can happen transitively, and we need to define the rules for it.

(iii) *The unification process:* When inheriting or cloning happens, an instance need to acquire information from the other classes/instances. Each slot value of the instance will not just be the one stated in its own frame, but will be merged with all the acquired information. The merging process is referred to as unification. This is a key aspect that has not been formalized in the past literature.

Step I: Obtaining generalizations of an instance

When retrieving information for an instance, the first step is to gather all classes that it belongs to. Here we discuss the rules for obtaining multiple inheriting classes for an instance. The slot "instance-of" directly encodes all the classes (immediate class) to which an instance belongs to, and the instance also recursively belongs to the immediate classes' superclasses.

The rules $i1$ and $i2$ encodes $instance\_of(X, Y)$ which means class Y is an immediate class of instance X.

```
i1: instance_of(X,Y) :-
            has(X,instance_of,Y).
i2: instance_of(X,Y) :-
            has(X,clean_instance_of,Y).
```

The rules $i3$ and $i4$ encodes $tc\_instance\_of(X, Y)$ which means instance X transitively belongs to class Y. Rule $i3$ uses the immediate class as the base case. Rule $i4$ means that if X is a $tc\_instance\_of\ M$, and Y is a superclass of M, then X is also a $tc\_instance\_of\ Y$.

```
i3: tc_instance_of(X,Y) :-
              instance_of(X,Y).
i4: tc_instance_of(X,Y) :-
              tc_instance_of(X,M),
              has(M,superclass,Y).
```

Step II: Obtaining what an instance clones from

Cloning is another mechanism that facilitates the reuse of the existing knowledge frames, and avoids repeated encoding of the same set of knowledge entries. *Default cloning* refers to the case where the information encoded for a prototype is transferred to other instances of the same class. *User-defined cloning* refers to the user specified facts of the form $has(X, cloned\_from, Y)$. Both cases need to be taken into account when deciding the set of instances the current instance is cloning from.

The rules $c1$ and $c2$ encodes $cloned\_from(X, Y)$ which means instance X directly clones from instance Y.

```
c1: cloned_from(X,Y) :- X!=Y,
               has(X,cloned_from,Y).
c2: cloned_from(X,Y) :- X!=Y,
               has(X,clone_built_from,Y).
```

The rules $c3$ to $c5$ encode $tc\_cloned\_from(X, Y)$ which means instance X transitively clones from instance Y. Rule $c3$ uses the direct cloning instance as the basic case. Rule $c4$ means X transitively clones from Y if it transitively clones from M, and M directly clones from Y. Rule $c5$ takes into account "prototypes". For any instance X that is a member of class M (immediately or transitively), X clones from class M's prototype(s).

```
c3: tc_cloned_from(X,Y) :-
             cloned_from(X,Y).
c4: tc_cloned_from(X,Y) :-
             tc_cloned_from(X,M),
             cloned_from(M,Y).
c5: tc_cloned_from(X,Y) :- X!=Y,
             tc_instance_of(X,M),
             has(Y,prototype_of,M).
```

Step III: The unification process

We now discuss the information acquiring process for an instance after we know what this instance clones from. Recall that we use the predicate $inheritable(S)$ to specify the list of slots that can be inherited by other instances.

When an instance is trying to obtain slot values (i.e. properties) from other instance(s), as we explained in the previous section, it can not simply append the values on top of its own. To solve this issue, we first define the predicate "may_have" to stand for the slot values that an instance may get via cloning. In the previous example, b1 may have two bodies, two legs, and one head from cloning.

Rule $m1$ means that X may obtain the value V of slot S from Y, when X clones from Y. We uses the predicate "detailed_cloned_from" to include several different types of

cloning, which will be elaborated later. Rule $m2$ means that X may have the properties that it already has.

```
m1: may_have(X,S,V,Y) :- has(Y,S,V),
                detailed_cloned_from(X,Y),
                inheritable(S).
m2: may_have(X,S,V,X) :- has(X,S,V),
                inheritable(S).
```

After having "may_have", we need to specify the cases where an instance shouldn't have a value. We use the predicate "not_to_have" to denote such cases. From the previous example we know that we only want b1 to have body2, and not to copy body1 from a1. The justification is that b1 already has a property of the type "body", so it doesn't need to acquire a new one.

Rule $m3$ defines the predicate "not_to_have": Using it we can conclude that b1 should not have body1 by reasoning with rule $m3$ in the following way. b1 may have body2 from b1, b1 may have body1 from a1, b1 clones from a1, and body2 subsumes body1; so we conclude that b1 not_to_have body1.

```
m3: not_to_have(X,S,V2) :-
              may_have(X,S,V1,Y1),
              may_have(X,S,V2,Y2),
              detailed_cloned_from(Y1,Y2),
              subsume(V1,V2),
              Y1!=Y2, V1!=V2.
```

The above rule $m3$ uses the predicate "subsume" which we explain and define (using ASP) below. The ASP encoding of "subsume" captures the following: $V1$ subsumes $V2$ if V1 is more specific, i.e. $V1$ belongs (immediately or transitively) to all the classes that $V2$ belongs to. Rule $s1$ specifies the condition when $V1$ doesn't subsume $V2$, that is, if there exists a class C that $V2$ belongs to, but not $V1$.

```
s1: not_subsume(V1,V2) :- V1!=V2,
                ins(V1), ins(V2),
                tc_instance_of(V2,C),
                not tc_instance_of(V1,C).
```

Rule $s2$ negates rule $s1$ to obtain the cases when $V1$ subsumes $V2$.

```
s2: subsume(V1,V2) :- V1!=V2,
              ins(V1), ins(V2),
              not not_subsume(V1,V2).
```

In s2 we use the predicate "ins(V)" to specify that V is an instance. The following rule states what value X actually obtains from cloning. X will have all the "may have" values minus the "not to have" values.

```
h1: hasc(X,S,V) :- may_have(X,S,V,Y),
              not not_to_have(X,S,V).
```

Thus, considering a1 and b1 of Figure 1 (1) and (3) we obtain the following properties of b1 after unifying with a1:

```
hasc(b1, has_part, head1).
hasc(b1, has_part, body2).
hasc(b1, has_part, leg1).
hasc(b1, has_part, leg2).
```

This matches with the unified result for b1 in Figure 1 (4).

Subsequent Microcloning

The rules from the previous section leave one problem, how will b1 get body1's information? When b1 keeps body2 and not body1, it also means that body2 is the instance that will clone from body1. We use predicate "microclones" to denote this type of cloning.

Example *Let's take a second look at Figure 1 (1) and (3), we know that because b1 clones from a1 and body2 subsumes body1, so b1 will only keep body2, and body2 will microclone from body1. In a similar fashion, when body2 microclones from body1, because again skin2 subsumes skin1, so body2 only keeps skin2, but will let skin2 microlone from skin1.*

From the example we can tell that microcloning first happens when X doesn't need to keep some value, but will let its property to clone from that value. Microclone also happens recursively and will expand to the deepest level of the object graph.

Rule $r1$ encodes the base case for microclones. If the subsume relation holds as $subsume(V1, V2)$, then value $V1$ needs to acquire information from $V2$. Rule $r2$ encodes the transitive case for microclones.

```
r1: microclones(V1,V2,Y1,Y2) :-
              may_have(X,S,V1,Y1),
              may_have(X,S,V2,Y2),
              tc_cloned_from(Y1,Y2),
              subsume(V1,V2),
              Y1!=Y2, V1!=V2.

r2: microclones(V1,V2,F1,F2) :-
              may_have(X,S,V1,Y1),
              may_have(X,S,V2,Y2),
              microclones(Y1,Y2,F1,F2),
              subsume(V1,V2),
              Y1!=Y2, V1!=V2.
```

We now put together all the cases for clones using the predicate "detailed_cloned_from". As we can see from rules $d1$ to $d3$, this predicate transitively includes " tc_cloned_from" and "microclones". Recall that in the last section both "may_have" and "not_to_have" rely on this "detailed_cloned_from" predicate; that is because when an instance is deciding on what it may need to acquire information from, it needs to take all the types of cloning into account, no matter if it is stated as facts or is derived.

```
d1: detailed_cloned_from(X,Y) :-
              tc_cloned_from(X,Y).
d2: detailed_cloned_from(X,Y) :-
              microclones(X,Y,F1,F2).
d3: detailed_cloned_from(X,Y) :-
              detailed_cloned_from(X,Z),
              tc_cloned_from(Z,Y).
```

Correctness of the ASP encoding

Definition 9 *Given a knowledge base KB, the ASP program Π_{KB} is the answer set program consisting of:*

(i) the following rules: i1 to i4 for instance of, c1 to c5 for clones from, m1 to m3 for may_have, s1 to s2 for subsumes, h1 for hasc (has property after cloning), r1 to r2 for microclones, d1 to d3 for detailed clones and

(ii) all facts of the form $has(X, S, V)$ that are either in the class hierarchy graph, or in the object graphs and all $inheritable(S)$ facts.

Proposition 1 X is an instance of Y w.r.t a KB iff:

$$\Pi_{KB} \models tc\_instance\_of(X, Y)$$

Proposition 2 X is tc_cloned_from Y w.r.t a KB iff:

$$\Pi_{KB} \models tc\_cloned\_from(X, Y)$$

Lemma 1 X subsumes Y w.r.t. a KB iff:

$$\Pi_{KB} \models subsume(X, Y)$$

Theorem 1 X has value V for slot S after cloning and unification in a KB iff:

$$\Pi_{KB} \models hasc(X, S, V)$$

Proof (sketch): The proof is based on dividing Π_{KB} to several strata. The bottom stratum (0) consists of the predicates $inheritable$, ins, has, $instance\_of$, $cloned\_from$, $tc\_instance\_of$ and tc_cloned_from. The next stratum (1) consists of the predicate $not\_subsume$. The next stratum (2) consists of the predicate $subsume$. The next stratum (3) consists of the predicate not_to_have, may_have, $detailed\_clone\_from$, and $microclones$. The next stratum, the top one, consists of the predicate $hasc$. From this stratification, we have that Π_{KB} is a stratified program and thus it has a unique answer set. Now using the stratification we can define each of the predicates, four of which are a sort of transitive closure predicates. They are: $tc\_instance\_of$, tc_cloned_from, $detailed\_clone\_from$, and $microclones$. Facts about them that is true in the answer set can be characterized using the minimum number of iterations needed to get them and this number matches with the shortest inference path with respect to the graphs of KB.

Efficient question answering for specific question patterns

There are various kinds of questions one may pose to a knowledge base. Some of those questions can be more efficiently answered by modifying some of the rules given in the earlier section. This is similar to to the use of magic sets in efficiently answering Datalog queries where Datalog rules are transformed based on query patterns. In the following we show how some rules can be modified for efficiency purposes without sacrificing correctness for the specific kind of queries.

We choose two types of queries for illustration. For questions like *"What is X?"*, we are only interested in the instances/prototypes that are related to class X, and for questions like *"What is the difference between X and Y?"*, we are only interested in instances/prototypes related to X or Y. By adding those instances of interest as a "search guidance" to the previous encodings for instance-of/cloned-from, etc, we can constrain the rules to search a much smaller knowledge base and rule out a large number of irrelevant (to the query) facts to maximize the efficiency. We now explain how to obtain the instances/prototypes of interest, and how they are applied to the previous encodings using the following definitions:

Definition 10: *Let the question posed to the KB be of the form $Q(QID, QType, X_1, X_2...X_n)$, in which QID and $QType$ refers to the question ID and type, and $\{X_1, X_2...X_n\}$ is the list of concepts (classes) mentioned in the question. Let the answer A to the question Q have the form $A(QID, QType, Ans)$, in which Ans is the answer to Q.*

From this notational definition we know that the question *"What is X?"* has the form $Q(QID, what\_is, X)$, and the question *"What is the difference between X and Y?"*, has the form $Q(QID, what\_difference, X, Y)$. QID can be a unique number randomly assigned to the questions.

Ans can take forms of a constant (true/false questions), a variable (compute slot values), or user-defined predicates (for more general type of questions). We investigated several types of questions and what the Ans will be for those types, and will elaborate on this in the following section. In this section we show how some of the rules can be modified for efficiency without sacrificing correctness.

Definition 11: *Given $Q(QID, QType, X_1, X_2...X_n)$, we define the "query-related-class" as "qrc(Q,C)", which contains all the X_i, and all X_i's superclasses, $i \in [1, n]$, and define the query-related-prototype as "qrp(Q,P)", as the combination of:*

(i) all prototypes P_all of the query-related-classes, and

(ii) all the instances/prototypes that P_all clones from.

Rules $qr1$ to $qr4$ are used to infer qrc and qrp according to the definition above. Rule $qr1$ is the pseudo encoding which means that all classes in the question encoding will become the query-related-class. Rule $qr2$ transitively includes all of X_i's superclasses, for $i \in [1, n]$. Rule $qr3$ and $qr4$ include P_all (see the definition) and P_all's clone-froms.

```
qr1: qrc(Q,Xi) :-
      Q(QID, QType, X1, X2, ... Xn).
(for all 1<= i<= n)
qr2: qrc(Q,C) :- qrc(Q,M),
       has(M,superclass,C).
qr3: qrp(Q,P) :- qrc(Q,C),
       has(P,prototype_of,C).
qr4: qrp(Q,P) :- qrp(Q,X),
       has(X,cloned_from,P).
```

We define the query-related-instances as "qri(Q,I)", which will be the constraint used for other rules so that only the instances that are of interest is selected and reasoned with.

Definition 12: *"qri(Q,I)" contains:*

(i) all the query-related-prototypes from the previous definition, and

(ii) all the inheritable nodes in the object graphs for the query-related-prototypes.

Rules $qr6$ and $qr7$ encode the definition for $qri(Q, I)$:

```
qr6: qri(Q,I) :- qrp(Q,I).
qr7: qri(Q,I) :- qri(Q,Y),
                 has(Y,S,I), inheritable(S).
```

Here we show how $qri(Q, I)$ is applied to the other rules as a filter or a constraint. We discussed the rule for "tc-cloned-from" in section 3, which will compute for all the instances in the KB. By applying the constraint to $c1$ and making it $c1'$ (similarly to $c2$) the "tc-cloned-from" rules will only compute the instances of interest.

```
c1': cloned_from(X,Y) :- X!=Y,
        has(X,cloned_from,Y), qri(Q,X).
```

In rule $s2$ where we compute the subsume relations between two instances, we need to constrain both instances using $qri(Q, I)$. The resulting rule $s2'$ is as follows:

```
s2': subsume(V1,V2) :-
            qri(Q,V1), qri(Q,V2),
            V1!=V2, inheritable(S),
            not not_subsume(V1,V2).
```

After all the necessary constraints are applied to the rules $i1$, $i2$, $c1$, $c2$, $m1$, $m2$ (all similar to $c1'$), and $s1$, $s2$ (all similar to $s2'$), the resulting program is significantly more efficient.

Query execution timings

We tested the rules on a corpus that contains 5180 classes, 32339 instances, 5673 entries in the class hierarchy graph, and 198301 entries in the various object graphs. The running time with the efficient encoding is around 3 seconds per question, while the earlier encoding ran for more than two hours without giving an answer.

Correctness result

Definition 13 *We define a program Π'_{KB} w.r.t the Knowledge base as the answer set program consisting of:*

(i) all the rules in Definition 9 (i) substituted with the corresponding rules that have $qri(Q, I)$ (as described in Definition 12) as a constraint, and

(ii) all the facts as in Definition 9 (ii).

Theorem 2 Given a query Q, if $qri(Q, X)$ is true then X has value V for slot S after cloning and unification in a KB iff:

$$\Pi'_{KB} \models hasc(X, S, V)$$

Proof (sketch): The proof is similar to that of Proof of Theorem 1, with one additional element, the use of $qri(Q, X)$. It is used in all the rules to focus the answer set of Π'_{KB} on instances that are related to the query. Hence, if for some X, $qri(Q, X)$ is true, then the answer set of Π'_{KB} has all the necessary facts related to X.

Corollary 1 Given a query Q, if $qri(Q, X)$ is true then

$$\Pi_{KB} \models hasc(X, S, V)$$

iff

$$\Pi'_{KB} \models hasc(X, S, V)$$

Encoding of Question-Answering

The AURA system supports seven types of questions, namely: *computing a slot value, checking if an assertion is true or false, identifying superclasses, comparing individuals, describing a class, computing the relationship between two individuals, and giving an example of a class.* Our current ASP based system can answer all the types of questions that is supported by AURA. In this section, we show some of the comparatively difficult ones. (We do not show the ASP rules for "identifying superclasses" and "giving an example of a class", as they are straight forward.)

Describe a class: The most simple yet fundamental question is: "What is X?". There are two ways to answer this question, (i) return the full list of properties for X, or (ii) return the most distinguishable properties of X, in comparison to its immediate superclass.

The first way provides a direct answer, and can be obtained by using the object graph $G'_{obj}(X)$ and obtaining a list of facts of the form $hasc(X, S, V)$ from it. However, it has the downside of returning a long list of properties which does not specify the unique properties that X has and its superclasses do not.

The second way is better in a sense that only the unique properties of X are returned. In this approach one needs to compare X with its superclasses and return the differences.

We use the following rule to find X's immediate superclass(es) (also referred to as "most-specific-class"):

```
msc(X, MSC) :-
        has(X, superclass, MSC).
```

We then define that the unique properties of X as the ones that X has but the MSC doesn't, and we use the following predicate to denote it

```
diff(X, MSC, S, V, h1).
```

The detailed encoding of this predicate is given below under the heading of question type III: "Class differences".

Class similarities: This is another fundamental type of question, which asks for the similarities between two classes. For example, "What are the similarities between prokaryotic cell and eukaryotic cell?" To answer, we introduce the predicate $sim(X, Y, S, V)$ to stand for the same value V of the same slot S shared by two instances X and Y. The rules for this predicate are as follows:

```
sim(X,Y,S,V) :-
        hasc(X,S,V), hasc(Y,S,V1),
        V==V1, X!=Y.
sim(X,Y,S,V) :-
        hasc(X,S,V), hasc(Y,S,V1),
        X!=Y, V!=V1,
        tc_cloned_from(V,V1),
        tc_cloned_from(V1,V).
```

We say two values are the same if either they have the same name, or they clone from each other.[10]

Class differences:

Here we discuss how to answer questions like: "What are the differences between prokaryotic cell and eukaryotic cell?" To answer, we define "difference" as: *What instance X has but Y doesn't, and vice versa.* We use the predicate *diff* (X, Y, S, V, H) to stand for: X has V for S but not Y if H=h1, and Y has V for S but not X if H=h2. We introduce H to represent which one of X and Y holds the value V.

```
diff(X,Y,S,V,h1) :- X!=Y,
        hasc(X,S,V), not sim(X,Y,S,V).
diff(X,Y,S,V,h2) :- X!=Y,
        hasc(Y,S,V), not sim(X,Y,S,V).
```

Computing a slot value: This is a common type of questions, and a representative example is *What is the agent of adhension-of-water (X)?*. We can think of this question as a one-hop search of the slot value A given an instance X and a slot name S. So the answer is A if the relation $hasc(X, S, A)$ holds.

We consider another example of the question: *What chemical bond is the agent of adhension-of-water (X)?*, in which more properties (as constraints) are posed in the question. The answer to this question is A if the following relations hold: hasc(X,S,A), tc_instance_of(A, P1), P1 = chemical_bond.

We use $P1$ to denote the additional property, and according to the question the answer A must be a "chemical bond". The answer is "hydrogen bond" which is both the agent of X, and is a subclass of chemical bond.

Check if an assertion is true or false: Such a question is asking whether an assertion of the form *(X,R,Y)* is true or not. For example, "Is it true that an animal cell is a eukaryotic cell?" is asking whether *(animal_cell, is-a, eukaryotic_cell)* is true or not.

So the answer to the question is true if we can conclude $h(X, R, Y)$.

```
answer(Q, true) :- h(X,R,Y).
```

The rules to infer $h(X, R, Y)$ need to be added according to the content of the question. For this example, we add the following rule, which means that *X is-a Y holds if Y is an ancestor of X*.

```
h(X,R,Y) :- ancestor(X,Y), R = is_a.
```

In addition we have the rules:

```
ancestor(X, A) :-
   has(X, superclass, A).
ancestor(X, A) :-
   ancestor(X, M),
   has(M, superclass, A).
```

The following rule returns $false$ when we cannot conclude that the answer is $true$.

```
answer(Q, false) :-
   not answer(Q,true), question(Q).
```

Compute relationship between two individuals: We use an example to illustrate this type of questions: "What is the relationship between photosynthesis and cellular respiration?" The ideal answer would be: the result of cellular respiration is the raw material for photosynthesis. Hence the question is looking for the same values shared by different slots of the two instances. We define the predicate "relation" to entail this meaning:

```
relation(X,SX,Y,SY,V) :-
          hasc(X,SX,V), hasc(Y,SY,V1),
          V==V1, X!=Y, SX!=SY.
relation(X,SX,Y,SY,V) :-
          hasc(X,SX,V), hasc(Y,SY,V1),
          tc_cloned_from(V,V1),
          tc_cloned_from(V1,V),
          X!=Y, SX!=SY.
```

This encoding is similar to $sim(X, Y, S, V)$, with an additional constraint that the slot names have to be different.

Conclusion and Future Work

Similar to AI planning where there is somewhat of a mismatch in terms of research planners (mostly not HTN based) and industrial planners (mostly Hierarchical Task Network - HTN (Sacerdoti 1974) based) there seems to be a mismatch between KR logics and the formalisms used in large knowledge bases. A large number of the latter seem to use frame based representation while there is comparatively less theoretical research (outside of DL) on frame based representation. Some exceptions include DLV+, FAS and F-logic (Kifer and Lausen 1989). In this paper we show how answer set programming can be used to encode some procedural reasoning mechanisms in frame based systems, in particular "cloning" and "unification". We then show that those ASP encodings can be further enhanced with appropriate filters for making answer finding query (pattern) driven, as in Magic Sets (Bancilhon, Sagiv, and Ullman 1986). Thus we show how to answer various kinds of questions with respect to large knowledge bases constructed by domain experts in particular domains in a reasonable amount of time. In the future we plan to explore answering other kinds of questions, such as Why and How questions with respect to such large knowledge bases.

Acknowledgement

We would like to thank Vinay Chaudhri for encouraging us to pursue research in this direction, for clarifying many of our doubts regarding Aura and frame based knowledge representation in general, and for giving us access to an expert constructed large knowledge base. We acknowledge the support of NSF grant 0950440.

[10]There exist cycles caused by the clone-from relation in KB. However, it may not always be correct that X is the same as Y if X clones from Y and Y clones from X. Here we just point out that the logic rules can easily accommodate different conditions.

References

Altman, R.; Buda, M.; Chai, X.; Carillo, M.; Chen, R.; and Abernethy, N. 1999. Riboweb: An ontology-based system for collaborative molecular biology. *Intelligent Systems and Their Applications, IEEE* 14(5):68–76.

Alviano, M.; Ianni, G.; Marano, M.; and Martello, A. 2008. Versatile semantic modeling of frame logic programs under answer set semantics. *The Semantic Web* 106–121.

Baader, F.; Calvanese, D.; McGuinness, D.; Nardi, D.; and Patel-Schneider, P. 2003. *The description logic handbook: theory, implementation, and applications*. Cambridge Univ Press.

Bancilhon, F.; Sagiv, Y.; and Ullman, J. 1986. Magic sets and other strange ways to implement logic programs. In *Proc. 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*.

Baral, C. 2003. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press.

Brewka, G. 1987. The logic of inheritance in frame systems. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, 483–488.

Chaudhri, V. K.; Clark, P. E.; Mishra, S.; Pacheco, J.; Spaulding, A.; and Tien, J. 2009. Aura: Capturing knowledge and answering questions on science textbooks. Technical report, SRI International.

Clark, P.; Porter, B.; and Works, B. 2004. Km: The knowledge machine 2.0: Users manual.

Fikes, R., and Kehler, T. 1985. The role of frame-based representation in reasoning. *Communications of the ACM* 28(9):904–920.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Ostrowski, M.; Schaub, T.; and Thiele, S. 2008. A user's guide to gringo, clasp, clingo, and iclingo. *November* 77:78–80.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In Kowalski, R., and Bowen, K., eds., *Logic Programming: Proc. of the Fifth Int'l Conf. and Symp.*, 1070–1080. MIT Press.

Gunning, D., and et, a. 2010. Project halo update - progress toward digital aristotle. *AI Magazine*.

Hayes, P. J. 1979. The logic of frames. In Metzing, D., ed., *Frame Conceptions and Text Understanding*. Walter De Gruyer. 46.

Horty, J. 1994. Some direct theories of non-monotonic inheritance. In Gabbay, D., and Hogger, C., eds., *Handbook of Logic in AI and logic programming*.

Karp, P.; Riley, M.; Saier, M.; Paulsen, I.; Collado-Vides, J.; Paley, S.; Pellegrini-Toole, A.; Bonavides, C.; and Gama-Castro, S. 2002. The ecocyc database. *Nucleic acids research* 30(1):56.

Kifer, M., and Lausen, G. 1989. F-logic: A higher-order language for reasoning about objects, inheritance, and scheme. In *SIGMOD Conference*, 134–146.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2006. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL)* 7(3):499–562.

Minsky, M. 1975. A framework for representing knowledge. In Winston, P., ed., *The Psychology of Computer Vision*. New York: McGraw-Hill. 211–277.

Niemelä, I., and Simons, P. 1997. Smodels—an implementation of the stable model and well-founded semantics for normal logic programs. *Logic Programming and Nonmonotonic Reasoning* 420–429.

Ricca, F., and Leone, N. 2007. Disjunctive logic programming with types and objects: The dlv+ system. *Journal of Applied Logic* 5(3):545–573.

Sacerdoti, E. 1974. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence* 5:115–135.

Touretzky, D. 1986. *The mathematics of inheritance systems*. Los Altos, Ca.: Morgan Koufmann.