Lecture Six: Representing Defaults

301315 Knowledge Representation and Reasoning

A General Strategy for Representing Defaults - What Is a Default

- **default** is a statement of natural language containing words such as “normally”, “typically”, or “as a rule”.
- Example: “Normally, birds can fly.”
- We use them all the time. Well, we normally use them.
- The fact that something is a default implies that there are exceptions.
- Therefore, any conclusions based on the default are tentative.

A General Strategy for Representing Defaults - What Is a Default

Example (1)

A scenario: *John and Alice are Sam and Bill's parents. YOU (the agent) are Sam's teacher and Sam is doing poorly in class. You tells John that Sam needs some extra help to pass, because you are thinking:*

(a) *John is Sam's parent.*

(b) *Normally, parents care about their children.*

(c) *Therefore, John cares about Sam and will help him study.*

Question: How do you represent the default?

A General Strategy for Representing Defaults - What Is a Default

We may use the following rule:

```
cares(X,Y) :- parent(X,Y).
```

But what does happen if we later on find out John does not care about his children's study. Can we add the following rule into the knowledge base:

```
-cares(john,X) :- parent(john,X).
```

We get a contradiction!

A General Strategy for Representing Defaults - Using "Abnormal Predicate"

In ASP a default d, stated as "Normally elements of class C have property P," is often represented by a rule:

$$\begin{aligned} p(X) \quad \leftarrow \quad & c(X), \\ & \mathit{not}\ ab(d(X)), \\ & \mathit{not}\ \neg p(X). \end{aligned}$$

- $ab(d(X))$ is read "X is abnormal with respect to d" or "a default d is not applicable to X"
- $\mathit{not}\ \neg p(X)$ is read "$p(X)$ may be true."

A General Strategy for Representing Defaults - Using "Abnormal Predicate"

Example (2 - Continued)

```
cares(X,Y) :- parent(X,Y),
              not ab(d_cares(X,Y)),
              not -cares(X,Y).
```

Note that we have no problem when we add

```
-cares(john,C) :- parent(john,C).
```

The new program is consistent and entails $\neg cares(john, sam)$ and $cares(alice, sam)$.

A General Strategy for Representing Defaults - Two Types of Exceptions

- **Weak exceptions** make the default inapplicable. They keep the agent from jumping to a conclusion.
- **Strong exceptions** allow the agent to derive the opposite of what the default would have them believe.

Cancellation axiom for weak exception:

$$ab(d(X)) \leftarrow not\ \neg e(X)$$

Cancellation axiom for strong exception:

$$\neg p(X) \leftarrow e(X).$$

A General Strategy for Representing Defaults - Implementation of Exceptions

- When encoding a weak exception, add the *cancellation axiom*:

  $$ab(d(X)) \leftarrow not\ \neg e(X),$$

  which says that d is not applicable to X if X may be a weak exception to d.
- When encoding a strong exception, add the *cancellation axiom* (above), and the rule that defeats the default's conclusion.

  $$\neg p(X) \leftarrow e(X).$$

A General Strategy for Representing Defaults - Implementation of Exceptions

Example (3 - Weak Exception)

Suppose our agent doesn't want to assume too much about folks caring about their children if they haven't ever been seen at school.

Notice that doesn't mean that the agent assumes the worst - only that it doesn't know and wants to be cautious.

So, it doesn't want to apply the cares(P,C) default to anyone that is "absent".

What should it assume about Alice caring for Sam if it knows:

- that Alice has been seen at Sam's school ($\neg absent(alice, sam)$)?
- that Alice has never been seen at Sam's school ($absent(alice, sam)$)?
- nothing about Alice's absence?

A General Strategy for Representing Defaults - Implementation of Exceptions

Following our general method for defaults, we'll add the cancellation axiom:

$$ab(d(X)) \leftarrow not\ \neg e(X).$$

For default `d_cares`, the axiom is as follows:

```
ab(d_cares(P,C)) :- not -absent(P).
```

"A person P is abnormal w.r.t. the default about caring for child C if P may be absent."

A General Strategy for Representing Defaults - Implementation of Exceptions

Now Let's put everything together:

```
parent(alice,sam).
parent(john,sam).
cares(X,Y) :- parent(X,Y),
              not ab(d_cares(X,Y)),
              not -caress(X,Y).
ab(d_cares(X,Y)) :- not -absent(X,Y).
```

A General Strategy for Representing Defaults - Implementation of Exceptions

What can we obtain from this program?

- What can we get provided `absent(alice,sam)`?
- What can we get provided `-absent(alice,sam)`?
- What can we get provided nothing?

A General Strategy for Representing Defaults - Implementation of Exceptions

Example (4 - Strong exception)

We already represented uncaring John in our program as following strong exception:

```
-cares(john,C) :- parent(john,C).
```

In a more general case of handling exceptions, we need two rules to encode both weak and strong exceptions as follows:

$\neg p(X) \leftarrow e(X),$
$ab(d(X)) \leftarrow not\ \neg e(X).$

Back to our example, these become:

```
-cares(john,C) :- parent(john,C).
ab(d_cares(john,C)) :- not -parent(john,C).
```

A General Strategy for Representing Defaults - Implementation of Exceptions

But in our this example, we do not have to include the second rule:

```
-cares(john,C) :- parent(john,C).
ab(d_cares(john,C)) :- not -parent(john,C).
```

Why? Because we already have

```
-cares(john,C) :- parent(john,C).
```

This rule directly defeats the following rule:

```
cares(X,Y) :- parent(X,Y),
              not ab(d_cares(X,Y)),
              not -caress(X,Y).
```

A General Strategy for Representing Defaults - Implementation of Exceptions

```
parent(alice,sam).
parent(john,sam).
cares(X,Y) :- parent(X,Y),
              not ab(d_cares(X,Y)),
              not -caress(X,Y).
ab(d_cares(X,Y)) :- not -absent(X,Y).
-cares(john,C) :- parent(john,C).
%%
%% ab(d_cares(john,C)) :- not -parent(john,C).
```

But in a more general case, we will need both weak and strong exception rules.

A General Strategy for Representing Defaults - Implementation of Exceptions

Example (5)

Scenario: *Suppose there is a mythical country, called u, whose inhabitants don't care about their children. Suppose our knowledge base contains information about the national origin of most but not* **all** *recorded people.*

Pit and Kathy are Jim's parents. Kathy was born in Moldova, but we don't know where Pit is from. He could have been born in country u.

We also assume that both parents have been seen at school, so the absence thing doesn't come into play.

A General Strategy for Representing Defaults - Implementation of Exceptions

```
parent(pit,jim).
parent(kathy,jim).
born_in(kathy,moldova).
%% A person can only be born in one country
-born_in(P,C1) :- born_in(P,C2), C1!= C2.
%% The original default
cares(X,Y) :- parent(X,Y),
              not ab(d_cares(X,Y)),
              not -cares(X,Y).
%% Representing the strong exception
-cares(P,C) :- parent(P,C),
               born_in(P,u).
ab(d_cares(P,C)) :- not -born_in(P,u).
```

A General Strategy for Representing Defaults - Implementation of Exceptions

Note that the second rule of exception cannot be omitted, because without this rule, the agent would believe that Pit is a caring parent of Jim!

This is somewhat too brave reasoning.

A General Strategy for Representing Defaults - Implementation of Exceptions

10 min classroom exercise

Consider the knowledge base Σ consisting of the following rules:

$$
\begin{array}{lcl}
enrol(X, Y) & \leftarrow & not\ ab(d\_preRequsite(X, Y)). \\
 & & not\ \neg enrol(X, Y). \\
\neg enrol(X, Y) & \leftarrow & \neg completed\_preRequsite(X, Y). \\
ab(d\_preRequsite(X, Y)) & \leftarrow & not\ completed\_preRequsite(X, Y).
\end{array}
$$

Now suppose we have the following facts:

$enrol(alice, databases).$
$completed\_preRequsite(bob, ai).$

What can we derive from Σ about Alice and Bob's enrolling units?

Preference between Defaults - Orphans Example Revisited

Recall Orphans program *orphans.lp*:

```
%% Facts
  child(mary). child(bob).
  father(mike,mary). father(rich,bob).
  mother(kathy,mary). mother(patty,bob).
  dead(rich). dead(patty).

%% Rules for Closed World Assumption
  -child(X) :- not child(X).
  -father(F,C) :- child(C),
                  not father(F,C).
  -mother(M,C) :- child(C),
                  not mother(M,C).
  -dead(X) :- not dead(X).
  -orphan(X) :- not orphan(X).
```

Preference between Defaults

```
%% Defining "parents_dead"
  parents_dead(P) :- father(F,P),
                     mother(M,P),
                     dead(F),
                     dead(M).

%% Defining orphans
  orphan(P) :- child(P),
               parents_dead(P).

%% Defining negation of "parents_dead"
  -parents_dead(X) :- -orphan(X).
```

Preference between Defaults - New Information

- Now we remove the assumption that we have information for every child's parents - note this is a more realistic situation.
- We also want to add some policies for supporting children:
  (a) Orphans are entitled to program 1
  (b) All children are entitled to program 0
  (c) Program 1 is preferable to program 0
  (d) No child can receive support from more than one program
- Question: how can we represent these policies into Orphans program?

Preference between Defaults - Representing Defaults

```
%% Default d1: An orphan is entitled to program 1:
entitled(X,1) :- record_for(X),
                 orphan(X),
                 not ab(d1(X)),
                 not -entitled(X,1).
%% Default d2: A child is entitled to program 0:
entitled(X,0) :- record_for(X),
                 child(X),
                 not ab(d2(X)),
                 not -entitled(X,0).
%% A person is not entitled to more than one program:
-entitled(X,P2) :- record_for(X),
                   entitled(X,P1),
                   P1 != P2.
```

Preference between Defaults -Capturing Strong Exceptions

```
%% Strong exception:
%% An orphan is not entitled to program 0:
-entitled(X,0) :- record_for(X),
                  orphan(X).
%% Default d2 cannot be applied if a person
%% may be an orphan - weak exception:
ab(d2(X)) :- record_for(X),
             not -orphan(X).
%% Other strong exceptions:
%% X is not entitled to any program if X is dead:
-entitled(X,N) :- record_for(X),
                  dead(X).
%% X is not entitled if X not a child:
-entitled(X,N) :- record_for(X), -child(X).
```

Preference between Defaults - Verifying Joe

Now consider what kind of supporting program will living child Joe get if

- Joe he is an orphan?
- Joe is a child but not an orphan?
- we don't know whether he is an orphan?

Preference between Defaults - Working with Unknowns

We can detect when we don't know something about a person in our KB. If we add the following rule to the program:

```
check_status(X) :- record_for(X),
                   not -orphan(X),
                   not orphan(X).
```

Then a query on `check_status(X)` will list everyone that we don't have orphan information about.

Preference between Defaults - Adding New CWA's

Some sample records.

```
%% Records
record_for(bob). father(rich,bob).
mother(patty,bob). child(bob).

record_for(rich). father(charles,rich).
mother(susan,rich). dead(rich).

record_for(patty). dead(patty).

record_for(mary). child(mary).
mother(kathy,mary).
```

Preference between Defaults - Adding New CWA's

We add new Closed World Assumption rules:

```
-dead(P) :- record_for(P),
            not dead(P).

-child(X) :- record_for(X),
             not child(X).
```

Only apply to people in the database because we are only asking questions about people with records.

Inheritance Hierarchies with Defaults - Submarines Example Revisited

Recall that we have a rule stating that all submarines are black:

```
has_color(X,black) :- member(X,sub).
```

Suppose we want to change this statement to "normally, submarines are black", which can be represented as:

```
has_color(X,black) :- member(X,sub), not ab(d(X)),
                      not -has_color(X,black).
```

Now suppose we have:

```
is_a(blue_deep,sub).
has_color(blue_deep,blue).
-has_color(X,C2) :- has_color(X,C1), C1 != C2.
```

Inheritance Hierarchies with Defaults - Submarines Example Revisited

We can also allow exceptions to an object not belonging to two sibling classes at the same time.

```
member(X,C) :- is_a(X,C).
member(X,C) :- is_a(X,C0), subclass(C0,C).

siblings(C1,C2) :- is_subclass(C1,C),
                   is_subclass(C2,C),
                   C1 != C2.

-member(X,C2) :- member(X,C1),
                 siblings(C1,C2),
                 C1 != C2,
                 not member(X,C2). %% add this part
```

Inheritance Hierarchies with Defaults - Adding Exception to Membership

Now, suppose we have the following facts:

```
is_a(darling, car). % darling is both a car and a sub
is_a(darling, sub).
is_a(narwhal, sub).
```

Question: Will we derive contradiction about `darling`'s membership?

Inheritance Hierarchies with Defaults - Hierarchy with Defaults

In commonsense reasoning, quite often, we need to deal with defaults with different specificities.

> Scenario: *Eagles and penguins are types of birds. Birds are a type of animal. Sam is an eagle, and Tweety is a penguin. Tabby is a cat. Animals normally do not fly, birds normally fly, but penguins normally don't fly.*

Can Sam fly?

Inheritance Hierarchies with Defaults - Hierarchy with Defaults

- Our common sense tells us that Sam can fly, because we all accept that more specific information overrides less specific information.
- Thus, when encoding defaults of classes, we assume that default: "normally elements of class C1 have property P", is preferred to the default: "normally elements of class C2 have property $\neg$ P" *if* C1 is a subclass of C2.

Inheritance Hierarchies with Defaults - Hierarchy with Defaults

Example (6)

Some part of the code for program *fly.lp*:

```
%% Sibling classes are disjoint unless we are
%% specifically told otherwise.
siblings(C1,C2) :- is_subclass(C1,C),
                   is_subclass(C2,C),
                   C1 != C2.

-member(X,C2) :- member(X,C1),
                 siblings(C1,C2),
                 C1 != C2,
                 not member(X,C2).
```

Inheritance Hierarchies with Defaults - Hierarchy with Defaults

```
%% default d1: Animals normally do not fly.
-fly(X) :- member(X,animal),
           not ab(d1(X)),
           not fly(X).

%% default d2: Birds normally fly.
fly(X) :- member(X,bird),
          not ab(d2(X)),
          not -fly(X).

%% default d3: Penguins normally do not fly.
-fly(X) :- member(X,penguin),
           not ab(d3(X)),
           not fly(X).
```

Inheritance Hierarchies with Defaults - Hierarchy with Defaults

```
%% X is abnormal with respect to d2 if X might
%% be a penguin.
ab(d2(X)) :- not -member(X,penguin).

%% X is abnormal with respect to d1 if X might
%% be a bird.
ab(d1(X)) :- not -member(X,bird).
```

Tutorial and Lab Exercises

1. Consider the following program Π:

r_1: $q(a, b)$.
r_2: $r(a, c)$.
r_3: $p(X, Y) \leftarrow q(X, Y), not\ ab(d\_p(X, Y))$,
r_4: $ab(d\_p(X, Y)) \leftarrow not\ \neg r(X, Y)$,
r_5: $\neg r(X, Y) \leftarrow not\ r(X, Y)$.

- Translate this program into *clingo* syntax, and run it under *clingo*. Note that in order to make your *clingo* program be safe, you will need to modify some rules in the original program.
- Then by removing rule r_5, run the program again. Compare the result with the previous one, what do you observe?

Tutorial and Lab Exercises

2. Complete Example (6) program, by adding:
   - subclass facts - the eagle is a kind of bird, the penguin is a kind of bird, the bird is a kind of animal, the cat is a kind animal ;
   - animal facts - Sam is an eagle, Tweety is a penguin, and Tabby is a cat;
   - subclass rules; and
   - membership rules;

   and run the program under *clingo*.

Tutorial and Lab Exercises

3. Creating a knowledge base in the domain of student unit enrolment, which is described as follows:
   - In order to enrol a unit, the following conditions must be satisfied: (1) the unit is on offer in the current semester; (2) the student has to complete all pre-requisites of this unit; and (3) the student's study load must not be over four units altogether in the current semester.
   - All units enrolled by a student in the current semester must be mutually excluded in pre-requisites.
   - Your knowledge base should precisely encode the above general rules, and contain at leat 10 different units and 20 students.

   Based on the knowledge base you create, write a *clingo* program and test it with various queries.