Table of Contents I

Representing Defaults

- A General Strategy for Representing Defaults
- Knowledge Bases with Null Values
- Simple Priorities between Defaults
- Inheritance Hierarchies with Defaults
- Indirect Exceptions to Defaults

Reading

- Read Chapter 5, *Representing Defaults*, in KRR book.

What Is a Default?

- **default** — 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.

Example: Uncaring John

- Let's consider our family story where John and Alice are Sam and Bill's parents.
- You are Sam's teacher and he is doing poorly in class.
- You tell John that Sam needs some extra help to pass.
- You're thinking:
  1. John is Sam's parent.
  2. Normally, parents care about their children.
  3. Therefore, John cares about Sam and will help him study.
- How do we represent the default?

Why Can't We Just Use a Strict Rule?

- If we add a strict rule

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

  and later find out that John doesn't care about his kids:

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

  then we get a contradiction!

A General Way of Representing Defaults

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

$$\begin{array}{lcl} p(X) & \leftarrow & c(X), \\ & & not\ ab(d(X)), \\ & & not\ \neg p(X). \end{array}$$

- $ab(d(X))$ is read "X is abnormal with respect to d" or "a default d is not applicable to X"
- $not\ \neg p(X)$ is read "$p(X)$ *may* be true."
- This works regardless of the arity of p.

Example: Normally parents care about their children.

```
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 $cares(alice, sam)$.

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.

General 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 *and* the rule that defeats the default's conclusion.

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

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

Example: Adding a Cancellation Axiom

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

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

for default `d_cares` 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."

Example: What does the agent know about Alice?

Let's put the default together with the cancellation axiom:

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

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

What does the agent conclude given

- `-absent(alice)`?
- `absent(alice)`?
- no information about Alice's absence?
- What if it knew `cares(alice,sam)`?
- How about `-cares(alice,sam)`?

Example: Strong Exception — General Methodology

We already represented uncaring John in our program as follows:

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

How would we implement the strong exception of uncaring John using the general methodology?
It says to add two rules:

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

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

In our case,

- $p(X)$ is $cares(john, C)$,
- $e(X)$ is $parent(john, C)$, and
- $d(X)$ is $d\_cares(john, C)$.

Thus, our two rules are

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

Example: General Methodology

Sometimes, we can do better than the general methodology.
It's a one-size-fits-all, and we can make it tailor made.
Here is the default plus the two rules again:

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

-cares(john,C) :- parent(john,C). %rule 1

ab(d_cares(john,C)) :- not -parent(john,C). %rule 2
```

Check that we don't need rule 2 *in this case.*

Example: Sometimes We Need Rule 2 (The Cancellation Axiom)

- Let's consider another strong exception to *d_cares*.
- Suppose the 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 there Pit is from. He could have been born in *u*.
- Let's assume that both parents have been seen at school, so the absence thing doesn't come into play.

Representing the Strong Exception

Assume we have all necessary sorts and predicates. Does Kathy care about Jim? Does Pit? What if the last rule were missing?

```
father(pit,jim).
mother(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).
```

Example: Cowardly Students

1. Normally, students are afraid of math.
2. Mary is not.
3. Students in the math department are not.
4. Those in CS may or may not be afraid.

The first statement corresponds to a default. The next two can be viewed as strong exceptions to it. The fourth is a weak exception.

Let's look at the implementation in `s_cowardly.sp` on the book webpage: `http://pages.suddenlink.net/ykahl`.

What does the program assume?

- `? afraid(john,math)`
- `? afraid(mary,math)`
- `? afraid(pat,math)`
- `? afraid(bob,math)`
- Let's add a new person, Jake, whose department is unknown. What does the agent assume?

Defaults with Known Information

- Let d be a default "Elements of class C normally have property P" and e be a set of exceptions to this default.
- If our information about membership in e is complete, then its representation can be substantially simplified.
- If e is a weak exception to d then the Cancellation Axiom can be written as

$$ab(d(X)) \leftarrow e(X).$$

- If e is a strong exception then the cancellation axiom can be omitted altogether.

Example: Defaults with Known Information

- Suppose we had a complete list of students in the CS and math departments.
- The cancellation axiom for CS students could be simplified to

```
ab(d(X)) :- in(S,cs).
```

- The one for the math students could simply be dropped. Remember, we still have

```
-afraid(S,math) :- in(S,math_dept).
```

Knowledge Bases with Null Values

Consider a database table representing a tentative summer schedule of a Computer Science department.

Professor	**Course**
mike	pascal
john	c
staff	prolog

Here "staff" is a null value.

Course Catalog Implementation

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sorts

#prof = {mike, john}.
#prof_values = #prof + {staff}.
#course = {pascal, c, prolog}.
#default = d(#prof_values, #course).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
predicates

teaches(#prof_values, #course).
ab(#default).
```

Course Catalog Implementation, cont.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
rules

teaches(mike,pascal).
teaches(john,c).
teaches(staff,prolog).

-teaches(P,C) :- not ab(d(P,C)),
                 not teaches(P,C).

ab(d(P,C)) :- teaches(staff,C).
```

How does the agent answer queries:

- `? teaches(mike,c)`
- `? teaches(mike,prolog)`

Another Type of Incompleteness

Professor	Course
mike	pascal
john	c
{mike, john}	prolog

In this case, we simply add:

```
teaches(mike,prolog) | teaches(john,prolog).
```

Another Type of Incompleteness

Here are the rules of the program:

```
teaches(mike,pascal).
teaches(john,c).
teaches(mike, prolog) | teaches(john, prolog).

-teaches(P,C) :- not teaches(P,C).
```

How does the agent answer queries:

- ? $teaches(mike, c)$
- ? $teaches(mike, prolog)$
- ? $teaches(mike, prolog) \wedge teaches(john, prolog)$

Simple Priorities between Defaults

- Recall our orphans story.
- Remove the assumption that we have info about every child's parents. (Note that this is more realistic.)
- Add information about some regulations:
  1. Orphans are entitled to assistance from government program 1.
  2. All children are entitled to program 0.
  3. Program 1 is preferable to program 0.
  4. No one can receive assistance from more than one program.

Orphans Example: Representing the 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.
```

Orphans Example: Expressing Preference for Program 1

- Treat orphans as strong exception to the second default.
- (We can use weak exceptions to express preference, too.)
- Recall: We don't have complete info about who is an orphan because we don't have complete info about status of parents.

```
%% 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:
ab(d2(X)) :- record_for(X),
             not -orphan(X).
```

Orphans Example: We Have 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 to any program if he is not a child:
-entitled(X,N) :- record_for(X),
                  -child(X).
```

Information about *dead* and *child* is complete, so we don't need the cancellation axioms.

Orphans Example: Verifying Joe

Let's look at the entitlement rules together and check whether all is well.
See `s_orphans2.sp` on the book webpage:
`http://pages.suddenlink.net/ykahl`.

What kind of assistance will living child Joe get if

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

Orphans Example: Working with Unknowns

We can detect when we don't know something about a person in our KB.

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

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

Orphans Example: Some Sample 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(patty,mary).
```

Orphans Example: CWA's

We know who is a child and who is dead:

```
-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.

Orphans Example: Defining Orphans Given Incompleteness

- The positive part doesn't change, but CWA not valid for the negative part.
- Use a weaker statement.

```
orphan(P) :- child(P),
             parents_dead(P).

-orphan(P) :- record_for(P),
              not may_be_orphan(P).

may_be_orphan(P) :- record_for(P),
                    child(P),
                    not -parents_dead(P).
```

Orphans Example: Support Predicates

This time, we have to define when parents are not dead.

```
parent(X,P) :- father(X,P).
parent(X,P) :- mother(X,P).

parents_dead(P) :- father(X,P),
                   dead(X),
                   mother(Y,P),
                   dead(Y).

% -parents_dead(P) is true if we can find a parent
% that is not dead.
-parents_dead(P) :- parent(X,P),
                    -dead(X).
```

Orphans Example: Adding New Knowledge

Suppose our administrator did her research and found that Mary has a father, Mike, who is alive. She knows that he is not a child and not dead, so she can add a record for him.

```
father(mike,mary).
record_for(mike).
```

Now Mary is entitled to program 0 but not 1.

Absence of Information vs. Falsity

Why don't we enter records for Charles and Susan (Bob's grandparents)?

In the end we know:

- who is entitled to which program;
- who is not entitled;
- who we don't have enough information about even though they are in our KB and when that's a problem and when it's not.

Orphans Example: Adding Defaults to SPARC

```
sorts
#person = {mary, bob, rich, patty, charles, susan}.
...
#default1 = d1(#person).
#default2 = d2(#person).
#default = #default1 + #default2.

predicates
ab(#default).
...
```

Submarines Revisited

Change "all submarines are black"

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

to "normally, submarines are black."

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

Consider

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

Membership Revisited

Now we can 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
```

So, there is no contradiction with

```
is_a(darling, car). % is both a car and a sub
is_a(darling, sub).
is_a(narwhal, sub). % still just a sub and not a car
```

The Specificity Principle

Here is a classic story that came from the study of inheritance hierarchies.

> *"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, penguins normally don't fly."*

Can Sam fly? How do you know?

The Specificity Principle as Prioritized Defaults

- Our common sense tells us that he can because *more specific information overrides less specific information.*
- David Touretsky first formalized this idea known as the **specificity principle**.
- Thus, when encoding defaults of classes, we assume that The default "normally elements of class C_1 have property P" is preferred to the default "normally elements of class C_2 have property $\neg P$" if C_1 is a subclass of C_2.

Hierarchy with Defaults

See s_tweety.sp at http://pages.suddenlink.net/ykahl.

Indirect Exceptions to Defaults

- These are rare exceptions that come into play only as a last resort, to restore the consistency of an agent's world view when all else fails.
- Probably can't be done with straight ASP.
- Can be done with CR-Prolog, an extension of ASP.

The Contingency Axiom

- The **contingency axiom** for default $d(X)$ which says that *"Any element of class c can be an exception to the default $d(X)$ above, but such a possibility is very rare and, whenever possible, should be ignored."*
- This can be expressed by adding rules to ASP which fire only when there is a contradiction which can be resolved by their consequences.
- (CR-Prolog allows us to define preferences between these rules, but we will not cover that now.)

Example: Restoring Consistency

$$
\begin{array}{l}
p(a) \leftarrow \ not\ q(a). \\
\neg p(a). \\
q(a) \stackrel{+}{\leftarrow} .
\end{array}
$$

The regular part of this program is inconsistent. However, the third rule allows for the resolution of the conflict and the program's answer set is $\{q(a), \neg p(a)\}$.

Abductive Support and Answer Sets of CR-Prolog

- Let Π^r denote the regular rules of program Π.
- Let $\alpha(R)$ be the set of regular rules obtained from consistency-restoring rules by replacing $\stackrel{+}{\leftarrow}$ with $\leftarrow$.

Definition

(Abductive Support)

A minimal (with respect to the preference relation of the program) collection R of cr-rules of Π such that $\Pi^r \cup \alpha(R)$ is consistent (i.e. has an answer set) is called an **abductive support** of Π.

Definition

(Answer Sets of CR-Prolog)

A set A is called an *answer set* of Π if it is an answer set of a regular program $\Pi^r \cup \alpha(R)$ for some abductive support R of Π.

Example: Broken Car

Default: People normally keep their cars in working condition:

$$
\begin{array}{lcl}
\neg broken(X) & \leftarrow & car(X), \\
 & & not\ ab(d(X)), \\
 & & not\ broken(X). \\
broken(X) & \stackrel{+}{\leftarrow} & car(X).
\end{array}
$$

Turning the ignition key starts the car's engine:

$$
\begin{array}{lcl}
starts(X) & \leftarrow & turn\_key(X), \\
 & & \neg broken(X). \\
\neg starts(X) & \leftarrow & turn\_key(X), \\
 & & broken(X).
\end{array}
$$

$$car(c).$$

$$turn\_key(c).$$

Regular rules conclude: $\neg broken(c)$ and $starts(c)$.
What if $\neg starts(c)$?

What Is It For?

- planning
- diagnostics
- reasoning about an agent's intentions

How Do I Run It?

- CRModels is a solver for CR-Prolog which includes preferences, etc.
- For our use, we can run simple programs in SPARC using `:+` instead of $\stackrel{+}{\leftarrow}$.