1988

Artificial Intelligence through Prolog by Neil C. Rowe

Rowe, Neil C.

Prentice-Hall

http://hdl.handle.net/10945/36984

Abstraction of facts

Abstractions of search problems were considered in the last chapter. We'll now look at abstractions on groups of facts: *frames*. Frames are invaluable when you've got lots of facts because they organize them. We'll discuss the different kinds of frames, the components of frames, and new issues in inheritance with frames.

Partitioning facts

Frames--sometimes called *classes*, sometimes *prototypes*, and sometimes *structured-object descriptions* (which doesn't necessarily mean "objects" with physical presence)--partition facts in an artificial-intelligence system. Partitioning of large programs into subroutines and procedures is very familiar in computer science; Section 6.6 mentioned partitioning rules into groups related by topic. But we can also partition data (facts). The computer language Smalltalk exploits extensive fact partitioning. The usual way is to group together facts about the same thing (or *object*).

Prolog dialects usually already have built-in fact partitioning in their indexing together of the facts with the same predicate name. But frames usually work differently: they group facts with the same argument values. So these facts

```
a_kind_of(enterprise,carrier).
a_kind_of(vinson,carrier).
location(enterprise,san_diego).
location(vinson,san_francisco).
color(enterprise,gray).
color(vinson,gray).
east_of(vinson,enterprise)
```

might make two frames (partitions): one of facts about the Enterprise (the first, third, fifth, and seventh facts) and one of facts about the Vinson (the second, fourth, sixth, and seventh facts). With the built-in Prolog indexing by predicate name, there would be four implicit partitions: the two **a_kind_of** facts, the two **location** facts, the two **color** facts, and the **east_of** fact. The first way seems more like how people organize facts in their heads.

Frames, like semantic networks, are intended for two-argument predicates. This includes the Chapter 2 categories of relationship predicates (in which both arguments are objects), and property predicates (in which the first argument is an object, the second argument a property of that object). (If you have more than two arguments to your facts, you must convert them to relationships and properties somehow.) Relationship-predicate facts are generally stored only with the first-argument frame; then for the second-argument frame, a different "reverse" predicate name is used. For instance, for **a_kind_of(X,Y)**, **a_generalization_of(Y,X)** is the reverse; for **part_of(X,Y)**, **contains(Y,X)** is the reverse.

Frames and slots

A frame is more than a collection of facts; it is an abstraction in its own right. Often we feel that certain facts are the only essential ones about some object, the ones giving descriptive *completeness*. For instance, for a ship the name, identification number, type of ship, nationality, tonnage, and location might be the important

features. We should make sure we always put those facts in a frame for a ship.

There's a problem, however: we may want to describe sets of objects with a frame, not just a single object. Why? We might want to generalize about certain groups or classes of objects, identifying properties each group shares. Then we can't always fill in all the essential facts in the frame since different objects may have different facts true. For instance, a frame for American ships: each ship has a different identification number, so we can't assert a single identification-number fact for the entire frame.

But identification number is still a *potential* fact for the frame. The terminology is that frames have *slots* that can be *unfilled* or *filled*; filled slots represent facts. For instance, ships vary in location, so frames for "ship" and "carrier" should leave the "location" slot unfilled; but the "transport medium" slot of ships should always be filled with the word "water". Some slots are so important to fill that their frames don't make sense otherwise. Such slots are *definitional*, part of the definition of the frame. For instance, the "nationality" slot of the frame for "American ships" must be filled in with the value "American", because that's what the frame is all about. Many pairs of frames having **a_kind_of** relationships have such slots, which came about when the more general frame was restricted in some slot to create the more specific frame.

Inference rules can fill unfilled slots, especially inheritance rules (Section 4.9). That is, to fill a slot in a frame F, find a related frame F2, take the value for the same-named slot in F2 provided the slot inherits across the relationship between F and F2. For example, if all carriers are gray, that color fact can be stored with the "carrier" frame and inherited via **a_kind_of** to the frames for the carriers "Enterprise" and "Vinson". Inheritance can be overridden by putting explicit values into normally-unfilled slots, stating the exceptions to general principles. For instance, the few carriers that are not gray can have an explicit color value in their "color" slot.

Slots qualifying other slots

Slots may have more than a value attached to them: they can have information that explains or qualifies that value. Slots with numeric values can have associated information about the units of measurement and its accuracy. Slots with nonnumeric values can have a format (a formal description of what the values look like). Slots representing real-world data can have an associated location, time, and observer. Slots can have default or "usual" values. Slots may have associated sets of permissible values, given by a list if the number of such values is finite, or by a range if values are numeric. Slots may also have sets of "unusual" values that should generate a warning if seen.

For instance, a slot for the current latitude of a ship can have as explanatory or qualifying information:

--an indication that the format is a one-digit or two-digit number followed by the letter N or S;

--the units "degrees";

--the time a ship was at that position;

--who reported it;

--a note that the number must be less than 91;

--a note that it's unusual for the number to be more than 60;

--a default value that is the latitude of Norfolk (a good default for U. S. Navy ships, anyway).

Such *slot qualification* information can be considered as slots itself, and we'll call them *qualifying slots*. They can inherit like regular slots. For instance, the restriction that the latitude be less than 91 holds for anything on the surface of the earth, not just a ship, and so could inherit from the corresponding information in a more general frame like "vehicle_on_earth" or "physical_object_on_earth". Notice that qualifying slots generally inherit from a higher-generalization frame than do the slots they describe, when both inherit.

Frames with components

Frames can represent things with components. Often just listing component names is insufficiently descriptive. Instead we should have a separate frame for each component, and **part_of** slots pointing to them. This has several advantages:

--We can distinguish properties of components not shared by the whole (inheritance can always be used otherwise).

--We can describe relationships of the components to one another, like the relative location of parts of a physical object or the relative time of subevents of an action. Frames for which components can be put in a linear sequence are called *scripts*.

--We can indicate more easily how many times component frames occur in relation to the main frame (*cardinality* information, which can be expressed as a qualifying slot). For instance, a wheel is part of a car, and cars have four of them, each with their own properties; and a sunroof is an optional part of a car, occurring either zero or one times in every car.

--We only need identify the most important, "top-level" components in our main frame. The frames for these components can describe their subparts, and so on.

Frames as forms: memos

Frames have many applications to management of paperwork, because frames are a lot like forms. For instance, consider memos used for communication in organizations. Figure 12-1 shows some memo frames linked in an **a_kind_of** lattice. All memo frames have certain slots: author, intended readers (addressees), date written, subject, type (formal or informal) and text. Often the first four are even written on the paper to look like slots, with the symbols TO, FROM, DATE, and RE followed by colons.

The general memo frame has many (not necessarily exclusive) sub-frames, each linked by **a_kind_of(X,memo)** facts. Examples are a frame for all the memos written by Ann, a frame for all the memos received by Ann, and a frame for all memos about the budget. These last two have, among others, a sub-frame of their own: all memos received by Ann about the budget. And this might happen to have another sub-frame, for a specific memo Ann got on this subject the other day. Of course, there are many other memo sub-frames. Just considering sub-frames restricting the values in the "subject" slot, there are "policy" memos, "announcement" memos, and "personal" memos. Considering sub-frames restricting the "date" slot, there are memos of critical and short-term importance, memos of immediate but long-term importance, and memos of only long-term importance. Finally, note that the "memo" frame itself is a sub-frame of a "business form" frame.

Since memo components appear in sequence down a page, their frames are scripts. But organizations do differ in these orders, so the component sequence is inherited from frames specific to each organization.

The author, addressee, and text slots shown are special to memos because they must be filled in for every specific memo *instance* (that is, real-world memo), though not for every memo frame (which can represent classes of memos), with new slots added at various levels in the lattice of memo frames. This is a special type of "cardinality" information for those slots, which can be expressed in qualifying slots.

Slot inheritance

All memos have a text slot, but this isn't filled in unless we're talking about a specific real-world memo; every memo's text is different, obviously. But you must agree that since the general "memo" frame has a "text" slot, the "memos to Ann" frame must have a "text" slot too. This is inheritance, but of a fundamentally different sort than any in this book so far: it's inheritance of the mere *concept* of a slot, and not the value in a slot. We'll call this *slot inheritance*, and the regular kind *value inheritance*. In one sense, slot inheritance is more limited than value inheritance in that it usually only works with the **a_kind_of** relationship. That is, situations when some frame F has a slot S, then anything that is "a kind of" F also has slot S. But in another sense, slot inheritance is more general than value inheritance because it applies higher up in frame hierarchies, to slots that don't yet have values filled in.

Usually slot inheritance tells us most of the slots that must be in a sub-frame given the slots in a frame. But some slots may be unique to the sub-frame. For instance, carrier ships have airplanes, whereas ships in general don't. So slots referring to airplanes, like those indicating number and types, are unique to the "carrier" frame. But slots in a frame can't "disappear" in a sub-frame.

Part-kind inheritance

Besides value and slot inheritance there is a third fundamental kind of inheritance: part-kind inheritance. It occurs in the interaction between **a_kind_of** and **part_of** (see the example in Figure 12-2). A ship has decks, a hull, and a propulsion system. So the ships Enterprise and Vinson have those things. But the Enterprise's hull is *different* from the Vinson's hull. It may have different properties: it may be damaged while the Vinson's hull or the hull of ships in general isn't. Some kind of inheritance is happening from the "hull" frame, but it's not inheritance of values or slots because "hull" isn't either a value or slot but a frame itself. We'll call this *part-kind inheritance*. Formally, it's the inheritance of a component FC of some frame F of which frame FK is "a kind of"; the inherited frame becomes a component of FK, inheriting slots and values from both FK and FC. Part-kind inheritance means you don't need to define every frame in advance for which you want to assert slot values: some frames must exist because others exist. Part-kind inheritance is often signaled in English by possessives, like "Enterprise's" in the phrase "the Enterprise's hull".

Extensions versus intensions

Philosophers make an important distinction between abstract concepts (what they call *intensions* or *meanings* of concepts) and real-world things that frames correspond to (what they call *extensions*). (Don't confuse this first word with the more common but differently-spelled one "intention".) For instance, the intension of "carrier" is an abstract specification of what it means for something to be a carrier. One extension of "carrier" is the set of all currently existing carriers, another the set of all carriers that existed a year ago, another the set of all carriers ever built, and another the set of all carriers that will exist ten years from now (the "world" of

the extension may be hypothetical).

Extensions and intensions generally require separate frames, linked by "extension" and "intension" pointers to one another, because the slots can be filled differently. For instance, it's technically incorrect to say that the Enterprise is at 14N42W, only that the extension of the Enterprise for the current instant of time has a location value of 14N42W. So only extensions of things can have a location. In general, extensions can have statistics while intensions cannot. So if you want to fill a slot with something that's usually true or true on the average, you need an extension frame. For instance, if you want to say "most tankers are Liberian", you're making a statement about the nationality statistics of some extension of the abstract concept of "tankers".

Properties of extensions usually inherit from their intensions: an abstract concept constrains the examples of it. For instance, the fact that ships float on water by definition (i.e., as a property of their intension) means that existing ships (i.e., an extension) float on water. But statistical properties of extensions rarely inherit in any direction. For instance, the average length of a memo in some organization isn't likely to be the average length of a policy memo in that organization. Weak inferences are sometimes possible between sets and their subsets, if the condition of statistical independence holds; sampling theory from statistics can help us recognize such situations.

Procedural attachment

Inferences rules (or inference procedures) can be values in slots. Usually they appear in qualifying slots, and represent a way to fill the value in the main slot from other accessible values. This is *procedural attachment*, and the procedures are sometimes called *if-needed rules*. Rules can inherit just like other slot values, so we can specify a rule for many frames with a single value entry.

Extensive or exclusive use of procedural attachment in an artificial-intelligence system leads to a whole new style of programming, *object-oriented programming*. We mentioned it back in Section 6.9, as one interpretation of parallelism in rule-based systems. Object-oriented programming is especially useful for writing simulations. The simulation is divided into *objects*, each with its own frame and rules running independently. This is a sort of opposite to the *procedure-oriented programming* using Prolog that is emphasized in this book, for which procedures (rules) call on data. With object-oriented programming, data (in frames) calls on procedures. As such, object-oriented programming requires a whole different mindset or programming philosophy than procedure-oriented programming, and it's hard to intermingle it with the other things we discuss in this book. If you're interested, take a look at the Smalltalk literature.

Frames in Prolog

Frames are best implemented in Prolog with software modules where each module has its own local facts, rules, and indexing scheme. Unfortunately, only some of the currently available Prolog dialects provide true modules. But some of the meaning of frames can be captured by identifying them with files, making each frame a separate file.

Filled slots in a frame can just be facts in the file with the usual predicate names. Unfilled slots can be represented by facts of the form

```
slot(<object>,<slot-name>).
```

assuming **slot** is not a slot name itself. This special predicate can inherit by slot inheritance from that object is

a kind of.

Qualifying information (like units, bounds, and measurement accuracy) can be specified as special property facts where the predicate name is the name of the qualifying slot, the first argument is name of the slot qualified, and the second argument is the value in the qualifying slot. For instance:

```
units(length,meters).
```

Ordering between parts can be modeled by additional relationship facts besides those designating the parts themselves. For instance:

```
beneath(engine_room,flight_deck,carrier).
```

which says that the engine room part of a carrier is beneath the flight deck part.

Example of a frame lattice

Here are some interrelated frames about cars; Figure 12-3 shows their relationships. To make inference rules simpler, we use triples to represent slot values instead of the usual two-argument predicates. The syntax is **value(<object>,<slot>,<value>)** which means that the **<slot>** of the **<object>** has the **<value>**.

Physical-object frame:

```
slot(physical_object,weight).
slot(physical_object,name).
slot(physical_object,use).


units(physical_object,weight,kilograms).
```

Vehicle frame:

```
value(vehicle,a_kind_of,physical_object).
value(vehicle,use,transportation).
value(vehicle,has_a_part,propulsion_system).


slot(vehicle,owner).
slot(vehicle,dealers).
slot(vehicle,year).
slot(vehicle,age).
slot(vehicle,propulsion_method).


units(vehicle,age,years).
units(vehicle,year,years).
```

Car frame:

```
value(car,a_kind_of,vehicle).
value(car,propulsion_method,internal_combustion_engine).
value(car,has_a_part,electrical_system).
value(car,extension,cars_on_road).
```

```
slot(car,make).
slot(car,model).


possible_values(car,make,
  [gm,ford,chrysler,amc,vw,toyota,nissan,bmw]).
```

Electrical-system frame:

```
value(electrical_system,part_of,car).
value(electrical_system,has_a_part,battery).
value(electrical_system,has_a_part,starter).
```

VW-Rabbit frame:

```
value(vw_rabbit,a_kind_of,car).
value(vw_rabbit,make,vw).
value(vw_rabbit,model,rabbit).
```

Cars-on-the-road-now frame:

```
value(cars_on_road,intension,car).


statistic(cars_on_road,mean,age,6.4).
```

Joe's-VW-Rabbit frame:

```
value(joes_rabbit,a_kind_of,vw_rabbit).
value(joes_rabbit,extension,joes_rabbit_now).
value(joes_rabbit,owner,joe).
value(joes_rabbit,year,1976).
```

Joe's-VW-Rabbit-now frame:

```
value(joes_rabbit_now,subset,cars_on_road).
value(joes_rabbit_now,intension,joes_rabbit).


statistic(joes_rabbit_now,size,none,1).
```

Joe's-VW-Rabbit-battery frame:

```
value(joes_rabbits_battery,extension,joes_rabbits_battery_now).
value(joes_rabbits_battery,part_of,joes_rabbit).
```

Joe's-VW-Rabbit-battery-now frame:

```
value(joes_rabbits_battery_now,intension,joes_rabbits_battery).
value(joes_rabbits_battery_now,contained_in,joes_rabbit_now).
value(joes_rabbits_battery_now,status,dead).
```

Now for some inference rules applying to these frames. We first define a single general-purpose value-

inheritance rule, that can do inheritance for arbitrary predicate names. This makes inheritance implementation a lot simpler since we don't have to write a separate rule for each predicate that inherits. Technically, we've implemented something close to a *second-order logic*, something that reasons about predicate names as well as arguments, as we'll explain more in Chapter 14.

```
has_value(Object,Slot,V) :- value(Object,Slot,V), !.
has_value(Object,Slot,V) :- inherits(Slot,Relation),
  value(Object,Relation,Superconcept),
  has_value(Superconcept,Slot,V).


inherits(S,a_kind_of) :-
  member(S,[use,
   propulsion_method,dealers,year,age,make,model]).
inherits(S,part_of) :-
  member(S,[owner,dealers,year,age,make,model]).
```

Slot inheritance must be defined separately:

```
has_slot(Object,Slot) :- slot(Object,Slot), !.
has_slot(Object,Slot) :- value(Object,a_kind_of,Superconcept),
 has_slot(Superconcept,Slot).
```

Qualifying slot values also inherit downward. For instance, units of a slot:

```
has_units(Object,Slot,U) :- units(Object,Slot,U), !.
has_units(Object,Slot,U) :- value(Object,a_kind_of,Superconcept),
 has_units(Superconcept,Slot,U).
```

Here's a rule that says to get a slot value of an extension, get the corresponding slot value (possibly inherited) in the intension.

```
has_value(Extension,Slot,V) :- value(Extension,intension,I),
  has_value(I,Slot,V).


has_slot(Object,Slot) :- value(Object,intension,I), has_slot(I,Slot).
```

As we said, statistics can sometimes inherit from one extension to another, if an "independence" condition holds:

```
statistic(Extension,Statname,Slot,Value) :-
  has_value(Extension,subset,Bigextension),
  independent(Extension,Bigextension),
  statistic(Extension,Statname,Slot,Value).
```

And the subset relationship holds between two extensions whenever their intensions have an **a_kind_of** relationship:

```
has_value(Extension,subset,Bigextension) :-
  value(Extension,intension,I), has_value(I,a_kind_of,BigI),
  value(BigI,extension,Bigextension).
```

Also, some redundant slots may have their values defined in terms of other slot values. The **has_a_part** relationship is just the opposite of the **part_of** relationship:

```
value(X,has_a_part,Y) :- value(Y,part_of,X).
value(X,part_of,Y) :- value(Y,has_a_part,X).
```

And the age slot value can be defined from the year slot value:

```
value(Object,age,A) :- value(Object,year,Y), current_year(Y2),
  A is Y2 - Y.


current_year(1987).
```

Here are some sample queries run with the preceding facts and rules:

```
?- has_value(joes_rabbit_now,use,U).
U=transportation


?- has_value(joes_rabbits_battery_now,age,A).
A=11


?- has_slot(joes_rabbit_now,name).
yes


?- has_value(joes_rabbit_now,subset,X).
X=cars_on_road
```

Expectations from slots

An important applications of frames is to modeling and reconstruction of stereotypical situations in the world from incomplete knowledge. Empty slots in a frame have "expectations" about what should fill them: from inheritance, from qualifying-slot information (possible values and permissible values), and from extension statistics. Consider purchasing of equipment for a bureaucratic organization, which usually involves many steps and many details; if we know some of the details (slot values) of the purchase, then other details (slot values) are often obvious. For instance, an arriving order was probably ordered six to three weeks ago; orders from accounting-supplies companies are for the Accounting department; orders that come by express mail are probably for management and should be delivered immediately.

Frames for natural language understanding (*)

People must exploit expectations to understand natural languages, because speakers and writers try to avoid wordiness. So it's not surprising that frames are very helpful for natural-language understanding by computers, for the *semantics* or meaning-assignment subarea (as opposed to the *syntax* or parsing subarea we discussed in Section 6.12). That is, with a good frame representation we can efficiently capture the meaning of some natural-language sentences, so as to answer questions about it.

Usually the goal in interpretation of a sentence or sentences is to get a set of interrelated frames, in which each frame represents a verb or noun and its associated modifiers. Verbs and nouns are frames, and modifiers are slots or have something to do with slots. So for instance the sentence

> "Yesterday we sent headquarters by express mail the budget memo that Tom drew up for Ann on

6/12".

can be represented by three frames as in Figure 12-4: an instance or sub-frame of a "sending" frame, an instance of a "memo" frame (like those in Figure 12-1), and an instance of a "drawing-up" frame. These three are linked by the uses of their names as slot values. Some implications of the sentence are also filled in; for instance, the person a memo is drawn up for is assumed an addressee of the memo. Note that if we know more about the sending, drawing up, or the memo itself from other sentences, we could fill in additional slots in the frames without necessarily requiring more frames.

Filling in frames the right way to capture the meaning of a sentence can involve search. To be sure, the parse of the sentence (see Section 6.12) helps us considerably by identifying the grammatical categories of each word. But there are many ambiguities that can't be resolved by a parse. For instance, compare the previous sentence to:

> "Yesterday we sent several times by 4 P.M. the budget memo that Tom drew up for practice on the plane".

Here we have "several times" instead of "headquarters", "by 4 P.M." instead of "by express mail", "for practice" instead of "for Ann", and "on the plane" instead of "on 6/12". In all cases we've substituted something similar grammatically. But the functions of the substitutions are different: "several times" describes the style of the sending, not the place we sent to; "by 4 P.M." is a time limit, not how we sent; "for practice" is a purpose, not a beneficiary; and "on the plane" is a location, not a time. We must figure these things out making guesses about words, drawing from their possible meanings, and checking the resulting interpretation of the whole sentence for reasonableness.

Multiple inheritance (*)

We haven't discussed how to handle "multiple inheritance" paths for some slot of a frame. For instance, a policy memo from your boss is simultaneously a memo from your boss and a policy memo. Things are fine if only one path provides an inheritance for a slot, or if the paths all agree on some value. But if different paths give different values for a slot, we must do something. If we can assign priorities to paths, we can take the value from the one with highest priority. Or we can compromise or find a middle ground between different values. Or we can decide certain values are wrong. The next section presents an example application.

A multiple inheritance example: custom operating systems (*)

Frames are valuable in managing large software systems. As an example, take the operating system of a computer, the top-level program that runs other programs on your orders. Operating systems are standardized for an average computer user, with not too many things you can adjust. Users would prefer operating systems more custom-fitted to their needs.

One way is by a hierarchy of user models. Think of each user model as a frame holding information about defaults peculiar to a user or class of users, defaults about how they need or prefer to use particular programs and facilities. Some possible slots are:

--default terminal setting parameters;

--storage and time allocations (different programs and facilities have quite different requirements);

--the project for which this computer time should be charged;

--default non-file parameters of the program or facility itself (like for a printer, whether output is double-spaced);

--protection information for any input and output files (to prevent reading nonmeaningful files or overwriting valuable files);

--additional character strings (extensions) to be automatically added to the names of input and output files (like "pro" for all Prolog programs);

--interrupt-condition handling (like what to do on arithmetic overflow);

--common misspellings (so they can be recognized);

--common bugs (so they can be caught before damage is done);

--pointers to documentation;

--names of default editors, document handlers, and programming languages (so the user can just type short words like "edit" and the operating system will know what they mean).

The interesting thing about this application is the three independent inheritance hierarchies (see Figure 12-5). First, there's a hierarchy of user classes: frames for Tom, frames for people in Tom's project group above that, and frames for everybody using the operating system above that. This hierarchy may be a lattice, because Tom may belong to more than one project group, each with different associated projects. Second, there's a hierarchy on programs and facilities of the operating system: a frame for Tom using the Prolog interpreter, and a frame above that for Tom using any programming language, and a frame above that for Tom doing anything under the operating system. This hierarchy could be a lattice too, since for example a Prolog interpreter is both a Prolog-related facility and an interpreter. Third, there's a hierarchy on time: Tom's use of the Prolog interpreter today is below a frame for Tom's use of the Prolog interpreter anytime.

So there are at least three independent dimensions of frame generalization for the frame representing Tom using the Prolog interpreter today, the bottom frame in Figure 12-5. We can represent them by a three-dimensional lattice (which Figure 12-5 tries to suggest) in which the first (user-class) dimension runs northwest, the second (facility-class) dimension runs northeast, and the third (time) dimension runs straight north. Though each of the three hierarchies here is a linear sequence, they form a lattice when put together, since there are many routes between the top frame and the bottom frame.

The big problem is thus multiple inheritance: there are three different directions to reason. This doesn't affect slot inheritance for which nothing conflicts, but it is a serious problem for value inheritance. One approach is to give the **a_kind_of** link for each hierarchy a different name, like **a_kind_of_1**, **a_kind_of_2**, and **a_kind_of_3**. Then for every slot, designate one of these as preferred for inheritance, and store this in a qualifying slot of the original slot. For instance:

--inherit a value for the "sponsoring projects" slot via the user-class hierarchy, since projects are

composed of people;

--inherit terminal settings from the program and facility class hierarchy;

--inherit program storage allocation from the time hierarchy, the typical allocation in the past.

This designated-predicate idea runs into trouble when a user wants to override defaults obtained by inheritance. For instance, suppose user Tom wants to override default terminal settings for the Prolog interpreter, obtained from traveling up the program-class hierarchy to "Tom on languages today", and suppose Tom wants this override for all time (that is, Prolog programs might benefit from consistently unconventional settings). So he should place the overriding settings in the frame above him in the time dimension ("Tom on Prolog anytime") representing all his uses of the Prolog interpreter for all time. But now we must prevent inheritance along the program dimension. A fix is to store for each slot a sequence of inheritance predicates that should be tried to find the value of the slot--for this example perhaps the sequence of the time dimension, the program dimension, and the user-class dimension. Each slot can have an associated sequence, and this can inherit.

This approach to inheritance is general, but sometimes unwieldy. For particular slots we may be able to do something simpler. For instance, we may be absolutely certain that values won't conflict because of the way we've built the frames hierarchy. Or for slots with numeric values, we can apply a numeric function to the values inherited along different dimensions to get a "compromise" number. For example, for a storage allocation slot, we could take the maximum of the storage allocation values found along the three dimensions, because each dimension represents a different requirement that must be met. For the time allocation slot (the amount of time given to a running program before asking the user if they wanted to continue), we could take the average of values, since each number is just a rough guess about an intuitive parameter. This idea of a compromise value applies to nonnumeric slots too. For example, if Tom's project group insists on putting its name at the end of every file created by that group, while a Prolog compiler insists on putting "procomp" at the end of the name of every file compiled from Prolog code, you can compromise by putting first the project name, then "procomp". Such specialized multiple-inheritance strategies (not to be confused with "conflict-resolution" strategies for rule-based systems) can be flagged by a special "multiple-inheritance method" qualifying slot. And this slot can inherit too.

.SH Keywords:

object-based representation
frame
class
slot
filling a slot
definitional slots
qualifying slots
script
value inheritance
slot inheritance
part-kind inheritance
intension
extension
procedural attachment
semantics
multiple inheritance
frame lattice

Exercises

12-1. Consider a frame representing any class meeting at any university (that is, a generalized class hour).

(a) Give four example slots that are always filled with values and what those filled values are.

(b) Give a superconcept frame (a frame that this frame is a kind of) and a subconcept frame (a frame that is a kind of this frame).

(c) Give a slot that inherits from your superconcept frame and one that does not.

(d) Describe a script associated with a class meeting frame.

12-2. (E) People dealing with bureaucracies are constantly filling in the same information on forms over and over again. Discuss the use of inheritance to lighten this burden. In particular, explain how inheritance could have a concrete physical meaning.

12-3. (R,A) Consider a frame representing any purchase order (a form ordering the buying of something).

(a) Give an example of value inheritance from a value in this frame to a value in some other frame.

(b) Consider the "units" slot associated with some slot S in the purchase order frame (so if S is the cost slot, units would be "dollars"). Suppose S inherits its value from the same slot S of some other frame P2. What does this tell you about from where the "units" slot inherits?

12-4. (A) Different slots in a frame system require different qualifying slots to accompany them. But every time we use a slot with a particular name, we want it to have the same qualifying slots, and the same values in those qualifying slots that have values filled in. We can't always just inherit this information from a higher (more general **a_kind_of**) frame because, say, two different uses of a "length" slot might be in frames having no common ancestor having a "length" slot. (This is especially a problem when different people are building different frames for a big system.) Suggest an elegant way to handle this sort of problem.

12-5. (R,A) Every computer has a manufacturer's name and an ID number. Every computer has a CPU as a component. Consider the frame representing any computer sold by Floodge Manufacturing, Inc.

(a) Give an example of downward value inheritance from this frame.

(b) Give an example of downward slot inheritance from this frame.

(c) Give an example of downward part-kind inheritance involving this frame.

12-6. Contrast the intension and extension of the concept "memo".

12-7. Draw a semantic network representing the following facts. Represent property values by nodes too (though this isn't always a good idea). Represent what these mean, not what they literally say.

> Wrenches are tools.
> Hammers are tools.

Tools have handles.
The handle of a hammer is hard.
Wrenches are hard.
Most hammers have a steel handle.
Wrenches are 10 inches long on the average.

12-8. Names of frames often correspond to nouns. For instance, a ship frame corresponds to the meaning of the English word "ship".

(a) What, in frame terminology, does an adjective modifying the noun name of a frame often correspond to? For instance, the adjectives "big", "American", and "merchant" in the description "big American merchant ships".

(b) What, in frame terminology, does a preposition in a prepositional phrase modifying the name of a frame often correspond to? For instance, the prepositions "in" and "at" in the description "ships in the Mediterranean at noon yesterday".

12-9. (A) Consider the following example of reasoning by modus ponens:

Given: Military organizations are widely dispersed geographically.
Given: The Navy is a military organization.
Hence the Navy is widely dispersed geographically.

That conclusion makes sense. But consider:

Given: Navy organizations are widely dispersed geographically.
Given: The Naval Postgraduate School is a Navy organization.
Hence The Naval Postgraduate School is widely dispersed geographically.

That is an incorrect inference assuming the "Given"s are true. Discuss why the two situations are different. (To get full credit, you must show depth of understanding, not just cite a superficial difference.)

12-10. Represent the meaning of the following letter with a group of interlinked frames.

Dear Fly-by-Nite Software,

I tried your product "Amazing Artificial Intelligence Software" and it doesn't work. I tried Example 5 shown in the manual, and it crashed on an attempt to divide by zero. I next tried Example 7, and the lineprinter printed 100,000 line feeds. Then when I exited your program, I found it had destroyed all my files. I want my money back.

Sincerely,
Irate Programmer

12-11. Some slots are easier to handle with multiple inheritance than others.

(a) Consider associated with a nonnumeric slot a qualifying slot that lists all the conceivable values that first slot could have. Formulate a multiple inheritance policy for this qualifying slot.

(b) Consider associated with a numeric slot a qualifying slot that lists the maximum and minimum conceivable values for that first slot. Formulate a multiple inheritance policy for this qualifying slot.

12-12. For the user-model hierarchy example, formulate conflict-handling methods for multiple inheritance of the following.

(a) The directory to which output files go.

(b) A bit indicating whether the user should be warned before the operating system executes a command whose side effect is to destroy a file.

(c) The "CPU intensity" of usage, defined as the expected ratio of CPU time to login-connection time.

12-13. (A) Suppose that for our user model frames example we have qualifying slots describing how to resolve multiple inheritance on other slots. These qualifying slots could have qualifying slots themselves explaining how to resolve their own multiple inheritance problems, and these could have qualifying slots themselves, and so on. Why shouldn't we worry about an infinite loop?

12-14. (E) Chapter 7 considered the building of an expert system for appliance repair. Suppose instead of one general expert system we would like a set of expert systems, one for each kind of appliance. Many rules are common to different appliances, but some rules must be deleted, some added and some modified in going from one appliance to another. Discuss how it might be useful to define an appliance's rules as a frame and do inheritance. What slots would be necessary and useful in the frames, and how would you fill in their values?

12-15. (E) Discuss the use of a frames hierarchy to represent a contract negotiation between management and labor. Each frame will represent a particular offer or proposal. Some of these can be grouped together. State what slots can be useful for these frames. Provide a good representation of negotiation so that later analysis can more easily pick out patterns in the style of negotiation such as flexibility.

12-16. (H,E) Reasoning by analogy is a fundamentally different kind of reasoning than any so far considered in this book. It has similarities to inheritance, but it's really something else entirely: it involves four things instead of two. It's easiest to understand in terms of frames and slots. Suppose you want to know about the slots or slot values in some frame D. You could find frames A, B, and C such that the relationship of A to B is very similar to the relationship of C to D, the frame of interest. For instance, A might be a frame representing the circulatory system of the human body, and B a frame representing the medical emergency of a heart attack; then C might represent the cooling system of a car and D the serious malfunction of complete coolant blockage. Using the analogy, we might be able to infer that the "number of previous attacks" in frame B should correspond to a slot "number of previous blockages" in frame D. Furthermore, the value in the "immediate treatment" slot of frame B of "inject anticoagulants into the system" could have an analogy in a slot "immediate treatment" of frame D of "add anti-corrosives". Give a general strategy for finding such analogous slots and analogous slot values in reasoning-by-analogy situations.

Go to book index