10 Symbolic Reasoning

AI has been associated with logic since its inception. One of the earliest pioneers of AI, John McCarthy,[73] was a major proponent of logic as a computational tool in the field. And at the first AI conference[74], one of the big hits was Logic Theorist, a piece of software capable of performing logical proofs. The language Prolog, which came from the AI community, was built entirely on the presumption that computation itself could be expressed entirely in logic.

AI's connection with logic stems from many sources: the origins of AI (and computer science) in the mathematics community, the position that logic holds in the pantheon of feats of abstract human reasoning, and of course the natural connection between logic and computer systems. But another source was perhaps a supposed connection between logic and how the human brain might work.

A long time ago, people didn't even think the brain was involved in memory or thought at all. That was believed to be located somewhere in the chest (hence the notion of the "heart" being the center of "emotion"). They used to think that the brain's function was to filter the blood.

Have we come far since then? Yes and no. During the turn of the century we of course learned that much of the brain consists of neurons. The **axons** of neurons are connected to the **dendrites** of other neurons. Each such connection was called a **synapse**. Sometime later we discovered that incoming signals from the dendrites collectively excite (or inhibit) the neuron to fire an output signal (or **action potential**) down their axon to other neurons' dendrites. So in a sense the neuron performs a function on its inputs (the dendrites) to its output (the axon).

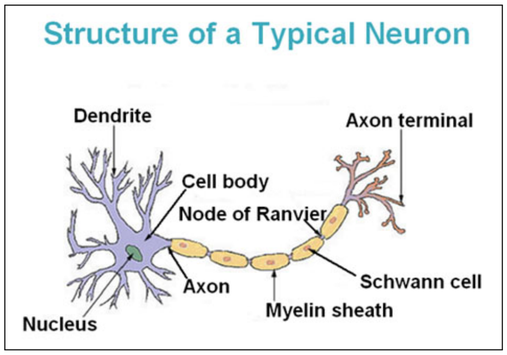

Figure 52 A Neuron. From Wikipedia. Repeated from Figure 19 in Section 4.

In the 1940s we believed that this function was probably some kind of **logical** connection. In other words, we thought that the neuron was the same as an electrical gate of sorts. Indeed, the McCulloch and Pitts neuron model (Section 4) was largely constructed on the notion that the neuron model could perform logical computation.

Consider the **grandmother cell model**[75] of neurons, where each neuron is responsible for storing and recognizing a single fact. Imagine that you owned a new red Beetle, and you had a neuron in your brain which was responsible for firing when you saw your Beetle. How did it fire? Perhaps its dendrites were hooked up to three neurons which fired when you saw (1) Red Things, (2) Cars, and (3) New Things. When you saw a Beetle which was red and new, all three of these neurons start firing, which was enough to trigger the My New Beetle Neuron to start firing, and you respond by saying, "Hey, that's my Beetle!"

[73]Indeed, he coined the term *Artificial Intelligence*, and invented the language Lisp.

[74]The Dartmouth Summer Research Conference on Artificial Intelligence, in 1956.

[75]Early on, it was thought that we stored information in our brains by growing neurons which were trained to recognize specific stimuli. For example, we used to believe that somewhere in our brains there was a cell which constantly fired when you saw a picture of your grandma (the so-called **grandmother cell**). That cell sent signals that routed to those parts of your brain that were supposed to respond in certain ways when your grandma showed up. What would happen if, after a night of binge drinking, you killed that cell and no longer recognized your grandma? Now we understand that this is not how things work.

This is essentially a logical formulation:[76]

$$\text{red} \wedge \text{car} \wedge \text{new} \Rightarrow \text{myBeetle}$$

To this we could add other neurons. Perhaps there's a neuron which fires when it sees fire trucks. If the vehicle was a fire truck, you'd need this neuron to **inhibit** the My New Beetle Neuron from firing, or you might look like a fool. Inhibition might be described as negation, so we could extend this new five-neuron arrangement to this formula:

$$\text{red} \wedge \text{car} \wedge \text{new} \wedge \neg\text{fireTruck} \Rightarrow \text{myBeetle}$$

This is *not* how neurons really work. But perhaps it gives you a taste of the early inspiration behind symbolic AI.

10.1 Logic

Many powerful AI algorithms and systems have been created using logic, and it is still one of *the* fundamental way of storing information in AI. The representation of information and storage of it in memory—particularly symbolic or logical information— is known as a **knowledge-base** or **KB**.[77] Large collections of logical rules have been used very successfully to make complicated guesses about the world, in specialized AI systems called **expert systems**. The field of storing knowledge in this fashion is commonly known as **knowledge representation**.

There are two common ways of storing logical statements in a logic KB. The first is **conjunctive normal form** (or **CNF**) and the other is **disjunctive normal form** (or **DNF**). CNF is by far the most common. In CNF your rules take the form of **clauses**, which are collections of literals ORed together. The clauses are then all ANDed together. For example:

$$\begin{array}{ll}
(\text{red} \vee \text{new}) & \wedge \\
(\text{blue}) & \wedge \\
(\neg\text{blue} \vee \text{ugly}) & \wedge \\
(\text{smart} \vee \text{ugly} \vee \neg\text{crazy}) & \wedge \\
\ldots &
\end{array}$$

In DNF, you have literals ANDed together to form clauses, and your clauses are ORed together, as in:

$$\begin{array}{ll}
(\text{red} \wedge \text{ugly}) & \vee \\
(\neg\text{smart}) & \vee \\
(\text{crazy} \wedge \neg\text{ugly}) & \vee \\
(\text{blue} \wedge \text{new} \wedge \neg\text{crazy}) & \vee \\
\ldots &
\end{array}$$

Querying in DNF can be very easy: often finding the answer to a query is as simple as looking to see if the query exists as a clause. But DNF can also be very large, and more problematically, usually they way we describe the world fits better with CNF. The problem is: CNF isn't as easy to

[76]I will assume that you understand how two first-order logics work: the propositional and predicate calculi. Also note that I use "$\wedge$" or "," for AND, "$\vee$" for OR, "$\Rightarrow$" for IF, and "$\neg$" for NOT. Another common symbol for not is $\sim$.

[77]Obviously "knowledge-base" is a play on the notionally less-sophisticated "database" or DB.

query. You can convert CNF to DNF, but it's an *exponential* computational complexity to do so. So usually it's just stuck in CNF and we have to figure out a good way to get answers out.

Why is CNF often more useful for describing real-world facts and rulew? Because typically our rules take the form of **Horn clauses**. A Horn clause is a clause that takes the following form:

$$x_1 \wedge x_2 \wedge x_3 \wedge ... \wedge x_n \Rightarrow y$$

Horn clauses have a **head** (the "y") and a **body** (the $x_1...x_n$), otherwise known as the **consequent** and the **antecedent**. None of the literals $x_1...x_n$, nor y, may be negated.

There are Horn clauses which allow negations of the literals $x_1...x_n$ (for example, we've seen one before: red $\wedge$ car $\wedge$ new $\wedge$ $\neg$fireTruck $\Rightarrow$ myBeetle. But the most common Horn clauses in knowledge representation, and those we will focus on here, are **definite Horn clauses**, and they require that all elements, on both sides of the clause, be positive.

A Horn clause of the form $x_1 \wedge x_2 \wedge x_3 \vee ... \vee x_n \Rightarrow y$ can be straightforwardly converted into a conjunctive clause (literals ORed together) of the following form:

$$\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee ... \vee \neg x_n \vee y$$

That is: negate the xs, OR them together instead of ANDing, and then OR the y. Does this look like a Prolog statement? Yep. In Prolog, a Horn clause looks like:[78]

```
y :- x1, x2, x3, ..., xn.
```

Prolog is an example of a sophisticated knowledge-base. Just as is the case in Prolog, many knowledge-bases have *only* Horn clauses and **facts**. A fact, for purposes of discussion here, is a single literal, like: x_2 or $\neg y_3$.

While all definite Horn clauses can be easily converted to CNF, not all CNF clauses can be converted into definite Horn clauses. For example:

$$\neg x_1 \vee \neg x_2 \vee x_3 \vee x_4$$

...since this has *two* positive literals, it can only be turned into a non-definite Horn clause like this:

$$x_1 \wedge x_2 \wedge \neg x_3 \Rightarrow x_4$$

Here's a simple example of a horn-clause KB and its equivalent CNF form:

	Stored in Knowledge Base (KB)	Converted to CNF
0.	red $\wedge$ stupid $\Rightarrow$ pleasing	($\neg$red $\vee$ $\neg$stupid $\vee$ pleasing) $\wedge$
1.	red	(red) $\wedge$
2.	sphere	(sphere) $\wedge$
3.	red $\wedge$ smooth $\Rightarrow$ pleasing	($\neg$red $\vee$ $\neg$smooth $\vee$ pleasing) $\wedge$
4.	sphere $\Rightarrow$ smooth	($\neg$sphere $\vee$ smooth)

[78]Obviously, the "..." is not part of the Prolog code.

10.2 Automated Theorem Proving in the Propositional Calculus

The example above is in propositional calculus. What if we now want to query our KB and ask the following question: pleasing?

To answer this our KB will have apply rules of inference repeatedly, generating new facts which it can store temporarily in its KB, until it either discovers "pleasing", or it runs out of new facts to generate. How do we know when it runs out? Perhaps when any application of any rule of inference to any combination of existing rules fails to create a new rule.

How do we know that it *will* run out? Won't it go on forever? Nope, at least not for the propositional calculus. Let's say that you have n literals in your KB, not counting negation. In our example above, we have "red, sphere, smooth, pleasing". Add on an additional symbol we'll call "$\Box$", the empty-clause. Now, what's the largest possible clause before we start repeating literals in the clause? It's one which has all of the n literals and all off their negations ORed together. So it's n in size. Let's assume that all of our clauses must be $< 2n$ in size. We'll make it simpler — let's assume that all of our clauses are exactly $2n$ in size – if they're less than $2n$ in size, we fill in the remaining space with $\Box$. Now, how many different clauses of size $2n$ can we create? To make it simple, in any off the $2n$ slots, let's say we can have any of the $2n$ literals or $\Box$. So we have $2n^(2n+1)$ or $O(n^n)$ clauses. That's a finite number. But for even small n, that's a pretty big number! In reality, it's better than that, but it's still way exponential. So although we know it'll run out eventually, it may take a *very long time*.

The reason this is true for the propositional calculus is due to two facts:

- Propositional calculus can only create a finite number of derived clauses from your ground information.
- We assume the **closed world assumption**: that if we can't derive something, it's not true.

This means that Propositional Calculus is **decidable** — with the full set of logical semantics at your disposal, you can determine if something is true, and you can determine if it is false. There are plenty of systems which are worse off than this. For example, consider the following "math logic"[79]

1. aNumber(4)
2. aNumber(x) $\Rightarrow$ aNumber($x+1$)

I'm indexing the facts in the knowledgebase (1, 2, etc.). This kind of "logic" allows us to "create" new numbers through the $x+1$ **deduction procedure**. It tells us that if x is a number, then so is $x+1$. This power to create new symbols (like 5, 6, etc.) is not available in the predicate calculus.

So using this "math logic", we ask: is 7 a number? The following **forward-chaining** proof tells us it is, by deducing more and more new facts until we discover 7:

3.	aNumber(5)	(given rules 1 and 2)
4.	aNumber(6)	(given rules 3 and 2)
5.	aNumber(7)	(given rules 4 and 2)

Great! But is 2 a number? We'll go on forever and ever applying forward-chaining, getting ever more numbers in our database, and never stumble across 2. Because you can't derive 2 from

[79]A trivial simplification of the Peano Axioms.

this. But there's no way our automatic deduction system, given the semantics available to us, can discover this!

Thus there exist some kinds of logics which are only **semi-decidable**: *if* a fact can be deduced, we can always deduce it. But if a fact *cannot* be deduced, we *may never be able to determine this*. This is very closely related to notions of **soundness** and **completeness** which will be discussed later. Suffice it to say, our "math logic" is only semi-decidable largely because it can produce an infinite number of symbols.

Resolution There are many ways of applying various rules of inference, but we will begin with a very simple rule of inference which in fact is the *only* rule of inference a computer needs to solve any propositional calculus query! This rule of inference is known as **resolution**, and it operates over CNF. The resolution rule says that if you have two clauses which differ in a literal (one says it's true, the other says it's false), as in:

$$\begin{array}{lr} A_1 \vee A_2 \vee A_3 \vee \ldots.A_n \vee & C \\ B_1 \vee B_2 \vee B_3 \vee \ldots.B_n \vee & \neg C \end{array}$$

(The A's and B's can be negated, or the same as each other — we only care about the C). Then you can **resolve** these two clauses to create a new valid clause of the form:

$$A_1 \vee A_2 \vee A_3 \vee \ldots A_n \vee B_1 \vee B_2 \vee B_3 \ldots B_n$$

Pretty simple, no? It's possible that you can create an empty clause with resolution. For example, if you resolve Z with $\neg Z$, you get nothing left. That's the **empty clause** (previously represented with $\square$). If you can derive the empty clause, then your rules have a contradiction in them somewhere. We will exploit this. Before we go on to show how you can use resolution to prove any propositional statement, let's prove that it's a valid inference method.

1.	$(A_1 \vee A_2 \vee A_3 \vee \ldots A_n) \vee C$	
2.	$(B_1 \vee B_2 \vee B_3 \ldots B_n) \vee \neg C$	
3.	$\neg(A_1 \vee A_2 \vee A_3 \vee \ldots A_n) \Rightarrow C$	Convert 1 to a (non-definite) Horn clause
4.	$C \Rightarrow (B_1 \vee B_2 \vee B_3 \ldots B_n)$	Convert 2 to a (non-definite) Horn clause
5.	$\neg(A_1 \vee A_2 \vee A_3 \vee \ldots A_n) \Rightarrow (B_1 \vee B_2 \vee B_3 \ldots B_n)$	Transitivity of inference
6.	$(A_1 \vee A_2 \vee A_3 \vee \ldots A_n) \vee (B_1 \vee B_2 \vee B_3 \ldots B_n)$	Conversion back to CNF

If your knowledge-base has no contradictions in it, then you can prove stuff with it using resolution. Here's an example. Let's say we want to prove: "pleasing" given the knowledge-base on Page 127. We're going to prove this by contradiction — we'll *assume* that ¬pleasing is true (we'll temporarily add it to our KB), and then we'll try to derive a contradiction out of the result. If we can derive a contradiction, then we know that pleasing is in fact true.

So let's start by temporarily adding ¬pleasing as clause 5 to our existing KB. Now we derive some new clauses ("$\mathcal{R}$" means "resolution with"), and add them to our KB as we go, until we successfully derive the empty clause.

Our CNF KB so far:

0.	$\neg$red $\vee$ $\neg$stupid $\vee$ pleasing	
1.	red	
2.	sphere	
3.	$\neg$red $\vee$ $\neg$smooth $\vee$ pleasing	
4.	$\neg$sphere $\vee$ smooth	
5.	$\neg$pleasing	(new clause)

Here's what we derive with resolution:

6.	$\neg$red $\vee$ $\neg$smooth	From: $\neg$pleasing $\mathcal{R}$ $\neg$red $\vee$ $\neg$smooth $\vee$ pleasing
7.	$\neg$red $\vee$ $\neg$sphere	From: $\neg$red $\vee$ $\neg$smooth $\mathcal{R}$ $\neg$sphere $\vee$ smooth
8.	$\neg$sphere	From: $\neg$red $\vee$ $\neg$sphere $\mathcal{R}$ red
9.	$\Box$	From $\neg$sphere $\mathcal{R}$ sphere

And since we were able to derive $\Box$, we're done with our proof!

The problem with resolution, as is the problem with many inference procedures is that if what you're trying to prove isn't actually true, then you'll go on and on and on until you exhaust all the possible rules, which as we've discussed can take a *long time.* Even if what you're trying to prove is *true*, you may still get lost taking the wrong path of inferences and take a very long time to find the solution.

10.3 Forward and Backward Chaining with Horn Clauses

There are two common techniques for performing logical inferences over horn clauses: forward and backward chaining. These techniques work with any kind of semantics (resolution is a forward-chaining technique, for example), but we'll look at them in action with horn clauses and solely using **modus ponens** as our semantics.

Resolution can be used to derive all possible facts from a knowledge-base. Forward- or backward-chaining with just modus ponens can't do this. It's not as powerful. But for a great many tasks involving horn clauses, it's sufficient to prove stuff we want to prove.

In forward chaining, if you want to prove something, you use your existing KB's clauses, plus inferential rules, to build up new clauses until you finally discover the item you want. In backward chaining, if you want to prove something, you *assume* that it's true, and work backwards to determine what needs to be true in order to make it true – then you try to figure out what needs to be true in order to make *that* true, and so on, until you are able to show that everything necessary is true (with your ground facts).

Here's our Horn-clause KB again:

0. red $\wedge$ stupid $\Rightarrow$ pleasing
1. red
2. sphere
3. red $\wedge$ smooth $\Rightarrow$ pleasing
4. sphere $\Rightarrow$ smooth

Let's backward-chain to prove: pleasing. First off, we will assume that ground facts (in our KB, they're "red." and "sphere." are the same thing as horn clauses with no body. So we can rewrite rules 1 and 2 as:

1. $\square \Rightarrow$ red
2. $\square \Rightarrow$ sphere

Now what we want to do is reduce pleasing to a bunch of empty clauses. We do this with an AND-OR tree, which has AND and OR dividers. To pass through an OR divider, you must find all the horn clauses whose heads match you.[80] Then your children are the bodies of those horn clauses. To pass through an AND divider, you break up your ANDed literals into separate clauses, and those become your children. If you get to $\square$, that's assumed "true". If you can't create any children, that's considered "false". For example, at right is an AND-OR tree proving pleasing.

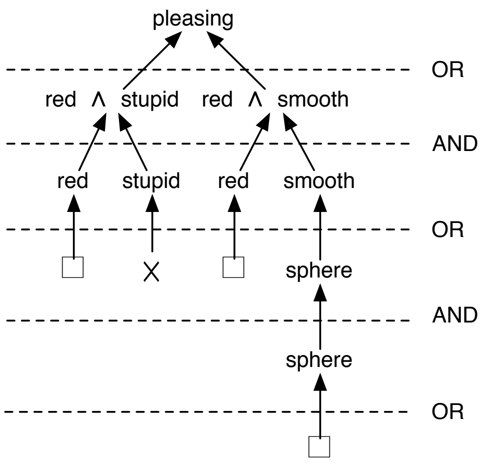

Figure 53 Propositional AND-OR Tree.

Now, let's bubble stuff back up. For an item above an AND line to be true, *all* of its children must be true. For an item above an OR line to be true, *at least one* of its children to must be true.[81] Figure 54 shows how things bubble up:

...so we know that pleasing is TRUE.

Forward chaining works in a similar fashion. In forward-chaining you create more and more clauses with inference until you finally create the clause you want. We'll use two inferential rules to do this. The first says that if A is true and B is true, then A $\wedge$ B is true. The second says that if A is true and A $\Rightarrow$ B is true, then B is true. For example:

5.	smooth	(From sphere	and	sphere $\Rightarrow$ smooth)
6.	smooth $\wedge$ red	(From smooth	and	red)
7.	pleasing	(From smooth $\wedge$ red	and	smooth $\wedge$ red $\Rightarrow$ pleasing)

...and we're done. Forward-chaining is often easier to implement, but backward-chaining is usually (in my opinion) much more successful at finding answers in a well-written KB. But backward-chaining can get caught in infinite loops if you're not careful, whereas forward-chaining tends to be more guaranteed. Prolog is a backward-chaining algorithm. Datalog (a popular database query mechanism) is a forward-chaining algorithm.

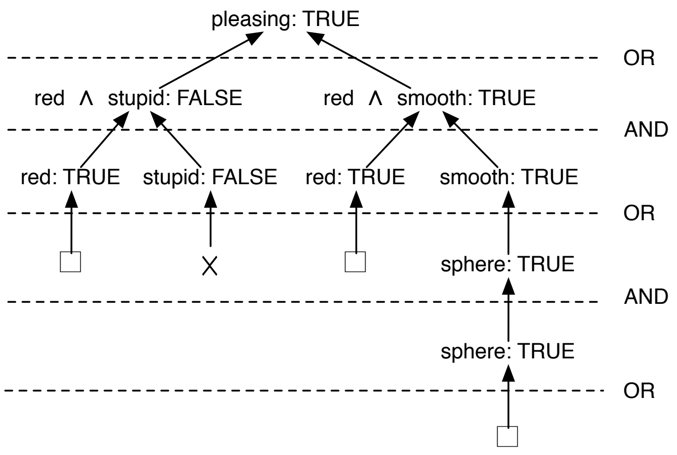

Figure 54 Propositional AND-OR Tree, Solved.

Resolution, forward-chaining, and backward-chaining are usually implemented using a state-space search algorithm. Perhaps a brute-force search like DFS or BFS, or if you have some heuristics you think help guide you, A* is often appropriate

[80]The term "OR" here makes more sense if you recall that Horn clauses can trivially be converted into CNF clauses consisting of disjunctions of all the literals. Or if you like you can call the OR layer a THEREFORE layer.

[81]Does this feel a little bit like the Min-Max game tree algorithm?

(Prolog uses DFS). What are your states? In backward-chaining, a state is your collection of things you've backtracked to, and an edge is your decision to move one of them through its next layer, which opens up some new things that change your state. In forward chaining and resolution, a state is all the clauses you know of so far (everything in your KB), and an edge is your decision to apply an inferential rule to two clauses in your KB, yielding new clauses. As you can see, the *size* of a state in forward chaining and resolution can be *much* bigger than the size of a state in backward-chaining. That's an implementation detail you'll have to deal with.

10.4 Automated Theorem Proving in the Predicate Calculus

The predicate calculus differs from the propositional calculus in that it has: (1) variables, (2) functions, and (3) quantifiers ($\forall$ and $\exists$). Because of its variables and functions, the predicate calculus can generate an infinite number of new clauses. That means that unlike the in propositional calculus, you can go on forever and forever generating new clauses in the predicate calculus, so your search for a solution is not guaranteed to ever halt.[82]

KBs that operate over the predicate calculus often store rules in Horn-clause form. However, in order to store them in a KB so that they can be queried easily, the predicate calculus clauses must be modified. The modification steps are as follows:

- Remove the existentials ($\exists$) through a process called **Skolemization**
- Remove the universals ($\forall$)
- Standardize variables

Let's tweak our KB to have some interesting predicate clause statements. Then we will go through the steps one by one to convert it into a queryable form. I will use capital letters (X, Y) for variables, and lower-case (block17) for constants.

Here's our theory to start with:

$$\begin{array}{l}
\text{red}(\text{block17}) \\
\text{sphere}(\text{block17}) \\
\text{red}(\text{block41}) \\
\text{cube}(\text{block41}) \\
\forall X, \text{red}(X) \wedge \text{smooth}(X) \Rightarrow \text{pleasing}(X) \\
\forall X, \text{sphere}(X) \Rightarrow \text{smooth}(X)
\end{array}$$

Skolemization Our first step is to remove existentials through Skolemization.[83] There aren't any in our theory (oops), but let's see how to do this for an example item that *might* be in our theory:

$$\forall Z, W, \exists X, Y, \forall Q,\ \text{foo}(X, W) \wedge \text{bar}(Z) \wedge \text{baz}(\text{yo}(Z, X), Y) \wedge \text{quux}(Q)$$

[82]You don't need two different theorem systems, because you can always convert the Propositional Calculus into the Predicate Calculus by changing each literal (foo) into a zero-variable predicate (foo()). And that's it!

[83]Thoralf Skolem was a noted Norwegian magician and logician. See http://en.wikipedia.org/wiki/Thoralf_Skolem

What we do in Skolemization is replace each $\exists$-quantified variable with a **Skolem function**.[84]

A Skolem function is a function we make up on the spot, and which consists of *all* the remaining variables *outside* the relevant $\exists$ statement. The idea is simple: $\exists$ means that there *exists* an instantiation of a certain variable. What we're doing in Skolemization is providing an instantiation.

There's a specific order you want to Skolemize your variables: from right to left (that is, because our quantifiers say $\forall Z, W, \exists X, Y, \forall Q$, we'll Skolemize Y first, then X). When we Skolemize Y, we replace it with a Skolem function, let's call it $s_1(Z, W, X)$, that consists of all the variables quantified to the left of Y. Those variables are Z, W, and X. Then we can get rid of the $\exists Y$ statement because Y isn't in the statement any more. Thus we have:

$$\forall Z, W, \exists X, \forall Q,\ \text{foo}(X, W) \wedge \text{bar}(Z) \wedge \text{baz}(\text{yo}(Z, X), s_1(Z, W, X)) \wedge \text{quux}(Q)$$

Next let's Skolemize X. The only variables quantified to the left of the $\exists X$ are Z and W. We make up a new Skolem function $s_2(Z, W)$, of these variables, and replace X accordingly:

$$\forall Z, W, Q,\ \text{foo}(s_2(Z, W), W) \wedge \text{bar}(Z) \wedge \text{baz}(\text{yo}(Z, s_2(Z, W)), s_1(Z, W, s_2(Z, W))) \wedge \text{quux}(Q)$$

Now we've converted our function into something with nothing but $\forall$ statements. Notice that one of the Skolem functions wound up with another Skolem function inside it. Once we've Skolemized all the clauses in our database, there are only $\forall$ statements left, no $\exists$ at all.

Handling Universality The second step is to remove the $\forall$ statements from the database. This is pretty easy. You just drop 'em. Let's go back to our theory and do that (remember, the Skolemized example above wasn't really in our theory — it was just done above to show how to Skolemize stuff).

$$\begin{array}{l}
\text{red}(\text{block17}) \\
\text{sphere}(\text{block17}) \\
\text{red}(\text{block41}) \\
\text{cube}(\text{block41}) \\
\text{red}(X) \wedge \text{smooth}(X) \Rightarrow \text{pleasing}(X) \\
\text{sphere}(X) \Rightarrow \text{smooth}(X)
\end{array}$$

Standardization Now all quantifiers are gone. The last step is to **standardize variables.** This means that each clause must have variables that have *different* names from each other. Two of our clauses both use a (different) variable called X. We need to change one of them:

[84]foo, bar, baz, and quux are the canonical first four metasyntactic variables. For additional variables, one faction of coders uses corge, grault, garply, waldo, fred, plugh, xyzzy, and thud (in that order). Though Guy Steele ("The Great Quux") popularized the notion of just using (as additional variables) quuux, quuuux, quuuuuux, and so on. Guy Steele is a famous computer scientist, the person behind both Scheme and Common Lisp, and the guy who wrote the semantic formalism behind Java. He's also crucially an early maintainer of the **Jargon File**, a famous dictionary of historical computer slang. Speaking of slang, I trust you understand the deep historical meaning behind xyzzy.

http://en.wikipedia.org/wiki/Metasyntactic_variable
http://en.wikipedia.org/wiki/Guy_Steele
http://en.wikipedia.org/wiki/Jargon_File
http://en.wikipedia.org/wiki/Xyzzy

$$
\begin{array}{l}
\text{red}(\text{block17}) \\
\text{sphere}(\text{block17}) \\
\text{red}(\text{block41}) \\
\text{cube}(\text{block41}) \\
\text{red}(X) \wedge \text{smooth}(X) \Rightarrow \text{pleasing}(X) \\
\text{sphere}(Y) \Rightarrow \text{smooth}(Y)
\end{array}
$$

Basic Queries And we're done! Now we can do queries on this KB. We do it using either forward or backward chaining. Let's do it with a *simple* kind of backward chaining (I'll leave forward chaining up as an exercise for you to think about). In a bit we'll learn that this isn't quite how you do it, but let's do it this way just to get the idea for the moment. We'll prove pleasing(block17). Consider Figure 55 at right.

From this Figure, we can determine that block17 is pleasing. Is block41 pleasing? Consider instead Figure 56. We can see that block41 is not smooth. This bubbles up just like in propositional calculus, and we discover that block47 is therefore not pleasing!

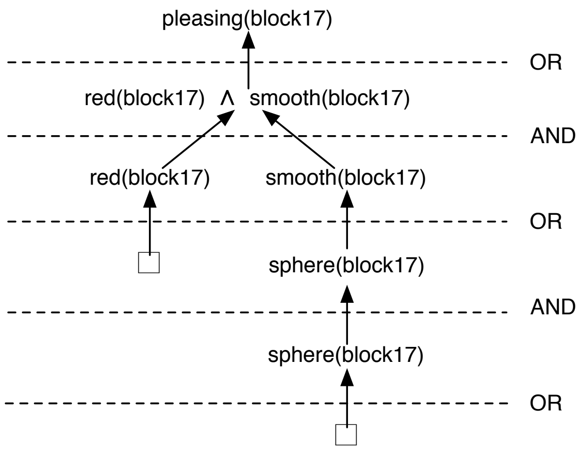

Figure 55 Predicate AND-OR Tree for the query: pleasing(block17).

Unification and Bindings But what if we want to ask: *which* blocks are pleasing? That is, pleasing(W)? But before we can do this kind of query, we need one more tool: **unification**. Unification tells us what variables must be equal to each other (or equal to what constants) in order for two clauses to "match". Let's take two clauses:

0. loves(X, mom(X))
1. loves(fred, Y)

Unification works like this. You march left to right across the clauses. At the first place the clauses differ, you try to find variable bindings to make the difference the same again. A variable binding is an assignment of a variable to another variable, to a function, or to a constant. Once a variable has been bound you can't bind it again. For example, the first place where these two clauses differ is that one says X but the other says fred. X is a variable. So we bind X to fred everywhere we see X:

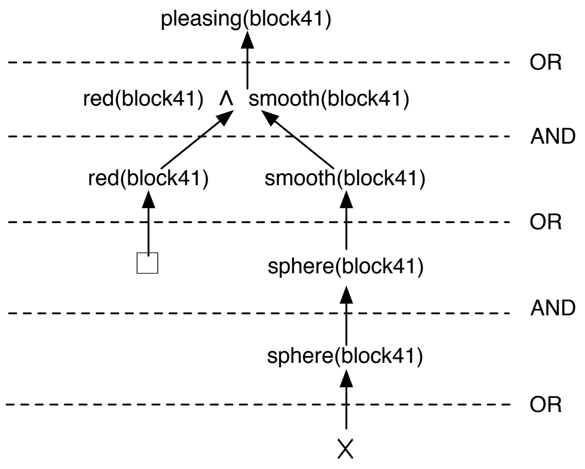

Figure 56 Solved Predicate AND-OR Tree for the query: pleasing(block17).

0. loves(fred, mom(fred)) Bindings: $X = \text{fred}$
1. loves(fred, Y)

Now our clauses differ because one says Y and the other says mom(fred). So we have a new binding $Y = \text{mom(fred)}$:

0. loves(fred, mom(fred)) Bindings: $X = \text{fred}$, $Y = \text{mom(fred)}$
1. loves(fred, mom(fred))

Now our clauses are unified, and we know what bindings are necessary to do it. It's possible that you just can't unify two clauses because they differ in a constant or in a function name – no variable to bind. Then you're stuck. For example:

0. loves(X, mom(X))
1. loves(Y, george)

Here we begin by binding X to Y:

0. loves(Y, mom(Y)) Bindings: $X = Y$
1. loves(Y, george)

...but now we're stuck: neither mom(Y) nor george is a variable. Our only tool is coming up with variable bindings. So there's no way to unify these two clauses.

Unification tries to be **most general**, meaning that we try to retain variables as often as we can rather than committing to constants. For example:

0. loves(X, Y)
1. loves(Z, mom(Z))

In our first step, we bind X to Z (or Z to X, take your pick, they're both variables):

0. loves(Z, Y) Bindings: $X = Z$
1. loves(Z, mom(Z))

...but what if we had instead decided to bind X and Y to a constant like "sean"? That would have made them match:

0. loves(sean, Y) Bindings: $X, Y = \text{sean}$
1. loves(sean, mom(sean))

Sure, we *could* have done that. After all, Sean does love his mom. But it wouldn't be a *general* matching – we've eliminated all sorts of possibilities (like the fact that George also loves *his* mom). So we don't do this if we can avoid it. Thus you *only* bind variables to constants if you can't bind them to other variables and still make the unification work. This is known as being a **Most General Unifier** (or **MGU**).

Now, we can do some backward chaining. But now in our backward chaining, at the OR layer we have to specify our bindings.

Now, bubbling up in this situation is a little trickier. We'll bubble up the bindings that make the parent true. Bindings of children bubble up to the parent through an OR layer by saying that *this* binding is true "or" *that* binding is true. Bindings of children bubble up to the parent through an AND layer only if they can be set to the same constants, or don't have to be set to a constant at all. For example:

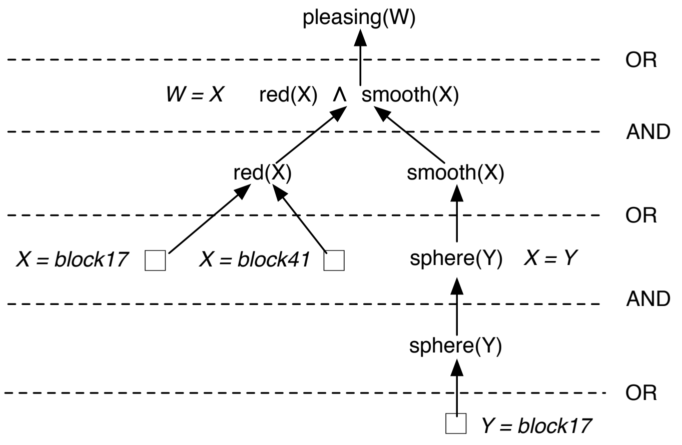

Figure 57 Predicate AND-OR Tree for the query: pleasing(W), showing bindings.

...and so we know that pleasing is true *if* $W =$ block17. Type this query into Prolog, and you'll discover it indeed prints out that $W =$ block17. That's why. Prolog is doing unification with its backward chaining.

10.5 Resolution in the Predicate Calculus

Resolution is also nicely done in the Predicate Calculus. Consider our prepared KB:

$$\begin{array}{l}\text{red}(\text{block17})\\ \text{sphere}(\text{block17})\\ \text{red}(\text{block41})\\ \text{cube}(\text{block41})\\ \text{red}(X) \wedge \text{smooth}(X) \Rightarrow \text{pleasing}(X)\\ \text{sphere}(Y) \Rightarrow \text{smooth}(Y)\end{array}$$

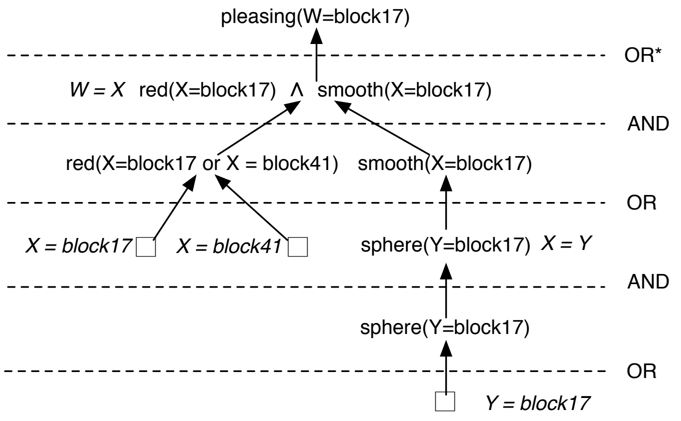

Figure 58 Solved Predicate AND-OR Tree for the query: pleasing(W), showing bindings. Note that at the layer labelled "OR*", to bubble up, only $X =$ block17 matches the same constant (block17) everywhere, and so it's the only one which passes through.

First, we must convert this into a form suitable for resolution: Namely, we need to change the horn clauses back into CNF:

$$\begin{array}{l}\text{red}(\text{block17})\\ \text{sphere}(\text{block17})\\ \text{red}(\text{block41})\\ \text{cube}(\text{block41})\\ \neg\text{red}(X) \vee \neg\text{smooth}(X) \vee \text{pleasing}(X)\\ \neg\text{sphere}(Y) \vee \text{smooth}(Y)\end{array}$$

To **resolve** two clauses in the predicate calculus, you perform a few steps:

1. Identify the literal that's different between the clauses.
2. Find a set of variable bindings which unifies that literal in clause 1 with the **negation** of the literal in clause 2.

3. Apply the variable bindings to the clauses.
4. Resolve the two clauses together as normal.
5. Before you add the new clause into the database, first standardize the variables to be different from other clauses in the database.

For example, let's **resolve** $\neg\text{red}(X) \vee \neg\text{smooth}(X) \vee \text{pleasing}(X)$ with $\neg\text{sphere}(Y) \vee \text{smooth}(Y)$. The steps are:

1. the literals we will use are $\neg\text{smooth}(X)$ and $\text{smooth}(Y)$.
2. We need to find a set of bindings that makes $\neg\text{smooth}(X)$ the same as $\neg\text{smooth}(Y)$. That's easy: set X to Y.
3. We apply the bindings to the whole clauses:

$$\neg\text{red}(Y) \vee \neg\text{smooth}(Y) \vee \text{pleasing}(Y) R \neg\text{sphere}(Y) \vee \text{smooth}(Y)$$

4. Perform resolution:

$$\neg\text{red}(Y) \vee \text{pleasing}(Y) \vee \neg\text{sphere}(Y)$$

5. We've already got Y as a variable in the database. Standardize with a new variable:

$$\neg\text{red}(Z) \vee \text{pleasing}(Z) \vee \neg\text{sphere}(Z)$$

...and there's our new fact! We can now add this fact to our database.

So let's try to prove $\exists X, \text{pleasing}(X)$. To do this, we need to add its negation to the database and try to derive a contradiction, just as we had done in resolution with propositional logic. The negation is:

$$\neg(\exists X, \text{pleasing}(X))$$

which becomes

$$\forall X, \neg\text{pleasing}(X)$$

Next, we (1) Skolemize — unnecessary, as there are no existentials in this example — then (2) remove the $\forall$ quantifiers, then (3) standardize the variable apart and add our new clause:

0.	$\text{red}(\text{block17})$	
1.	$\text{sphere}(\text{block17})$	
2.	$\text{red}(\text{block41})$	
3.	$\text{cube}(\text{block41})$	
4.	$\neg\text{red}(X) \vee \neg\text{smooth}(X) \vee \text{pleasing}(X)$	
5.	$\neg\text{sphere}(Y) \vee \text{smooth}(Y)$	
6.	$\neg\text{pleasing}(W)$	$\longleftarrow$ added

Now, let's use resolution to derive some new facts. Each time we add them to our KB:

$$\neg\text{red}(X) \vee \neg\text{smooth}(X) \vee \text{pleasing}(X) \quad \mathcal{R} \quad \neg\text{pleasing}(W) \Rightarrow \neg\text{red}(Q) \vee \neg\text{smooth}(Q)$$

$$\neg\text{red}(Q) \vee \neg\text{smooth}(Q) \quad \mathcal{R} \quad \text{red}(\text{block17}) \Rightarrow \neg\text{smooth}(\text{block17})$$

(here we bound Q to block17)

$$\neg\text{sphere}(Y) \vee \text{smooth}(Y) \quad \mathcal{R} \quad \text{sphere}(\text{block17}) \Rightarrow \text{smooth}(\text{block17})$$

(here we bound Y to block17)
...hey, we're getting somewhere here...

$$\neg\text{smooth}(\text{block17}) \quad \mathcal{R} \quad \text{smooth}(\text{block17}) \Rightarrow \Box$$

And we proved it!

10.6 Soundness and Completeness

In logic, a **system** is a set of basic axioms plus the legal **inference rules** one may perform on those axioms. The axioms in our KB are the clauses in the KB (stuff like red ∧ stupid ⇒ pleasing, or sphere, or sphere ⇒ smooth). The "inference rules" are the things we can apply to our axioms to derive new clauses for our KB. If you use resolution, that's an inference rule. If you use an and/or tree with depth-first backtracking search, then you're using **modus ponens** and a couple other inference rules. So our knowledge-base, plus our procedure for answering queries (be it backward chaining or resolution or whatever) comprise a logical "system".

The **set of theorems** of a system are all those clauses which we can possibly derive (prove) with a given system. This includes our basic KB clauses, plus all the new ones we can figure out as we're doing our resolution or backward/forward chaining. A **valid** theorem is one which cannot be "disproven" by any theorem in our system; in other words, we cannot derive the negation of the theorem. For example in our propositional-logic KB, ¬red is not a valid theorem because we can derive red, and thus we'd have a contradiction. Likewise, ¬pleasing is not a valid theorem. However, ¬stupid is a valid theorem because we can't *disprove* it by drawing a contradiction (though we can't prove it either).

Logic speaks of systems as **sound** and/or **complete**. A sound system is one whose set of theorems is entirely **valid** — there's nothing that you can prove and also disprove. In other words, a sound system has no contradictions. A complete system is one where every possible valid theorem can also be derived (proven) in our system.

Consider our simple Propositional Calculus KB again:

0. ¬red ∨ ¬stupid ∨ pleasing
1. red
2. sphere
3. ¬red ∨ ¬smooth ∨ pleasing
4. ¬sphere ∨ smooth

Is this KB **sound**? I think so: there's nothing we can prove with this KB which we can also disprove. Now, if we had an additional rule

5. $\neg$pleasing

...then we'd be in trouble. Because we can derive pleasing, and now we can also derive $\neg$pleasing. So we have a contradiction and the KB is no longer sound.

Is this KB **complete**? Nope. Because we can neither prove or disprove "stupid".

This is the formal logic way of thinking of soundness and completeness. But there's a more general notion. Imagine a system which either gives you an answer or says that it has no idea. A "sound" system is one which always comes up with correct answers — though it may not be able to figure everything out. A "complete" system always comes up with a correct answer if one exists — though it may sometimes give incorrect answers.

Many AI systems strive to be both sound and complete, so they always come up with answers if they exist, and never give incorrect answers. Hey, that's great. But is this necessary to be "intelligent"? After all, is *your* brain sound *or* complete? If you face a calculus problem, and you learned all the stuff necessary to solve the problem, often times you still can't figure it out (you're incomplete). And if you figure it out, sometimes you make a mistake (you're unsound). And yet we as humans still do pretty well. So soundness and completeness isn't absolutely necessary to still do a pretty decent job. Which is good news because it's *expensive* to be both sound and complete. Sometimes it makes sense to just make a system which takes its best shot but isn't guaranteed.

10.7 The Closed World Assumption

Soundness is sometimes easier to achieve than completeness. To compensate for possible incompleteness, one thing many AI systems assume is that they're operating in a **closed world**. Consider our KB again. What if we ask:

stupid?

What is the KB going to tell us? Many KBs will respond by simply saying: no. But the KB has no information one way or another. Why did it say no? This is because the KB *assumes* that it knows *everything* there is to know. If it doesn't know whether "stupid" is true or not, and we ask it "stupid?", it'll query its system, and when it can't find a match for "stupid?", it'll say that the answer is false.

The closed world assumption simplifies lots of stuff in databases, KBs, and logic. But it's often dangerous. Maybe the agent hasn't learned "stupid" yet. Or maybe part of its knowledge-base was unavailable at the time it did the query (perhaps the KB is spread all over the web!). Sometimes it's better to respond with "I can't say one way or the other".

10.8 Semantic Networks

Many KBs use logic underneath to come up with answers. But there's another popular approach, which is very old and is still of a lot of interest in certain AI communities, and particularly in those dealing with knowledge representation for natural language concepts.[85] This notion is called a

[85]For an easy example of this, see WordNet, a large semantic network consisting of all the words and concepts in the English language, connected by edges like "antonym" or "ISA" or "sense" (connects words to meanings). See http://wordnet.princeton.edu/

semantic network. Semantic networks have of late been in vogue for web tasks: for example, the **semantic web** is envisioned as a large semantic network.[86]

A semantic network is a collection of binary statements like hairColor(biff,brown) or age(george,24) or parent(biff,george) or parent(george,doug). These binary statements are often described with a picture, such as shown at right.

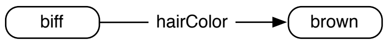

Figure 59 Simple semantic network.

The first item in the statement (biff) appears at the start of the edge. The second item (brown) appears at the end of the edge. The edge is labelled with the statement name (hairColor). A semantic network is a graph of lots of such things. Here's a slightly bigger semantic network:

These kinds of statements are called **ground statements**. They're basic facts in our semantic network. We can also have **inferred statements**. To do these, we can say things like:

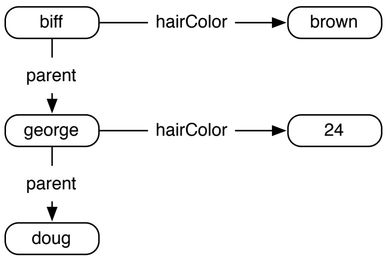

Figure 60 Slightly larger semantic network.

"Child" is the **corollary** of "Parent":

$$\text{father}(X, Y) \Leftrightarrow \text{child}(Y, X)$$

"Ancestor" is the **transitive closure** of "Parent":

$$\text{parent}(X, Y) \Rightarrow \text{ancestor}(X, Y)$$
$$\text{ancestor}(X, Y) \wedge \text{parent}(Y, Z) \wedge \text{ancestor}(X, Z)$$

With these inferred rules, we now know some new things:

child(george, biff)
child(doug, george)
ancestor(george, biff)
ancestor(doug, george)
ancestor(doug, biff) $\longleftarrow$ *this one is useful!*

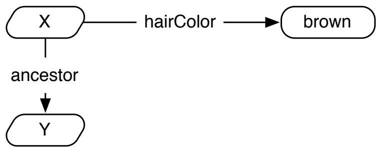

Figure 61 Semantic Network Query.

Usually in a semantic network you have relationships declared between objects, and then some statements that some relationships are specially inferred from other relationships (they're a corollary, or transitive, etc.)

A query in a network usually looks like Figure 61. This is called a **query subgraph**. This is the same as saying: $\text{hairColor}(X, \text{brown}) \wedge \text{ancestor}(X, Y)$. A semantic network often solves these subgraphs by trying to find all the matches in its network for the subgraph.

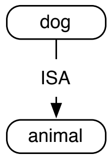

Figure 62 ISA.

Why not do this just in logic? Why bother with this graph-matching stuff? Because semantic networks often are used to express useful things which *cannot* be expressed in first-order logic: they are so-called **higher-order logics**. The most well-known such inexpressible thing is the **ISA** (or "is a") relationship, shown in Figure 62.

[86]In fact, Tim Berners-Lee has argued that he thought of the World-Wide Web as a semantic network but with the edges (links) between objects mistakenly left unlabelled. Who is Tim-Berners Lee, you ask? See http://en.wikipedia.org/wiki/Tim_Berners-Lee

This says that dog is an animal. What this means is that any relationships animal has, dog also has. For example, consider Figure 63.

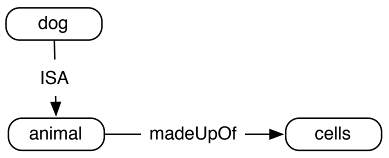

Figure 63 Transitivity of ISA.

Animals are made up of cells. Therefore dogs are also made up of cells. We can infer the fact madeUpOf(dog,cells). We say that the "madeUpOf" relationship is **transitive over** ISA.

This is essentially the same thing as saying:

$$\forall X, Y, Z, \text{madeUpOf}(X, Y) \wedge \text{ISA}(X, Z) \Rightarrow \text{madeUpOf}(Z, Y)$$

ISA transitivity works when the arrows are either direction. For example, see Figure 64. Earth contains all animals. A dog is an animal. Thus earth contains all dogs. We can infer the fact containsAll(earth,dog). This is roughly saying:

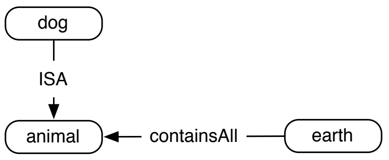

Figure 64 ISA.

$$\forall X, Y, Z, \text{containsAll}(X, Y) \wedge \text{ISA}(Y, Z) \Rightarrow \text{containsAll}(X, Z)$$

ISA is in itself **transitive**. Consider Figure 65.

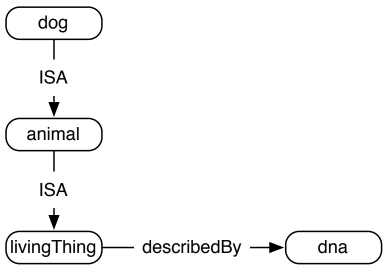

Figure 65 ISA.

DNA describes all living things. Therefore, DNA describes all animals. So we know describedBy(animal,DNA). Therefore, DNA describes all dogs. So we know describedBy(dog,DNA).

Okay, so far all this can be done easily in first-order logic. But consider the situation shown in Figure 66. Mammals are coveredWith fur. Dogs are coveredWith fur. So...is a chihuahua covered with fur or bare skin? Right now if we used logic, it would tell us that a chihuahua is covered with *both* fur and bare skin. That can't be right! Usually semantic networks have special **input semantics** which say things like: For any X, you can't have both coveredWith(X, Y) and coveredWith(X, Z) where Y isn't the same as Z. There are many such input semantics, as you can imagine. Imagine if our semantic network had input semantics for the coveredWith relation which said that you can only be covered with one item. What do you do? It's got to be one or the other.

Semantic networks use different mechanisms to deal with these possible contradictions. One common mechanism is called **inferential distance ordering** or **IDO**. IDO is very simple. It says: follow the paths along the ISA hierarchy until you get to each of the conflicting statements. Whichever one is a shorter path, that's the one you take. Deal with ties in some arbitrary fashion.

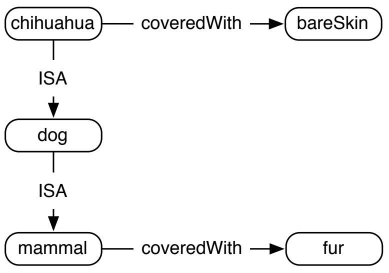

Figure 66 ISA.

chihuahuha is only 1 edge away from bareSkin, but it's 3 edges away from fur. So chihuahua is covered by bareSkin, not fur. This is an example of a **default mechanism**.

Stuff like IDO and input semantics rules are *very* useful! But they *cannot* be expressed in logic.

10.9 The Situation Calculus

Another thing logic can't do very easily is describe how things change in time. The **Situation Calculus** was an early attempt to fix this, and it's still very popular to describe stuff. But it's not used to actually *implement* things because proving with it is too slow. The Situation Calculus was invented by John McCarthy.[87]

In the Situation Calculus, you have three kinds of things:

- **Events** Things that happen.
- **Situations** Intervals of time between events (or before the first event or after the last event) where nothing changes.
- **Fluents** Adjectives which describe situations.

As an example, here are some events, in order:

1. Class Starts.
2. Sean breaks his chalk.
3. Sean starts talking about the Situation Calculus.
4. Sean breaks his chalk.
5. Sean breaks his chalk.
6. Class Ends.

Let's say that these events look like this:

START CHALK TALK CHALK CHALK END

Here we have six events, three of which are chalk-breaking events. We also have seven situations — the intervals between the events, and the time before START and the time after END. For some of those situations, we have some fluents we can say, for example:

- For the situation before START, no one is in the class.
- For the situation after END, no one is in the class. (the same fluent)
- For all the situations after TALK, the students are bewildered.
- For all the situations after the third CHALK breaks, the students are humored.

The Situation Calculus looks a lot like the predicate calculus, except with a few changes. First of all, all predicates in the calculus are fluents. A fluent is a predicate which looks like this (the V_i are variables).

[87]See Footnote 73 on page 125.

$$\textit{fluentName}(V_1, V_2, ..., V_n, \textit{situation})$$

Fluents don't have to have any variables if it's not useful. Fluents may also be negated. Some examples of possible fluents:

0.	holds(sean, chalk, s_4)	In situation 4, sean is holding chalk.
1.	$\neg$green(s_5)	It's not green in situation 5.

There is also a special function called **do**. This function takes a situation S, and an event, and returns a *new situation* resulting from the event E occurring at the end of S:

$$\text{do}(E, S)$$

Events are functions. So the situation resulting from Sean breaking his chalk in situation 4 is:

$$\text{do(break(sean,chalk),}s_4)$$

Now we're ready to say interesting things with the Situation Calculus.

The first situation is called situation 0 (or s_0. Let's say that in situation 0 the class is empty:

$$\text{empty}(s_0)$$

We might say that in situation 1 (or s_1), the class is filled with bored students:

$$\text{bored}(s_1)$$

Of course, we don't know that s_1 follows s_0: they're just symbols. Perhaps a better way to write it would be to say that in any situation immediately following a START event, the students are bored.

$$\forall S, \text{bored}(\text{do}(\text{start}(), S))$$

This says that if we're in a situation S, and we do the START event in that situation, in the resulting situation the students are bored.

Now let's say that if the students are bored in a situation S (a variable), and Sean breaks his chalk, then we get a new situation where the students are no longer bored:

$$\forall S, \text{bored}(S) \Rightarrow \neg\text{bored(do(break(sean,chalk), } S))$$

This says literally "if we're in a situation S where the students are bored, then the result of Sean breaking his chalk in S will be a new situation where the students are not bored".

You can also say interesting things like: if Sean is in the classroom, then the students are in the room (and vice versa). Let's invent a function called "location(person,room,S)" which returns a fluent that means that the person is in the room. Now we can say:

$$\forall S : \text{location(sean, classroom,}S) \Leftrightarrow \text{location(students, classroom,}S)$$

This way we can constrain things to be together.

Now it's easy to use the Situation Calculus to prove things like: when do students change from bored to not-bored? Or... what series of events are necessary in order to cause the students to leave the room? Etc. This is a simple form of **planning**—proving what sequence of events must occur to shift the current situation 0 into a situation that satisfies the features we would like.

What's *hard* to do in the Situation Calculus is to ask: when do students *not* change from bored to something else? The only way we could do this is to say something like this:

$$\forall S : \text{bored}(S) \Rightarrow \text{bored(do(talk)}, S)$$
$$\forall S : \text{bored}(S) \Rightarrow \text{bored(do(move(sean, left, right)}, S))$$
$$...$$

That is, we need to have a rule for *every* possible event that could happen, in order to state that the students are still bored even if that event occurs! You'd have to figure out how to say this for every situation! That makes for a *lot* of rules just to describe something simple. This difficulty is what is known as the **frame problem**. In the Situation Calculus (and in many forms of logic), it's easy to set up the logic to determine what *changes* as a result of certain actions—but it's much more expensive to set up the logic to determine what *doesn't change* as a result of certain actions.

This makes the Situation Calculus somewhat limited for planning tasks: part of this is because the Situation Calculus enables you to search through the space of *situations*, using events as transitions between them. More advanced planning approaches search through a different space: the space of *plans* consisting of sequences of events to perform.