https://www.cs.ubc.ca/~jordon/teaching/cpsc322/2019w2/

Lecture 20 - Bottom Up proof: soundness and completeness

March 3rd, 2020
https://www.cs.ubc.ca/~jordon/teaching/cpsc322/2019w2/lectures/lecture20.pdf

Goals:

- Prove that the bottom up proof system is **sound**
- Prove that the bottom up proof system in **complete**

Recap of soundness and completeness

Def: (Generic soundness of a proof procedure)
If G can be proven by the procedure ($KB \vdash G$) then G is logically entailed by the KB ($KB \models G$)

Def: Generic completeness of proof procedure
If G is logically entailed by the KB ($KB \models G$), then G can be proven by the procedure ($KB \vdash G$).

In other words:

- Everything **derived** from a **sound** proof procedure is **entailed** by the KB.
- Everything **entailed** by the KB can be **derived** a **complete** proof procedure

We had the algorithm for computing the set of consequences of KB:

$C := \{\}$

repeat:

select clause "$h \leftarrow b_1 \wedge ... \wedge b_m$" in KB such that $b_i \in C$ for all i, and $h \notin C$.
$C := C \cup \{h\}$

until no more clauses can be selected

So BU is sound if all of the atoms in C are logically entailed by KB

Soundness

Every atom in C is a logical consequence of KB.

Proof (by contradiction):
Suppose there exists an atom in C that is not a logical consequence of KB. If this is the case, let h be the first atom added to C that is not a logical consequence of KB. Let I be a model in which h is false.

Because h has been generated, there must be some definite clause of the form $h \leftarrow a_1 \wedge \ldots \wedge a_m$, such that $a_1, \ldots, a_m$ are all in C.

Because h is the first atom added to C that is not true in all models of KB, then all the a_i are generated before h are true in I. Thus it is a clause where the head is false but the body is true, and thus by the definition of truth clauses this clause is false in I. This is a contradiction to the fact that I is a model of KB. Thus every element of C is a logical consequence of KB.

Completeness

If G is logically entailed by the KB ($KB \models G$) then G can be proved by the procedure ($KB \vdash G$)

Proof:
Suppose that $KB \models G$. Then G is true in all models of KB. Thus G is true in any particular model of KB.

We will define a particular model such that if G is true in that model, G is proven by the bottom up algorithm. We will therefore define a particular interpretation I such that iff G is true in I, G is proved by the bottom-up algorithm. We will then show that I is a model. Thus we will define I such that if G is true in I, then $G \subseteq C$.

Let I be an interpretation where each element of C is **true**, and every other atom is **false**. We claim that I is a model of KB (which we'll call the minimal model)

> **Proof** (of claim):
> **Assume** that I is not a model of KB. **Then** there must exist a clause $h \leftarrow b_1 \wedge \ldots \wedge b_m$ in KB (having zero or more b_i's) which is **false** in I. The only way this an occur is if all of the b_i's are true in I (are in C) and h is false in I (not in C).
>
> But if each b_i belonged to C, bottom up would've added h to C as well. Therefore there can be no clause in KB that is false in interpretation I (which implies the claim)

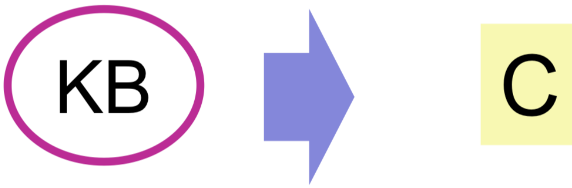
Lecture 21 - Domain Modeling and Top-Down Proofs

March 3rd & 5th, 2020
https://www.cs.ubc.ca/~jordon/teaching/cpsc322/2019w2/lectures/lecture21.pdf

Top-Down Proof Procedure

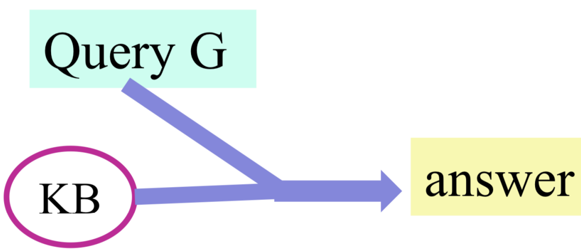
Bottom-up vs. Top-down

Bottom up starts with a knowledge base, and derives the set of consequences.



We have that G is proved, if at the end $G \subseteq C$. Therefore BU looks at the query only at the end.

The **key idea** of the top down proof system is to **search backwards** from a query G to determine if it can be derived from KB.



TD performs a backwards search, starting at G.

Top-Down Proof Procedure: Elements

Notation: An answer clause is of the form:

$$yes \leftarrow a_1 \wedge a_2 \wedge \ldots \wedge a_m$$

Express query as an answer clause (e.g.: if query $= a_1 \wedge \ldots \wedge a_m$), then this yields:

$$yes \leftarrow a_1 \wedge a_2 \wedge \ldots \wedge a_m$$

Rule of inference: (called SLD Resolution). Given an answer clause of the form

$$yes \leftarrow a_1 \wedge a_2 \wedge \ldots \wedge a_m$$

and the KB clause:

$$a_i \leftarrow b_1 \wedge b_2 \wedge \ldots \wedge b_p$$

You can generate the answer clause:

$$yes \leftarrow a_1 \wedge \ldots \wedge a_{i-1} \wedge b_1 \wedge b_2 \wedge \ldots \wedge b_p \wedge a_{i+1} \wedge \ldots \wedge a_m$$

Some examples are illustrated in the following table:

answer clause	KB clause	resulting inference
$yes \leftarrow b \wedge c$	$b \leftarrow k \wedge f$	$yes \leftarrow k \wedge f \wedge c$
$yes \leftarrow e \wedge f$	e ($e \leftarrow$)	$yes \leftarrow f$

(Successful) Derivations

An answer is an answer clause with $m = 0$. That is, it is the **empty** answer clause "$yes \leftarrow$"

A (successful) derivation of query "$?\, q_1 \wedge \ldots \wedge q_k$" from the KB is a sequence of answer clauses $Y_0, Y_1, \ldots, Y_n$ such that:

- Y_0 is the answer clause $yes \leftarrow q_1 \wedge \ldots \wedge q_k$
- Y_i is obtained by resolving Y_{i-1} with a clause in KB
- Y_n is the empty clause

Lecture 22 - TD as search, Datalog (variables)

March 5th, 2020
`https://www.cs.ubc.ca/~jordon/teaching/cpsc322/2019w2/lectures/lecture22.pdf`

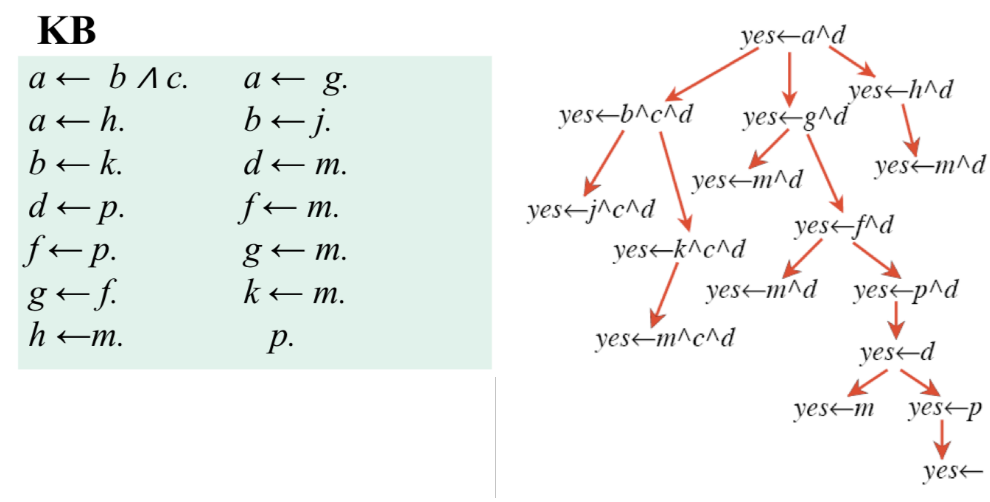
Top Down proof formulated as a search problem

We define the top down proof system as a search problem in the following way:

- **State** is an answer clause
- **Successor function** states resulting from substituting one atom with all the clauses of which it is the head.
- **Goal state** empty answer clause
- **Solution** start state
- **Heuristic function** ...

Search Graph

The following is what a search graph may look when we want to solve $yes \leftarrow a \wedge d$, given the following KB:



A **possible heuristic** could be the **number of atoms** in the answer clause.

However, we may also know that if the body of the answer clause contains a symbol that is not the head of *any* clause in the KB, then we know that the **most informative heuristic value** is ***zero***.

Datalog

Representation and Reasoning in Complex domains

In complex domains, expressing knowledge with **propositions** can be quite limiting:
up_s_2
up_s_3
ok_cb_1
ok_cb_2
$live\_w_1$
$connected\_w_1\_w_2$

It is therefore often natural to consider **individuals** and their **properties**.
$up(s_2)$
$up(s_3)$
$ok(cb_1)$
$ok(cb_2)$

$live(w_1)$
$connected(w_1, w_2)$

By using propositions we have no notion that for instance up_s_2 and up_s_3 share a meaning, or that $live\_w_1$ and $connected\_w_1\_w_2$ are about the same individual.

Consequently...

... we turn **propositions** into **relations** that are applied to individuals. We gain much by doing so. Notably:

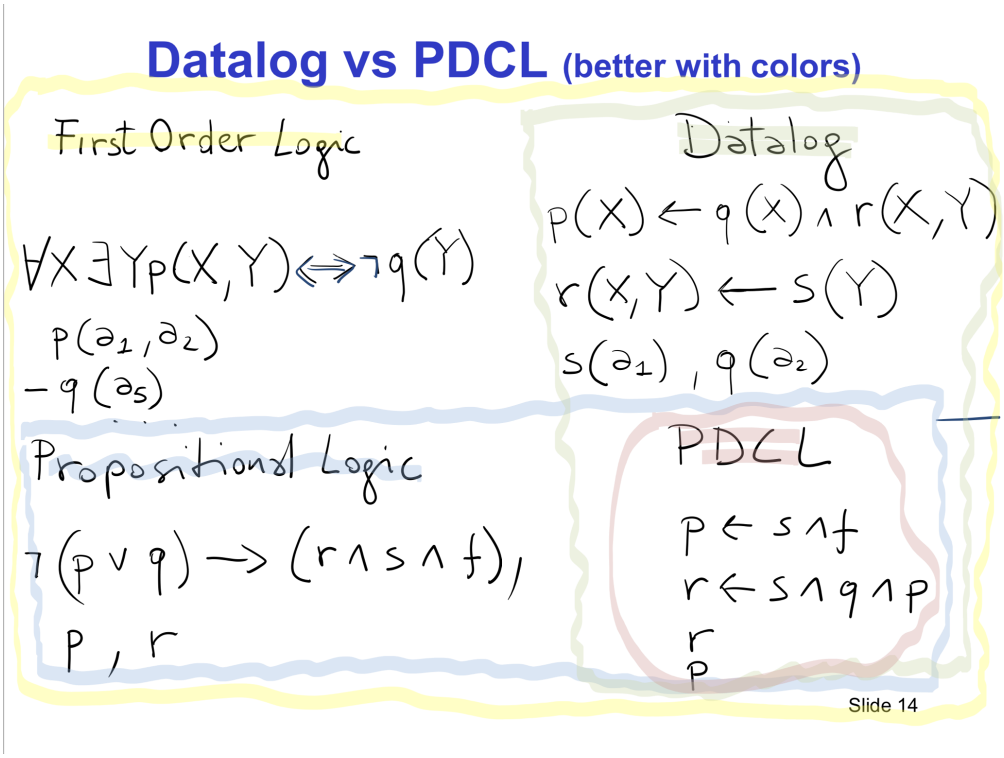
- We can express knowledge that holds for a set of individuals (by introducing variables):

$$live(W) \leftarrow connected\_to(W, W_1) \wedge live(W_1) \wedge wire(W) \wedge wire(W_1)$$

- We can **ask generic queries**:

$$? \quad connected\_to(W, w_1)$$

The following slide illustrates the relation between then different proof systems:



Datalog: a relational rule language

Datalog expands the syntax of PDCL(Propositional Definite clause logic). We have the following definitions, where a symbol (or word) is a sequence of letters, digits or an underscore _:

Def: A variable is a symbol starting with an upper case letter or with _ (Examples: X, Y)

Def: A constant is a symbol starting with a lower case letter or a sequence of digits, or is a number constant or a string (Examples: $alan, w_1$).

Def: A term is either a variable or a constant (Example: $X, Y, alan, w_1$)

Def: A predicate symbol is a symbol starting with a lower case letter. Constants and predicate symbols are distinguishable by their context in the knowledge base (Example: $live, part - of, connected, in$).

Def: An atom is a symbol of the form p or $p(t_1, \ldots, t_n)$ where p is a proposition or predicate symbol, and t_i are terms. Each t_i is referred to as an argument to the predicate. (Examples: $sunny, in(alan, X)$).

Def: A definite clause is either an atom (fact) or of the form:

$$h \leftarrow b_1 \wedge \ldots \wedge b_m$$

where h and the b_i are atoms (Read this as "h if b"). If $m > 0$, the clause is called a rule. If $m = 0$ the arrow can be omitted and the clause is called **atomic clause** or **fact**. An atomic clause has an **empty body**. Example:

$$in(X, Z) \leftarrow in(X, Y) \wedge part\_of(Y, Z)$$

Def: A knowledge base is a set of definite clauses.

Def: a query is of the form:

$$\textbf{ask} \quad a_1 \wedge \ldots \wedge a_m$$

Datalog: Top Down Proof Procedure

An extension of the top-down proof procedure can be applied to Datalog. The idea goes as follows:

- We find a clause in the KB whose *head* matches the query.
- Substitute variables in the clause with their matching constants

For instance consider the following KB (we replace $\wedge$ with `&`):

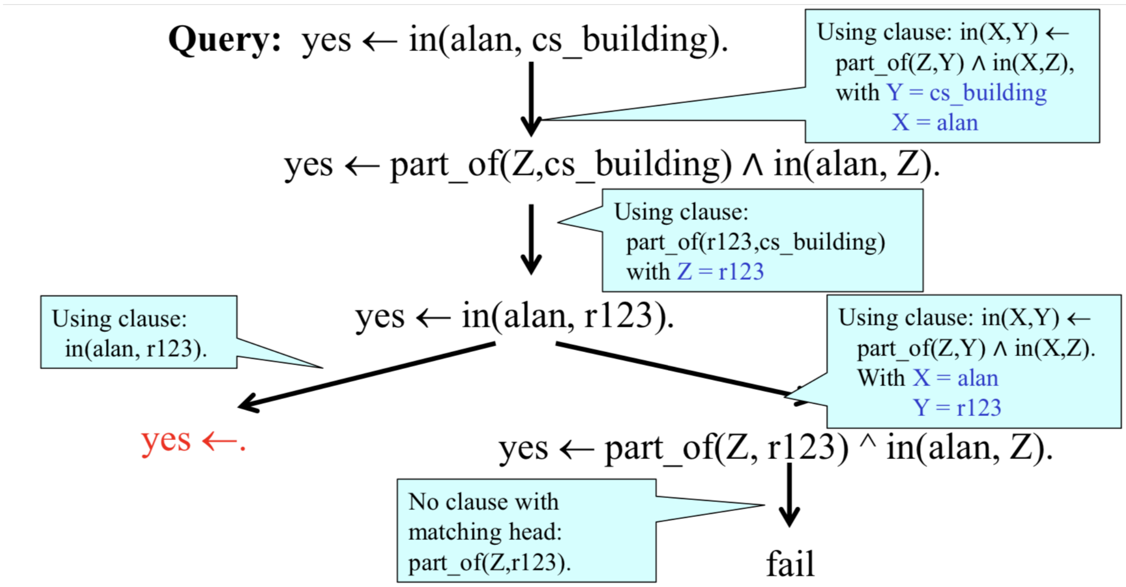
```
in(alan, r123)
part_of(r123, cs_building)
in(X,Y) <- part_of(Z,Y) & in(X,Z)
```

The if our query is "`yes <- in(alan, cs_building)`", then using our knowledge base this translates to:

```
yes <- part_of(Z,cs_building) & in(alan,Z)
```

A full sketch of the proof is shown in the following slide:

Jeremi Do Dinh
61985628



Datalog: queries with variables

Using our previous knowledge base, suppose now that our query is of the form:

```
in(alan, X1)
yes(X1) <- in(alan, X1)
```

What should the answer(s) be? **NOT QUITE SURE GO OVER**