Extending the Basic Reasoning System

CMPT 411/721
Topics

- Adding integrity constraints: Horn clauses
  - Assumption-Based Reasoning
- The closed world assumption
  - The Fitting operator
  - Datalog
- Adding disjunction
Beyond Definite Knowledge

- We begin first by considering two major extensions to the definite clause language:
  1. Add *integrity constraints* to definite clauses, giving *Horn clauses*.
  2. Adopt the *closed world assumption*, the assumption that our rules express *all* information about an atom.
Beyond Definite Knowledge

• We begin first by considering two major extensions to the definite clause language:
  1. Add *integrity constraints* to definite clauses, giving *Horn clauses*.
  2. Adopt the *closed world assumption*, the assumption that our rules express *all* information about an atom.

• Both extensions add a limited form of negation to our basic system.
  • Will later extend this further, in considering *answer set programming*.

• Following this we consider generalising the approach to effectively obtain propositional logic.
We now allow rules with the special atom \textit{false} (or: \(\bot\)), that is false in all interpretations, at the head of rules.

Clauses of the form

\[
\text{false} \iff a_1 \land \cdots \land a_k
\]

are called \textit{integrity constraints}.

Example: In the circuits domain, there is nothing to prevent a port being both on and off.

\[
\text{false} \iff \text{value}(X, \text{on}) \land \text{value}(X, \text{off})
\]
Extending the Basic Approach I:
Integrity Constraints and Horn Clauses

- We now allow rules with the special atom \textit{false} (or: \perp), that is false in all interpretations, at the head of rules.
- Clauses of the form \textit{false} \iff a_1 \land \cdots \land a_k are called \textit{integrity constraints}.
- A \textit{Horn clause} is a definite clause or an integrity constraint.
- Integrity constraints allow us to express that some combinations of atoms can’t all be true.
- That is, \textit{false} \iff a_1 \land \cdots \land a_k says that \( a_1, \ldots, a_k \) can’t all be true.
Extending the Basic Approach I: Integrity Constraints and Horn Clauses

• We now allow rules with the special atom $false$ (or: $\bot$), that is false in all interpretations, at the head of rules.

• Clauses of the form

\[ false \iff a_1 \land \cdots \land a_k \] are called integrity constraints.

• A *Horn clause* is a definite clause or an integrity constraint.

• Integrity constraints allow us to express that some combinations of atoms can’t all be true.

• That is, $false \iff a_1 \land \cdots \land a_k$ says that $a_1, \ldots, a_k$ can’t all be true.

• Example: In the circuits domain, there is nothing to prevent a port being both *on* and *off*.
  • With $false$ we can assert

\[ false \iff value(X, on) \land value(X, off) \]
Integrity Constraints and Horn Clauses

• Example:

\[ T_1 = \{ \text{false} \iff a \land b, \ a \iff c, \ b \iff c \} \]

• We conclude that \( c \) is \textit{false} in all models of \( T_1 \).

• In propositional logic this would be \( T_1 \models \neg c \).

  • Could also write this as \( T_1 \models \text{false} \iff c \).

\[ \text{Note that } \neg \text{ isn’t part of the KB language, so writing } \]
\[ T_1 \models \text{false} \iff c \text{ is better.} \]
Example (continued)

- Consider

\[ T_2 = \{ \text{false} \iff a \land b, \ a \iff c, \ b \iff d, \ b \iff e \} \]

- Write \( \alpha \lor \beta \) for a formula that is true in interpretation \( \mathcal{I} \) iff \( \alpha \) is true in \( \mathcal{I} \) or \( \beta \) is true in \( \mathcal{I} \) (or both).

Again, \( \lor \) isn’t a symbol in our object language.

- Given this notation we have:

\[ T_2 \models \neg c \lor \neg d \text{ and } T_2 \models \neg c \lor \neg e. \]

I.e. we have that

\[ T_2 \models \text{false} \iff c \land d \text{ and } T_2 \models \text{false} \iff c \land e. \]

- Note that we cannot handle unrestricted disjunctions and negations.

- However we can derive disjunctions of negations of atoms.
Reasoning with Horn Clauses

- We can use our previous top-down and bottom-up reasoners with Horn clauses.

- If $KB \models false$ then $KB$ is inconsistent.
  Example: $KB = \{\text{false} \leftarrow a., a.\}$.

- If the KB is consistent, then to derive (positive) atoms we can ignore integrity constraints. (Why?)

- However, we can exploit HC reasoning, as discussed next.
Assumption-Based Reasoning

The addition of integrity constraints seems minor; however it turns out to be a powerful tool.

- In many activities it is useful to know that some combination of truths are incompatible.
- Here we give an example in diagnosis.
- We will use the circuit example of the previous section.
  - Previously, given inputs, we could predict outputs.
  - For diagnosis, we may be given inputs, but the outputs may not be the expected outputs.
  - In this case we would like to prove what could be wrong with the circuit.
Assumption-Based Reasoning

- Define the **assumables** to be the atoms which we could accept as part of a (disjunctive) answer.
- Intuitively, assumables are things that we want to assume are true, if consistently possible.
  - In the circuit example, we will assume that a gate is not broken, where possible.
- If $T$ is a set of clauses, a *conflict* of $T$ is a set of assumables that, given $T$, imply *false*.
  - I.e. $C = \{c_1, \ldots, c_r\}$ is a conflict if
    \[
    T \models false \iff c_1 \land \cdots \land c_r
    \]
    that is,
    \[
    T \models \neg c_1 \lor \cdots \lor \neg c_r.
    \]
Assumption-Based Reasoning

- A minimal conflict is a conflict s.t. no subset is a conflict.
- Recall:

\[ T_2 = \{ \text{false} \iff a \land b, \ a \iff c, \ b \iff d, \ b \iff e \} \]

- In \( T_2 \), if \( \{c, d, e\} \) are the assumables, then \( \{c, d\} \) and \( \{c, e\} \) are minimal conflicts.
- The use of false in the head of a clause allows the possibility of a set of clauses being unsatisfiable.
  - Q: Show KB’s without integrity constraints are satisfiable.
Consistency-Based Diagnosis

Consider our circuit example from before.

- For the clauses involving how gates work, we add a predicate \( ok \) expressing that the gate is working.
- For \( \text{and} \) gates we have:

\[
\text{value}(\text{out}(D), \text{on}) \iff \text{gate}(D, \text{and}) \land \text{ok}(D) \\
\quad \land \text{value}(\text{in}(1, D), \text{on}) \\
\quad \land \text{value}(\text{in}(2, D), \text{on}).
\]

\[
\text{value}(\text{out}(D), \text{off}) \iff \text{gate}(D, \text{and}) \land \text{ok}(D) \land \text{value}(\text{in}(1, D), \text{off}).
\]

\[
\text{value}(\text{out}(D), \text{off}) \iff \text{gate}(D, \text{and}) \land \text{ok}(D) \land \text{value}(\text{in}(2, D), \text{off}).
\]
Example

• $ok(D)$ will be assumable.
• We add the clause

$$false \iff value(X, \text{on}) \land value(X, \text{off}).$$

• Given a set of observations (input and output) we want to ask whether there is a gate that is not $ok$:

$$\text{? } \neg ok(D)$$
Example

- We test our circuit by giving it the following inputs.

\[
\begin{align*}
\text{value}(\text{in}(1, \text{adder}), \text{on}), \\
\text{value}(\text{in}(2, \text{adder}), \text{off}), \\
\text{value}(\text{in}(3, \text{adder}), \text{on}), \\
\text{value}(\text{out}(1, \text{adder}), \text{on}), \\
\text{value}(\text{out}(2, \text{adder}), \text{off}).
\end{align*}
\]

- With these values, the circuit cannot be operating correctly.
Example

- There are two minimal conflicts:
  \[
  \{\text{ok}(x_1), \text{ok}(x_2)\}
  \]
  \[
  \{\text{ok}(x_1), \text{ok}(a_2), \text{ok}(o_1)\}
  \]

- Hence:
  - (At least) one of the exclusive-or gates is faulty.
  - One of the gates $x_1$, $a_2$, $o_1$ is faulty.

- We can distribute the answers to get the logically equivalent result:
  \[
  \neg\text{ok}(x_1) \lor (\neg\text{ok}(x_2) \land \neg\text{ok}(a_2)) \lor (\neg\text{ok}(x_2) \land \neg\text{ok}(o_1)).
  \]

- Each conjunction in this disjunction is called a \textit{diagnosis}. 
Implementation: Bottom-up algorithm

The bottom-up implementation is an augmentation of the bottom-up algorithm presented earlier.

- The conclusion is a set of pairs $\langle a, A \rangle$ where $a$ is an atom and $A$ is a set of assumables that together with the rules imply $a$.
- Initially the conclusion set $C$ is $\{ \langle a, \{ a \} \rangle \mid a \text{ is assumable} \}$.
- Rules can be used to form new conclusions:
  
  If there is a rule
  
  \[ h \Leftarrow b_1 \land \cdots \land b_m \]

  such that for each $i$ there is $A_i$ such that $\langle b_i, A_i \rangle \in C$, then add $\langle h, A_1 \cup \cdots \cup A_m \rangle$ to $C$.

- If we generate $\langle \text{false}, A \rangle$ we know the assumptions in $A$ form a conflict.
  
  So if $A = \{ a_1, \ldots, a_k \}$ then $T \models \neg a_1 \lor \cdots \lor \neg a_k$. 
A Bottom-up Procedure

First, we get rid of variables by \textit{grounding} all rules.

- That is, each rule is replaced by the set of its ground instances.
- We can do this here since we have a finite domain.
A Bottom-up Procedure

Algorithm:

\[ C := \{ \langle a, \{ a \} \rangle \mid a \text{ is assumable} \}; \]
repeat
\[
\text{choose } r \in T \text{ such that } \begin{align*}
    r \text{ is } & 'h \leftarrow b_1 \land \cdots \land b_m' \\
    \langle b_i, A_i \rangle \in C \text{ for all } i, \text{ and } \\
    A = A_1 \cup \cdots \cup A_m \text{ and } \\
    \langle h, A \rangle \notin C; \end{align*}
\]
\[ C := C \cup \{ \langle h, A \rangle \} \]
until no more choices
Example:

- Assume we have three and-gates, where the outputs from $a_1$ and $a_2$ are connected to the inputs of $a_3$.
- We observe that inputs $on/off/on/on$ give output $on$.
- Initially $C$ has the value:

$$\{ \langle ok(a_1), \{ ok(a_1) \} \rangle, \langle ok(a_2), \{ ok(a_2) \} \rangle, \langle ok(a_3), \{ ok(a_3) \} \rangle \}$$
The following shows a possible sequence of values added to $C$:

\[
\begin{align*}
\langle & \text{value}(in(2, a_1), \text{off}), \{\} \rangle \\
\langle & \text{gate}(a_1, \text{and}), \{\} \rangle \\
\langle & \text{ok}(a_1), \{\text{ok}(a_1)\} \rangle \\
\langle & \text{value}(out(a_1), \text{off}), \{\text{ok}(a_1)\} \rangle \\
\langle & \text{connected}(\text{out}(a_1), \text{in}(1, a_3)), \{\} \rangle \\
\langle & \text{value}(in(1, a_3), \text{off}), \{\text{ok}(a_1)\} \rangle \\
\langle & \text{gate}(a_3, \text{and}), \{\} \rangle \\
\langle & \text{ok}(a_3), \{\text{ok}(a_3)\} \rangle \\
\langle & \text{value}(out(a_3), \text{off}), \{\text{ok}(a_1), \text{ok}(a_3)\} \rangle \\
\langle & \text{value}(out(a_3), \text{on}), \{\} \rangle \\
\langle & \text{false}, \{\text{ok}(a_1), \text{ok}(a_3)\} \rangle
\end{align*}
\]

Thus we can prove $\neg \text{ok}(a_1) \lor \neg \text{ok}(a_3)$. 
Extending the Basic Approach II: Negation as Failure

- We can distinguish two types of “negative” situations with respect to trying to prove a query $G$:
  - We are able to show that $\neg G$ holds.
  - We are unable to show that $G$ holds.

- Sometimes for the second case we want to assume that $G$ is in fact false.

- This is known as *negation as (finite) failure* (naf).
Negation as Failure

• With our rule-based approach, we can justify naf if we assume that our rules express *all* knowledge about an atom.

• In this case, we can just store what is true, and so if we cannot derive something, it must be false.
  
  This is exactly the assumption made by relational databases.

• Thus an atom is false if none of the bodies implying the atom is true.
The Complete Knowledge Assumption

• For the ground case, consider where we have rules for atom \( a \):

\[
a \iff b_1 \\
\ldots \\
a \iff b_n
\]

• The Complete Knowledge Assumption says that if \( a \) is true then it must have been derived by one of the \( b_i \)'s.

• Hence one of the \( b_i \) must be true.

• I.e. \( a \Rightarrow b_1 \lor \cdots \lor b_n \), and thus

\[
a \iff b_1 \lor \cdots \lor b_n.
\]

• This is called the \textit{completion} of \( a \).
The Complete Knowledge Assumption

• For example, if

\[
\begin{align*}
\text{student} & \iff \text{grad} \\
\text{student} & \iff \text{ugrad}
\end{align*}
\]

then the completion is:

\[
\text{student} \iff \text{grad} \lor \text{ugrad}.
\]

• We won't go into it here, but this leads to a semantic account of the complete knowledge assumption (and negation as failure) known as the *Clark completion*. 
Implementation: Fitting Operator

- The bottom-up implementation incorporating naf is an extension of the procedure for definite clauses.
  - We now allow literals of the form $\sim p$ in the bodies of rules.
  - $\sim p$ expresses that $p$ \textit{finitely fails}.
    - I.e. $\sim p$ holds if we are unable to show that $p$ holds.
  - Can also add atoms of the form $\sim p$ to the set $C$ of consequences.

- From the complete knowledge assumption we have that:
  - The head atom of a rule must be true if the rule’s body is true.
  - An atom $p$ must be false if the body of each rule having $p$ as a head is false.
  - This leads to a three-valued model, in which atoms may be true, false, or undetermined.
- The Fitting operator can be implemented to run in linear time.
Implementation: Fitting Operator

- The bottom-up implementation incorporating naf is an extension of the procedure for definite clauses.
  - We now allow literals of the form $\sim p$ in the bodies of rules.
  - $\sim p$ expresses that $p$ finitely fails.
    - I.e. $\sim p$ holds if we are unable to show that $p$ holds.
  - Can also add atoms of the form $\sim p$ to the set $C$ of consequences.
- From the complete knowledge assumption we have that:
  - The head atom of a rule must be true if the rule’s body is true.
  - An atom $p$ must be false if the body of each rule having $p$ as a head is false.
- This leads to a three-valued model, in which atoms may be true, false, or undetermined.
- The Fitting operator can be implemented to run in linear time.
Example Rules

\[ p \iff q \land \neg r \]
\[ p \iff s \]
\[ q \iff \neg s \]
\[ r \iff \neg t \]
\[ t \]
\[ s \iff w \]
A Bottom-up Procedure:

\[ C := \{\}; \]
repeat
  either
    choose \( r \in A \) such that
    \( r \) is \( h \leftarrow b_1 \land \cdots \land b_m \)
    \( b_i \in C \) for all \( i \), and
    \( h \notin C \);
    \[ C := C \cup \{h\} \]
    or
    choose \( h \) such that for every rule
    \( h \leftarrow b_1 \land \cdots \land b_m \)
    either for some \( b_i \) we have \( \lnot b_i \in C \)
    or some \( b_i = \lnot g \) and \( g \in C \)
    \[ C := C \cup \{\lnot h\} \]
  until no more choices
Example

• Consider:
  
  \[ p \iff q \land \lnot r \]
  
  \[ p \iff s \]
  
  \[ q \iff \lnot s \]
  
  \[ r \iff \lnot t \]
  
  \[ t \]
  
  \[ s \iff w \]

• The following is a sequence of atoms added to \( C \):

  \[ t, \lnot r, \lnot w, \lnot s, q, p. \]
Top-down Procedure

The top-down procedure proceeds by \emph{negation as finite failure}.

- Consider:

  \[ a \leftarrow b_1 \]
  \[
  \vdots
  \]
  \[ a \leftarrow b_n \]

- If we try to prove each $b_i$ and fail each time, we can conclude that each $b_i$ is false, and so is $a$.

- See a text on logic programming for more.
Logic in Databases: Datalog

- Datalog is a database query language based on definite clauses with negation as failure.
- A Datalog program consists of a finite set of facts and rules.
- Facts are assertions about the world, such as “John is the father of Harry”.
- Rules are sentences which allow us to deduce facts from other facts.
  E.g. “If $X$ is a parent of $Y$ and if $Y$ is a parent of $Y$, then $X$ is a grandparent of $Y$”. 
Syntax

• Facts and rules are represented as Horn clauses of the form

\[ L_0 \leftarrow L_1, \ldots, L_n \]

where

• each \( L_i \) is a literal of the form \( P(t_1, \ldots, t_k) \)
• such that \( P \) is a predicate symbol and the \( t_i \) are terms.
• and a term is either a constant or a variable.
  🕊 So no functions

• E.g. \( gp(Z, X) \leftarrow par(Y, X), \ par(Z, Y) \)

• The left-hand side of a Datalog clause is called its \textit{head} and the right-hand side is called its \textit{body}.

• Clauses with an empty body represent facts.
Datalog and Relational Databases

Consider two sets of clauses:

- **Extensional database (EDB):** Set of relations (ground facts) stored in the database.
  - Corresponds to a standard relational database instance
- **Intentional database (IDB):** A set of rules where the head does not appear in the EDB.
  - The IDB represents *derived* relations.
  - Can be thought of as *views.*
Pure and Extended Datalog

• “Datalog” has slightly different meanings depending on the reference.
• For us, pure Datalog will be the language where rules are composed of positive (EDB and IDB) predicates only.
• The “standard” version of Datalog (which we will be using henceforth) adds to pure Datalog:
  • Built-in special predicate symbols such as $>$, $<$, $\geq$, $\leq$, $=$, $\neq$.
  • These symbols can occur only in the body of a rule.
  • E.g. $X < 100$, $X + Y + 5 > Z$
  • Negation as failure.
    • The symbol $\sim$ can precede any predicate symbol in the body of a rule.
    • E.g. $\text{ugrad}(X) \leftarrow \text{st}(X), \sim \text{grad}(X)$
Examples

- \( \text{ExpensiveProduct}(X) \leftarrow \text{Product}(X, C, P), \ P > 1000 \)
- \( \text{BritishProduct}(X) \leftarrow \text{Product}(X, C, P), \ \text{Company}(C, "\text{UK}" ) \)
- \( \text{StrictAbove}(X, Y) \leftarrow \text{Above}(X, Y), \sim\text{On}(X, Y) \)
Safety

• A **safe** Datalog program should always have a finite output
  • I.e., the relations defined by a Datalog program must be finite.
• A program \( P \) is safe if, for each rule in \( P \):

  Every variable that appears anywhere in the query
  must appear also in a relational, nonnegated atom in
  the body of the query.

• Unsafe rules:
  • \( Q(X, Y, Z) \leftarrow R(X, Y) \)
  • \( Q(X, Y, Z) \leftarrow R(X, Y), \ X < Z \)
  • \( Q(X, Y, Z) \leftarrow R(X, Y), \sim S(X, Y, Z) \)

  In each case an infinity of \( Z \)’s can satisfy the rule, even
  though \( R \) and \( S \) are finite relations.
Example:
Find employees participating in projects that don’t involve their department heads:

\[ X: \text{Employee} \quad P: \text{Project} \]
\[ H: \text{Department head} \quad N: \text{Department} \]

\[ \text{EmpInv}(X, P, H) \iff \text{Proj}(P, X, S, E, B, D), \text{Empl}(X, N), \text{Dept}(N, H) \]
\[ \text{DHInv}(X, P, H) \iff \text{Proj}(P, H, S, E, B, D), \text{Empl}(X, N), \text{Dept}(N, H) \]

\[ \text{Answer}(X) \iff \text{EmpInv}(X, P, H), \sim \text{DHInv}(X, P, H). \]
From Relational Algebra to Datalog

Selection: $\sigma_{X>10}(R)$

$S(X, Y) \leftarrow R(X, Y), \ X > 10$
From Relational Algebra to Datalog

Selection: \( \sigma_{X>10}(R) \)
\[ S(X, Y) \leftarrow R(X, Y), \ X > 10 \]

Projection: \( \Pi_{X,Y}(R) \)
\[ P(X, Y) \leftarrow R(X, Y, Z) \]
From Relational Algebra to Datalog

Selection: $\sigma_{X>10}(R)$

$S(X, Y) \leftarrow R(X, Y), \ X > 10$

Projection: $\Pi_{X,Y}(R)$

$P(X, Y) \leftarrow R(X, Y, Z)$

Cartesian Product: $R \times T$

$P(X, Y, Z, W) \leftarrow R(X, Y), \ T(Z, W)$
From Relational Algebra to Datalog

Selection: $\sigma_{X>10}(R)$

$S(X, Y) \Leftarrow R(X, Y), \ X > 10$

Projection: $\Pi_{X,Y}(R)$

$P(X, Y) \Leftarrow R(X, Y, Z)$

Cartesian Product: $R \times T$

$P(X, Y, Z, W) \Leftarrow R(X, Y), \ T(Z, W)$

Natural Join: $R \bowtie T$

$J(X, Y, Z) \Leftarrow R(X, Y), \ T(Y, Z)$
From Relational Algebra to Datalog

**Selection:** \( \sigma_{X>10}(R) \)

\[ S(X, Y) \Leftarrow R(X, Y), \ X > 10 \]

**Projection:** \( \Pi_{X,Y}(R) \)

\[ P(X, Y) \Leftarrow R(X, Y, Z) \]

**Cartesian Product:** \( R \times T \)

\[ P(X, Y, Z, W) \Leftarrow R(X, Y), \ T(Z, W) \]

**Natural Join:** \( R \bowtie T \)

\[ J(X, Y, Z) \Leftarrow R(X, Y), \ T(Y, Z) \]

**Theta Join:** \( R \bowtie_{R.X > T.Z} T \)

\[ J(X, Y, Z, W) \Leftarrow R(X, Y), \ T(Z, W), \ X > Z \]
From Relational Algebra to Datalog II

Intersection: \( R(X, Y) \cap T(X, Y) \)

\[ I(X, Y) \Leftarrow R(X, Y), T(X, Y) \]
From Relational Algebra to Datalog II

Intersection:  \( R(X, Y) \cap T(X, Y) \)

\( I(X, Y) \Leftarrow R(X, Y), T(X, Y) \)

Union:  \( R(X, Y) \cup T(X, Y) \)

\( U(X, Y) \Leftarrow R(X, Y) \)
\( U(X, Y) \Leftarrow T(X, Y) \)
From Relational Algebra to Datalog II

Intersection:  \( R(X, Y) \cap T(X, Y) \)

\[ I(X, Y) \leftarrow R(X, Y), \ T(X, Y) \]

Union:  \( R(X, Y) \cup T(X, Y) \)

\[ U(X, Y) \leftarrow R(X, Y) \]
\[ U(X, Y) \leftarrow T(X, Y) \]

Difference:  \( R(X, Y) \setminus T(X, Y) \)

\[ D(X, Y) \leftarrow R(X, Y), \ \sim T(X, Y) \]
Expressivity

- Datalog, as developed to this point, is as expressive as the *relational algebra*.
  - So Datalog can serve as a logical query language in a relational DB.
- If we include recursive definitions (next slide), it is *more* expressive than the relational algebra.
  - However, still not Turing complete.
E.g. Can define the notion of a *path* in a graph by:

\[ \text{path}(X, Y) \iff \text{edge}(X, Y) \]
\[ \text{path}(X, Y) \iff \text{path}(X, Z), \text{edge}(Z, Y) \]

Note that this corresponds with *transitive closure*, which cannot be expressed in first-order logic.

However, there may be problems with recursion when combined with negation as failure.

Example:

\[ P(X) \iff R(X), \sim Q(X) \]
\[ Q(X) \iff R(X), \sim P(X) \]
Solution: Stratified Datalog Programs

A Datalog program $P$ is *stratified* if

- there is an assignment $str$ of integers 0, 1, ... to the predicates $p$ of $P$ such that for each clause $r$ in $P$ the following holds:

  If $p$ is the predicate in the head of $r$ and $q$ a predicate in the body of $r$, then
  - $str(p) \geq str(q)$ if $q$ is positive, and
  - $str(p) > str(q)$ if $q$ is negative.

Example:

- $\text{check sensors} \leftarrow \text{signal error, signal error} \leftarrow \text{valve closed, } \sim\text{signal 1}$
- $\text{signal error} \leftarrow \text{pressure loss, } \sim\text{signal 2}$
- $\text{signal error} \leftarrow \text{overheat, } \sim\text{signal 3}$

Assign 1 to check sensors, signal error and 0 to other atoms. Stratification condition is satisfied.
Solution: Stratified Datalog Programs

- A Datalog program $P$ is *stratified* if
  - there is an assignment $str$ of integers 0, 1, ... to the predicates $p$ of $P$ such that for each clause $r$ in $P$ the following holds:
    - If $p$ is the predicate in the head of $r$ and $q$ a predicate in the body of $r$, then
      - $str(p) \geq str(q)$ if $q$ is positive, and
      - $str(p) > str(q)$ if $q$ is negative.

- Example:
  - $check\_sensors \leftarrow signal\_error$
    $signal\_error \leftarrow valve\_closed, \sim signal_1$
    $signal\_error \leftarrow pressure\_loss, \sim signal_2$
    $signal\_error \leftarrow overheat, \sim signal_3$
  - Assign 1 to $check\_sensors$, $signal\_error$ and 0 to other atoms.
    - Stratification condition is satisfied.
Stratified Datalog Evaluation Algorithm

- Evaluate IDB predicates lowest-stratum-first
- Once evaluated, treat them as EDB
- Continue with next stratum, etc.
Extending the Basic Approach III: Disjunctive Knowledge

- We extend the Horn clause language to allow full disjunctive and negative knowledge.
- E.g. if I know that either a friend or her spouse is picking me up at the airport, then I know that I have a ride, without knowing who will pick me up.
- We also allow the direct statement of negative information, rather than assuming negative instances as in negation as failure.
Disjunctive Knowledge and Negation as Failure

- Disjunctive knowledge is incompatible with negation as failure.
- E.g. Given $a \lor b$ we can’t prove $a$, and so can assume $\neg a$, and similarly for $b$.
- However $\neg a$, $\neg b$ is inconsistent with the original sentence.
• We add the following to our language:
  • A **literal** is an atom or the negation of an atom.
  • A **clause** has the form

\[ L_1 \lor \cdots \lor L_k \iff L_{k+1} \land \cdots \land L_n \]

where the \( L_i \) are literals.

• So for a clause,
  • if \( k = 1 \) and all the literals are atoms we have a definite clause.
  • if \( k = n \) we have a disjunction of literals.

• This has the same expressive power as propositional logic, but is syntactically restricted.
Semantics

- The meaning of clauses is given by the normal model-theoretic semantics, with the expected account for $\neg$ and $\lor$.
- Note that we can “move” literals over the $\iff$ sign.
  - I.e. we can “swap” a literal over the $\iff$ if we negate it.
- Hence any set of formulas in propositional logic can be written as a set of formulas of the form

  \[ P_1 \lor \cdots \lor P_k \iff P_{k+1} \land \cdots \land P_n \]

  where each $P_i$ is an atom.
Semantics

• Unlike general clauses, definite clauses have a unique representation, up to commuting the conjuncts.

• The normal form of a general clause is an equivalent clause with no literals on the right hand side of the $\iff$ sign.
  • That is, the normal form of
    \[ L_1 \lor \cdots \lor L_k \iff L_{k+1} \land \cdots \land L_n \]
    is
    \[ L_1 \lor \cdots \lor L_k \land \neg L_{k+1} \land \cdots \land \neg L_n \iff \]
    (where $\neg\neg P$ is replaced by $P$).
  • The $\iff$ can then be omitted.

• Our notion of a query and an answer remain the same.
  • So, an answer $answer$ means that for some $\vec{X}$, $answer(\vec{X})$ is a logical consequence of the clause set $C$. 
Example: Extended Circuit Diagnosis

- With the circuit diagnosis problem, there are some things that require disjunction.
- One is the single fault assumption, that says that there is only a single fault in the system.
  - This assumption allows some control over the combinatorial explosion of possible diagnoses.
  - It generalises to the $n$-fault assumption, for fixed $n$.
- For our circuit example we can express the single fault assumption as

  \[ ok(G_1) \iff \neg ok(G_2) \land G_1 \neq G_2. \]

- For the adder example, if inputs were \textit{on}/\textit{off}/\textit{on}, and outputs \textit{on}/\textit{off}, we could prove that there is only one fault, \( \neg ok(x_1) \).
Another way to reduce the combinatorial explosion of possibilities is to assume that gates break down in a limited number of ways.

This is the *limited failure assumption*.

For example we might assume that a gate can only be *ok* or stuck *on* or stuck *off*:

\[
\begin{align*}
ok(G) &\iff \neg\text{stuckOn}(G) \land \neg\text{stuckOff}(G) \\
\text{val(out}(G), on) &\iff \text{stuckOn}(G) \\
\text{val(out}(G), off) &\iff \text{stuckOff}(G)
\end{align*}
\]
Example (continued)

- So with limited faults and multiple observations we can prune the possible errors.
- Note finally that we can combine these (or other) assumptions.
- In the case where we arrive at an inconsistency with one or more assumptions, we would know that that assumption (or one of our assumptions) was wrong.