Lecture 38:

- Quiz 9
- More on Answer-Set Programming (ASP)

Announcements:

- HW-8 out
- Final Project
- Final Exam

Answer Sets

Rules:

`r :- p, q.` ... $r \leftarrow p \wedge q$

- `r` is the rule **head**
- `p, q` is the rule **body** ... read "r if p and q"

Informally, Answer Sets ...

1. Satisfy all rules of the program ...
   - if the answer includes the body it also includes the head
   - if the answer doesn't include the body, the rule is trivially satisfied
2. Do not contain contradictions (e.g., p and $\neg p$)
3. Are minimal ... answer set as small as can be
   - the *Rationality Principle*: "Believe nothing you are not forced to believe"
   - always forced to believe asserted facts

Answer Set Examples

Q: What are the answer sets?

```
q :- p.
r :- p.
s :- q, r.
p.
```

{p, q, r, s}

Q: What are the answer sets?

```
q :- p.
r :- q.
t :- r, s.
p.
t.
```

{p, t, q, r} ... note s is not in the answer!

Answer Set Examples

Q: What are the answer sets?

```
q :- p.
r :- q.
```

{} ... rationality principle! (minimal answers)

Disjunction (|)

Q: What are the answer sets?

```
p | q.
```

... asserting that p or q is true

2 answer sets: {p}, {q}

- $\{p, q\}$ is not a minimal answer
- thus p | q and $p \vee q$ are (slightly) different

Answer Set Examples

Q: What are the answer sets?

```
p.  q.                          ... two (propositional) facts
r | s :- p.
r | s :- q.
```

2 answer sets: {p,q,r}, {p,q,s} ... why not {p,q,r,s}?

- {p,q,r,s} is not minimal

Allows us to represent "incomplete" knowledge ...

```
sunny | raining :- umbrella.
```

if we use an umbrella it is sunny or raining (but don't know which)

Answer Set Examples

Default Negation (`not`) ... e.g., `p :- not q`

- what can't be proved I believe to be false ... *negation as failure*

Q: What are the answer sets?

```
u :- not v.          ... need umbrella if can't prove overcast
u :- r.              ... need umbrella if raining
u :- s.              ... need umbrella if sunny
```

{u} ... Why?

- {} doesn't satisfy rules since it makes `not v` true
- {u} satisfies all the rules
- {s,u} and {r,u} aren't minimal

Important: Negation-free programs have a unique answer set

- ... disjunction in the head of a rule is a form of negation!

Answer Set Examples

Q: What are the answer sets?

```
p :- not q.
q :- not p.
r :- p.
s :- p.
```

{p,r,s} and {q}

- First two rules same as: `p | q.`

Garfield Example:

```
full :- not hungry.
hungry :- not full.
happy :- full.
asleep :- full.
```

Q: What are the answer sets?

```
p :- not p.
r :- not q.
```

Unsatisfiable because of the first rule!

- {r} alone implies p, {r,p} doesn't support p
- the first rule "poisons" the entire answer set

More Terminology

Atoms and Literals:

- A proposition (e.g., p, q, r) is called an **atom** (for "atomic statement")
- A **literal** is an atom or its negation ... a positive or negative atom
- e.g., p is an atom, and both p and $\neg p$ are literals

In predicate logic, an atom is of the form:

$$p(t_1, t_2, \ldots, t_n)$$

- p is a **predicate**
- the **arrity** of p is n ... denoted p/n
- each t_i is a **term** ... a constant, variable, or "function"
- a proposition is just an atom with arrity 0

Terms without variables are **ground**, atoms with only ground terms are **ground**

- only ground facts are allowed in `clingo` (no variables)

Predicate Logic Basics

Predicate examples

```
1  node(a)                                  % monadic, constant a
2  edge(a,b)                                % binary predicate
3  child(luke,padme)                        % two constants
4  child(X,Y)                               % X, Y are variables
5  employee(alice,google,engineer)          % employee/3
6  employee("Alice","Google","Engineer")    % same but with strings
7  enrolled_in(bob,ssn(bob),326)            % ssn is a function
8  list(cons(1,cons(2,nil)))                % nested functions
```

For ground atoms, works similarly as propositions

Q: What is the answer set:

```
child(a,b).
child(b,c).
grandchild(a,c) :- child(a,b), child(b,c).
```

{child(a,b), child(b,c), grandchild(a,c)}

Predicate Logic Basics

Q: What are the answer sets:

```
buys(bob,honda) | buys(bob,toyota).
prefers(bob,red) | prefers(bob,silver).
prefers(bob,red) :- buys(bob,honda).
```

There are three "possible worlds" ...

- {buys(bob,honda), prefers(bob,red)}
- {buys(bob,toyota), prefers(bob,silver)}
- {buys(bob,toyota), prefers(bob,red)}

Computing Answer Sets – Dealing with `not` (default negation)

Assume a program $\mathbf{P}$ and ground atoms $\mathcal{A}$ (a possible answer set)

$\mathbf{P}^{\mathcal{A}}$ is obtained from $\mathbf{P}$ by: ... $\mathbf{P}^{\mathcal{A}}$ is called a **reduct**

1. removing all rules containing `not` a for $a \in \mathcal{A}$ ... a is an atom
2. removing all other body occurrences `not` b ... since $b \notin \mathcal{A}$

... $\mathbf{P}^{\mathcal{A}}$ is a *positive* program ... it has a unique answer set!

$\Rightarrow$ **$\mathcal{A}$ is an answer set of $\mathbf{P}$ if $\mathcal{A}$ is the (minimal) answer of $\mathbf{P}^{\mathcal{A}}$**

Example: Assume $\mathcal{A} = \{$`p`,`t`$\}$

Original program ($\mathbf{P}$)

```
q :- not p.
t :- p, not r.
p.
```

Rewritten (positive) program ($\mathbf{P}^{\mathcal{A}}$)

```
% removed
t :- p.
p.
```

{`p`,`t`} is an answer set of $\mathbf{P}$ since it is an answer set of $\mathbf{P}^{\mathcal{A}}$

Computing Answer Sets – Variables

1. Can only use variables **safely**:
   - `q(X) :- ..., not p(X), ...` only if `X` occurs in a positive body literal $^{(*)}$
   - `q(X) :- ...` only if `X` used (positively) in the rule body
2. Answer sets are computed over **ground programs** (i.e., no variables):
   - Each rule with variables is replaced by its **ground instantitations**
   - ... where each variable is replaced by a ground term in the program

Example: *Original program:*

```
p(a).  p(b).
q(X) :- p(X).
```

Ground program (with substitutions):

```
p(a). p(b).
q(a) :- p(a).
q(b) :- p(b).
```

$\Rightarrow$ `Clingo` has *two computation phases*: **grounding** followed by **solving**

Integrity Constraints

A **constraint** is a filter on *entire* answer sets:

:- ℓ_1, ℓ_2, ..., ℓ_n

An answer $\mathcal{A}$ satisfies the constraint if at least one $\ell_i \notin \mathcal{A}$

- in other words, the body of the constraint **must be** *false*
- if it is true, $\mathcal{A}$ is not an answer set

Q: What are the answer sets:

```
buys(alice,honda) | buys(alice,toyota).
prefers(alice,red) | prefers(alice,silver).
:- prefers(X,silver), make(X,honda).     % no silver honda's
```

Similar three as before (i.e., red toyota, silver toyota, or red honda)

Generating Guesses via Choice Rules

Choice rules provide specialized syntax for disjunction:

n_1 `{p(Y) : q(X,Y)}` n_2 `:- r(X).`

- for each ground *body*, { } generates a *set* of choices
- in this case, one `p(Y)` choice for each ground `q(X,Y)`
- makes k "guesses" from the set for $n_1 \leq k \leq n_2$ ... also other variants

Example: ... here using $= n$ variant

```
color(red). color(green). color(blue).
{node_color(X,C) : color(C)} = 1 :- node(X).
```

Result is an answer set for every possible assignment of a color to a node

- `{node_color(1,red), has_color(2,red), ...}`
- `{node_color(1,red), has_color(2,green), ...}`
- and so on

Aggregation

clingo also supports aggregation functions

- `#count`, `#sum`, `#min`, and `#max`

The basic syntax:

v_1 `rel`$_1$ `f { `t_1`:`L_1`; `t_2`:`L_2`; ...; `t_n`:`L_n`}` `rel`$_2$ v_2

where:

- `f` is one of the aggregate functions
- v_1 and v_2 are integer values (or variables)
- `rel`$_1$ and `rel`$_2$ are (optional) comparators (<, >, =, <=, >=, !=)
- t_i is a term "tuple" and L_i is a literal "tuple"
- aggregate function works **over the set** of term tuples

Aggregation

Max of the set of values `{3,2,1,4,5}`

```
#max {3; 2; 1; 4; 5}
```
evaluates to 5

The sum of the unique term "weights" `{1,1,2,2}` ... as a set $\{1, 2\}$

```
#sum {1:p; 1:q; 2:p; 2:q}
```
evaluates to 3 if p and q are true

Sums of the `T` values for unique `T,F` pairs (from xkcd example)

```
:- 1505 != #sum { T,F : total_order(F,N,T) }.
```

Number of unique `X,Y` pairs (to count the unique graph edges)

```
edges(N) :- N = #count { X,Y : edge(X,Y) }.
```

Number of posts `N` liked by employees `E` of Google

```
N = #count { P : likes(E,P) }, works_for(E,"Google")
```

Additional Features

Recursion: e.g., computing transitive closures ... trees & graphs (w/ cycles)

```
descendent(X,Y) :- child(X,Y).
descendent(X,Y) :- child(X,Z), descendent(Z,Y).
```

can also do: `descendent(X,Y) :- descendent(X,Z), descendent(Z,Y).`

Recursion through Negation ... Q: What does this program do?

```
diner(X) :- adjacent(X,Y), cafe(Y).
cafe(X) :- building(X), not diner(X).
```

Assigns campus buildings with cafes or diners such that:

1. no two adjacent buildings have a cafe
2. every building without a diner has a cafe
3. every building has one or the other (but not both)

Additional Features

Minimization and Maximization: find smallest and largest answers

```
1  % guess edges to delete and compute a new graph
2  { del(X,Y) } :- edge(X,Y).
3  edge_1(X,Y) :- edge(X,Y), not del(X,Y).
4  % compute all original and new graph paths
5  path(X,Y) :- edge(X,Y).
6  path(X,Y) :- path(X,Z), path(Z,Y).
7  path_1(X,Y) :- edge_1(X,Y).
8  path_1(X,Y) :- path_1(X,Z), path_1(Z,Y).
9  % ensure same path relation
10 :- path(X,Y), not path_1(X,Y).
11 % count edges, keep new graphs with fewest remaining edges
12 edge_cnt(N) :- N = #count {X,Y : edge_1(X,Y)}.
13 #minimize {N : edge_cnt(N)}.
```

Intervals and Pooling

Intervals: `size(1..3)` is replaced by `size(1) size(2) size(3)`

Pooling: `color(red ; blue)` is replaced by `color(red) color(blue).`

... can also combine, e.g.: `p(1,2 ; a,b)`

From Lecture 1: Course Overview

Deep dive into programming language (PL) design & implementation

- along the way, implement a "made up" typed, procedural language
- including parsing, type checking, and execution

General Course Goals:

- more programming experience
- understanding of how PLs (compilers/interpreters) work
- understanding of language design (syntax, types, constructs, trade-offs)
- (brief) exposure to different programming "paradigms"

From Lecture 1: Why ...

"What I cannot create, I do not understand" – Richard Feynman

Understanding how PLs work ...

- can help make you a better programmer
- can help decrease time needed to learn new languages
- techniques useful for wide range of software dev problems
- example of a more complicated/larger engineering problem
- essential part of computer science (curriculum, PLs as a subfield)

Studying PL concepts generally ...

- new ways to think about programming and problem solving tools
- better understand language and performance trade-offs