Answer Set Programming

Mark Law
mark.law09@imperial.ac.uk

4	3	1	2
2	1	3	4
3	2	4	1
1	4	2	3

Extended Constructs in ASP

In the previous lecture, we covered the stable model semantics for normal logic programs. The stable model semantics grew into the field that is now known as Answer Set Programming (ASP). For normal logic programs, Answer Sets are exactly the same as stable models; however, we will see in the course of this lecture that ASP also allows for much more. Not only does ASP allow an extended syntax in which constraints, aggregates, classical negation and optimisation statements are allowed; it has become a new declarative programming paradigm.

The idea of Answer Set Programming is that you encode a problem as a logic program, solve it for Answer Sets and then map these Answer Sets back to the solutions of the original problem. We will see that ASP can not only be used like this to solve simple problems such as sudoku, but can actually solve ILP tasks for their optimal solutions.

Constraints

- Constraints, are ways of filtering any unwanted Answer Sets

$:- b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n.$

- They are written as rules with an empty head (which means false).
- When computing the reduct, this empty head is replaced by $\bot$.

Constraints can be used as a way of ruling out Answer Sets which are not intended solutions of the problem we are trying to represent. At the head of a constraint, is the atom $\bot$. $\bot$ can never be in an Answer Set, and so if the body of a constraint is satisfied by an interpretation X, then X can not be an Answer Set (ground(P)^X contains $\bot$, but X doesn't).

In fact, any constraint:
$:- b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n.$
is equivalent to the normal rule:
$s :- b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n, not\ s.$
where s is an atom which does not occur in the rest of the program.

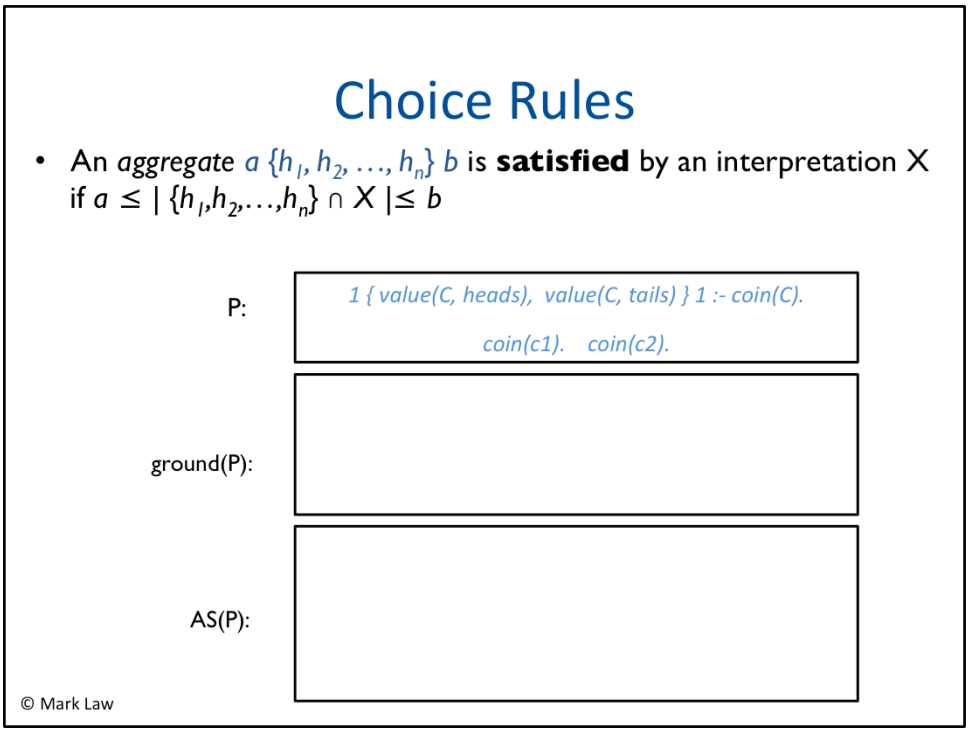
Constraints: example

- What are the Answer Sets of the program:

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

- What about:

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



Aggregates are a construct in ASP which allow the mapping of a set of atoms contained in an Answer Set to an integer. We will consider only the most common in this course – count. In this course, we will also restrict ourselves to rules in which aggregates are only allowed to appear in the head rather than the body. This particular kind of rule is called a *choice rule*.

Counting aggregates are useful when we want to express that there is a choice. For instance, the rule:
1 { value(Coin, heads), value(Coin, tails) } 1 :- coin(Coin).
expresses that every coin takes either the value “heads” or “tails” (but not both).

Choice Rules : Semantics

- We can compute the Answer Sets of a program *P* containing aggregates in the heads of rules by inserting a final step into the computation of the reduct P^X. For each rule *R* with an aggregate as the head:
  1. If the aggregate is not satisfied by *X* then we remove the head, converting *R* into a constraint.
  2. If the aggregate is satisfied then we generate one rule for each atom *A* in the aggregate which is also in *X*, with *A* at the head.

P: `1 { p, q } 1 :- r.` *X = { p, q, r }*

`r.`

P^X: `⊥ :- r.`

`r.`

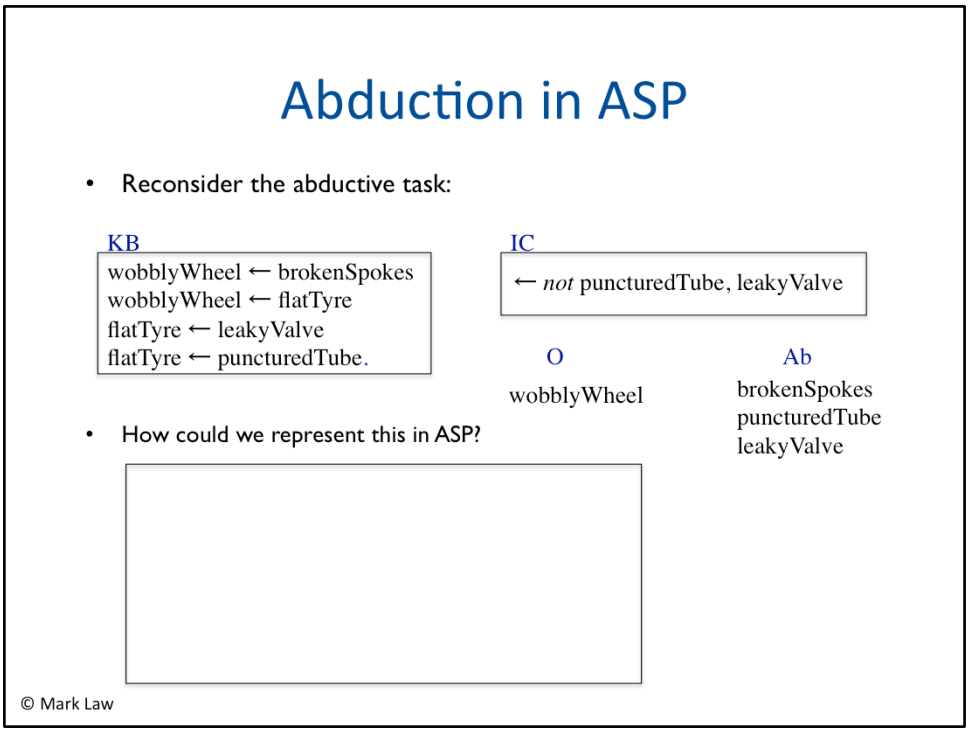
For the subclass of rules with aggregates we use, this description of the semantics of aggregates coincides with that described in *Answer Set Solving in Practice (Gebser et al. 2012)*. The restriction to rules only containing aggregates in the head allows us to simplify these definitions. When constructing the reduct of a ground program containing a rule *R* with an aggregate A as its head with respect to an interpretation *X*, we first apply the two steps for computing the reduct of a normal program (treating *A* as if it were an atom).
If *A* is not satisfied with respect to the interpretation *X*, we remove the rule *R*. If *A* is satisfied with respect to *X*, then we replace *R* with one rule for each atom in *A* which is true in *X* (where this atom becomes the head of the rule).

Choice Rules : Semantics

- We can compute the Answer Sets of a program *P* containing aggregates in the heads of rules by inserting a final step into the computation of the reduct P^X. For each rule *R* with an aggregate as the head:
  1. If the aggregate is not satisfied by *X* then we remove the head, converting *R* into a constraint.
  2. If the aggregate is satisfied then we generate one rule for each atom *A* in the aggregate which is also in *X*, with *A* at the head.

P: `1 { p, q } 2 :- r.` `r.` *X = { p, q, r }*

P^X: `p :- r.` `q :- r.` `r.`



We can represent any abductive task as an ASP program similarly to the one on this slide. The choice rule here (with no body) *generates* Answer Sets in which each of the abducibles is true. We then *test* each Answer Set using the integrity constraints and the constraint which says that the observation *wobblyWheel should* be true.

This structure to an ASP program is called *generate and test* and is used a lot when modeling using ASP. The idea is that you generate lots of Answer Sets and then use constraints to test that they really are solutions to the problem you are solving. Although, as previously mentioned in the course, generate and test is thought to be a naïve approach to ILP, the ASP solver is not actually doing a generate and test search underneath; in fact, we could devote many lectures (possibly even an entire course) to studying the different algorithms employed by ASP solvers. This one of the huge advantages to ASP, although the solver is doing some very clever things underneath, we do not see them. We write a program in a simplistic generate and test structure and let the solver do the hard work. Imagine writing the same program in Prolog; you would have to encode the logic of the search into the program.

We will see in the next lecture that ASPAL uses a very similar approach to solve an ILP task.

Optimisation Statements

- Consider *P*:

```
1 { value(C, heads),  value(C, tails) } 1 :- coin(C).

:- value(c1, X), value(c2, X)

coin(c1).       coin(c2).
```

- Its Answer Sets are:

```
{ coin(c1), coin(c2), value(c1, heads), value(c2, tails) },
{ coin(c1), coin(c2), value(c1, tails), value(c2, heads) }
```

- What do you think the optimisation statement below does?

```
#minimize [  value(c1, heads)=1,     value(c2, heads)=2  ]
```


Optimisation statements are useful when using ASP to model problems. We can set up the ASP program to have Answer Sets corresponding to the problems solution, and then use optimisation statements to give an ordering over the Answer Sets specifying which Answer Sets of the program are preferred over others.
ASP solvers such as clingo are then able to find optimal Answer Sets of the program, which correspond to the optimal solutions of the original problem.

Optimisation Statements

- Optimisation statements are of the form:

  $\#minimize\ [a_1=w_1,\ldots, a_n=w_n]$ or $\#maximize\ [a_1=w_1,\ldots, a_n=w_n]$

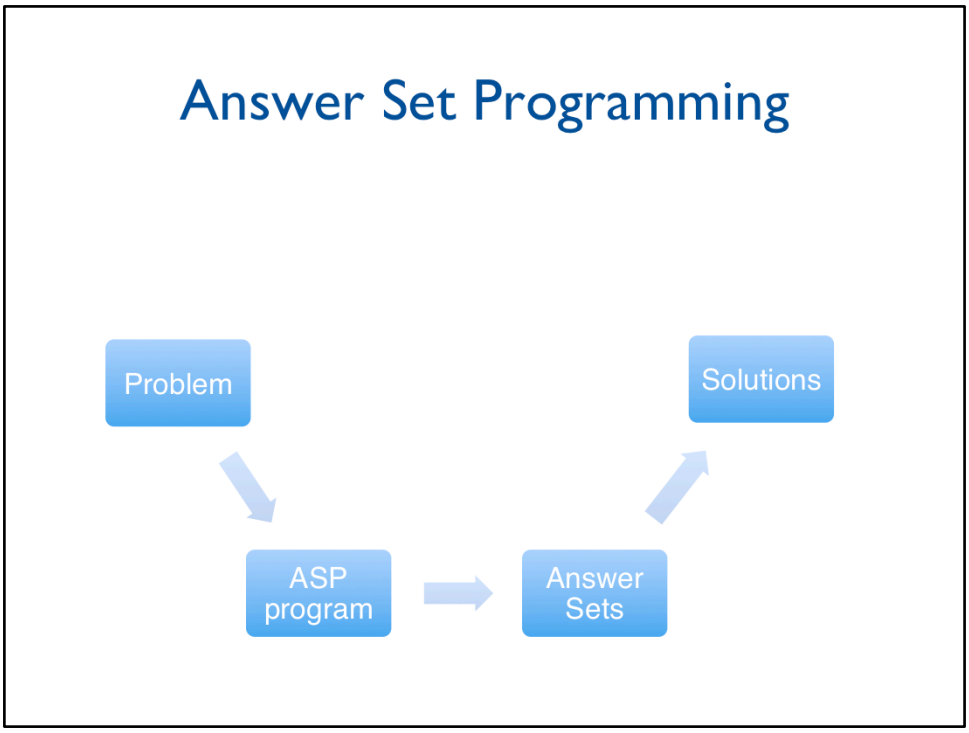
- The *w*'s are integer weights, and the a's are ground atoms.

- The solvers will search for **optimal** Answer Sets which *maximize* or *minimize* the weighted sum of the atoms.

ASP solvers allow the use of multiple optimisation statements with different priorities, but in this course we will consider ASP programs with only one optimisation statement. These optimisation statements are sets of ground atoms with weights.
They map Answer Sets to the weighted sum of the atoms which are true. If the statement is a minimize statement then the optimal Answer Sets are then those with the lowest sum. If the statement is a maximize statement then the optimal Answer Sets are those with the highest sum.

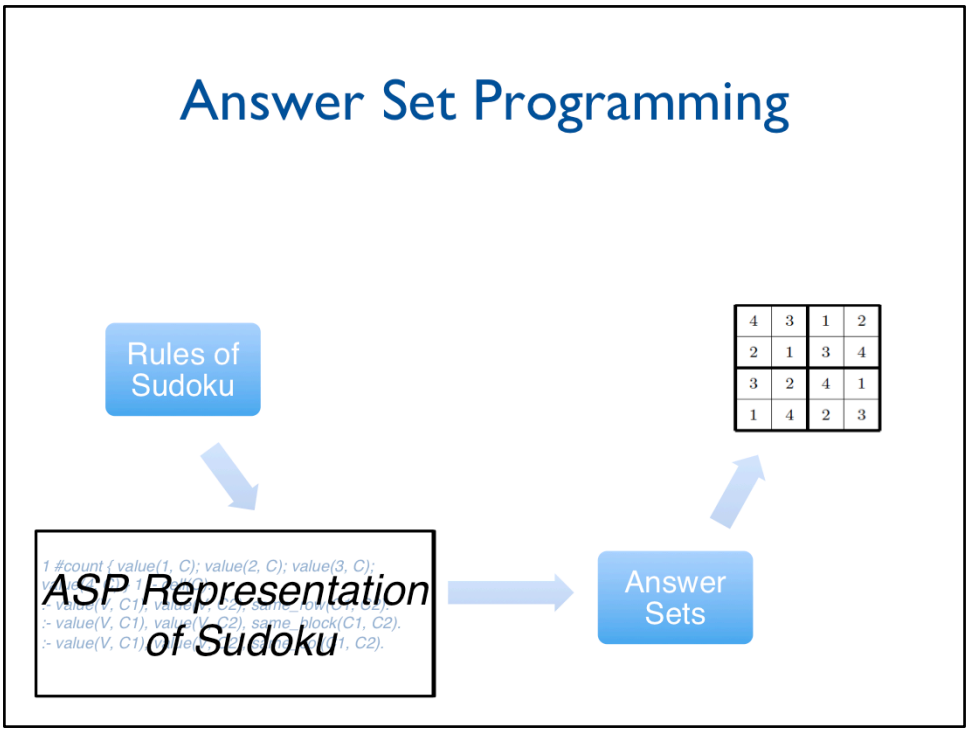
You should note that modern ASP solvers also allow for *weak constraints*. We will not consider these in this course, but they are equivalent to optimisation statements.

Modelling in ASP



The Answer Set Programming paradigm is to translate the problem we want to solve into an Answer Set Program such that when we solve the program for Answer Sets, these Answer Sets can then be translated back into solutions to the original problem. In the next few slides we will see such a translation for the solutions of sudoku.

Answer Set Programming



Sudoku

4	3	1	2
2	1	3	4
3	2	4	1
1	4	2	3

```
number(1..4).
cell(cell(X, Y)) :- number(X), number(Y).

same_row(cell(X1, Y), cell(X2, Y)) :-
        cell(cell(X1, Y)),  cell(cell(X2, Y)),  X1 != X2.

same_col(cell(X, Y1), cell(X, Y2)) :-
        cell(cell(X, Y1)),  cell(cell(X, Y2)),  Y1 != Y2.

same_block(cell(1, 1), cell(1, 2)).
same_block(cell(1, 1), cell(2, 1)).
same_block(cell(1, 1), cell(2, 2)).
...
```

Sudoku

4	3	1	2
2	1	3	4
3	2	4	1
1	4	2	3

```
1 {value(C, 1), value(C, 2), value(C, 3), value(C, 4)} 1 :- cell(C).

:- value(C1, V), value(C2, V), same_row(C1, C2).
:- value(C1, V), value(C2, V), same_col(C1, C2).
:- value(C1, V), value(C2, V), same_block(C1, C2).
```

In the next few lectures, we will see how we can learn this ASP program!

Further Reading

- *Knowledge Representation, Reasoning and Declarative Problem Solving*
  – 2003, Cambridge University Press, Baral, C.
- Knowledge Representation, Reasoning, and the Design of Intelligent Agents
  – 2012, Gelfond, M. Gelfond, Khal, Y.
- http://www.cs.utexas.edu/~vl/teaching/388L/clingo_guide.pdf
  – 2010, Gebser et al
- Knowledge Representation
  – Course 491

Summary

- Looked at the paradigm of Answer Set programming.
- Defined the concepts of:
  - Constraints
  - Aggregates (Choice rules)
  - Optimisation statements
- Modelling in ASP
  - Abduction in ASP
  - Solving sudoku in ASP