ASP (Answer Set Programming) Solvers v1.0b

Marek Sergot
Department of Computing
Imperial College, London

Feb 2010

There are various efficient (very efficient) implementations of systems for computing stable models and answer sets, employing splitting sets and other tricks and optimisations. Further development is an active area of current research, usually falling under the label of 'Answer Set Programming' (ASP).

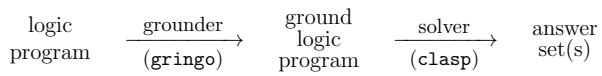
One well known system is Smodels. See `http://www.tcs.hut.fi/Software/smodels/`.

The system DLV is also widely used though I know little about it myself.
See `http://www.dbai.tuwien.ac.at/proj/dlv/`.

A state-of-the-art system that is becoming widely used is Clasp.
`http://www.cs.uni-potsdam.de/clasp`.
Clasp is installed on the DoC lab machines (see course web pages) and will (probably) be used as part of the assessed coursework.

Clasp: `gringo`, `clasp` and `clingo`

In common with most ASP systems, the computation of answer sets in Clasp has two phases: first, a *grounder* constructs ground instances of the clauses in the program, replacing variables by ground (variable-free) terms, possibly with some further transformations and simplifications. Then, the resulting ground program (usually in a special, machine-readable format) is passed to the actual *solver* which computes the answer sets.



In the Clasp system the grounder is called `gringo`; `clasp` is the solver itself.
Clasp also provides a utility `clingo` which calls `gringo` and pipes its output to the solver `clasp` automatically.

`lparse` is another widely used grounder. It is the grounder that comes with the Smodels and DLV systems. (The `clasp` solver can also take `lparse` outputs.)

The grounder (`gringo`)

The grounders used in ASP systems are very clever! There is much ingenuity here. They do much more than merely generating all ground instances of the clauses in naive fashion. From the Clasp manual: 'The main task of a grounder is to substitute the variables in a logic program by terms such that the result is a *finite* equivalent ground program. (Two programs are equivalent if they have the same answer sets.) Of course, a necessary condition for this is that the program possesses only finitely many finite answer sets.'
So, one can do various tricks during grounding, as long as they preserve answer sets.

Simple example

$$q(0, f(0)).$$
$$p(X) \leftarrow q(X, Y),\ \text{not } p(Y).$$

This program has an infinite Herbrand base (because it has terms 0, $f(0)$, $f(f(0))$, ...) and so infinitely many ground instances of its clauses. However, it is equivalent to the finite ground program

$$q(0, f(0)).$$
$$p(0) \leftarrow q(0, f(0)),\ \text{not } p(f(0)).$$

and has a single, finite answer set $\{q(0, f(0)),\ p(0)\}$. `gringo` can handle cases like this.

More generally, `gringo` deals with a special syntactic category of logic programs called

level-restricted logic programs

A level-restricted logic program is easy to detect syntactically, and has the property that it is equivalent to a finite ground program. Details of the definition are omitted here — they are not difficult but I did not have time to cover. Look at the Clasp documentation if you are interested.

Maximal stratified sub-programs $P' \subseteq P$ is a *stratified sub-program* of P if P' is stratified and no predicate p in P' is the head of any clause in $P - P'$.
A stratified sub-program $P' \subseteq P$ is *maximal* if every stratified sub-program of P is contained in P'.
From the Clasp manual: '...every logic program has a maximal stratified sub-program, which is also easy to determine.'

So, besides doing other stuff, `gringo`

- identifies the maximal stratified sub-program P' of P (I don't know how this is done but apparently it is easy);
- calculates the (unique) answer set of P' (no search required here — the iterated fixpoint method will do it — and this can be very efficient);
- uses the answer set of P' to simplify/ground the 'top' part $P - P'$, similarly to the way we used splitting sets to ground/simplify programs in earlier examples.

Note finally that the maximal stratified sub-program trick is not enough by itself: we still need the level-restricted constraint, or something similar.

Example (from the CLASP manual)

$$nat(0).$$
$$nat(s(X)) \leftarrow nat(X).$$

This program is stratified (trivially) so it has a unique answer set (obviously). But this answer set is not *finite*: `gringo` can't handle it.

It is often very instructive to see how much work `gringo` does when grounding a program. You can see the `gringo` output in readable form by running it with the `-t` option. See the brief instructions provided with the DoC installation (details on course web pages).

Answer Set Programming

This is a *huge* area. There are whole courses devoted to it. It is a kind of *constraint solving* paradigm. The program expresses the constraints; its answer sets are the solutions.

And ASP solvers can deal with much richer languages than (extended) logic programs:

- extended logic programs — almost always with the built-in consistency constraint
$$\leftarrow p, \neg p \qquad \text{(every atom } p\text{)}$$
- disjunctive logic programs
$$A \,;\, B \leftarrow L_1, \ldots, L_m$$
Note that the disjunction $A \,;\, B$ (sometimes written $A \mid B$) has a *special meaning* in disjunctive logic programs. (Maybe more on this later.)
- 'integrity constraints'
$$\leftarrow L_1, \ldots, L_m \qquad \Longrightarrow \qquad f \leftarrow L_1, \ \ldots, L_m, \ \text{not } f$$
- choice rules
$$\{ A_1, \ldots, A_m \} \leftarrow L_1, \ldots, L_n$$
- cardinality constraints
$$k_{min} \{ A_1, \ldots, A_m \} k_{max} \leftarrow L_1, \ldots, L_n$$
- weight constraints
- aggregation operators
- optimisation constructs
- ⋮

And more!

Constraints

$$\leftarrow L_1, \ldots, L_m$$

From the `clingo` guide: 'integrity constraints eliminate answer set candidates. They are merely tests that discard unwanted answer sets. That is, there are no answer sets that satisfy all literals in an integrity constraint.'

Integrity constraints are also sometimes written in the form:

$$\bot \leftarrow L_1, \ldots, L_m$$

($\bot$ is 'falsum' — always false).

As you have seen, an integrity constraint $\leftarrow L_1, \ldots, L_m$ can be encoded by means of the rule:

$$f \leftarrow L_1, \ \ldots, L_m, \ \text{not } f$$

where f is any atom not appearing anywhere else in the program.
(The same atom f can be used to encode all the integrity constraints, as is easy to check.)

Choice rules

$$\{ A_1, \ldots, A_m \} \leftarrow L_1, \ldots, L_n$$

Informally: if literals $L_1, \ldots, L_n$ are satisfied, the answer set must contain a subset (possibly empty) of literals $\{ A_1, \ldots, A_m \}$.

Example:

$$\{a, b, c\} \leftarrow$$

has answer sets: $\{\}, \ \{a\}, \ \{b\}, \ \{c\}, \ \{a, b\}, \ \{b, c\}, \ \{a, c\}, \ \{a, b, c\}$.

Example

$$\left.\begin{array}{l} \{a, b, c\} \leftarrow \\ \leftarrow b, c \\ \leftarrow a, c \end{array}\right\} \qquad \text{has answer sets:} \qquad \{\}, \ \{a\}, \ \{b\}, \ \{c\}, \ \{a, b\}$$

Example

$$\left.\begin{array}{l} \{a, b\} \leftarrow \\ \leftarrow \text{not } a, \ \text{not } b \end{array}\right\} \qquad \text{has answer sets:} \qquad \{a\}, \ \{b\}, \ \{a, b\}$$

and so is an encoding of the 'inclusive or' $a \vee b$.

Cardinality constraints

$$k_{min}\ \{\ A_1, \ldots, A_m\ \}\ k_{max}\ \leftarrow\ L_1, \ldots, L_n \qquad (k_{max} \text{ optional})$$

Informally: if literals $L_1, \ldots, L_n$ are satisfied, the answer set must contain at least k_{min} and at most k_{max} of the literals $\{\ A_1, \ldots, A_m\ \}$. (The upper bound k_{max} is optional.)

Examples:

$\{\ 1\ \{a,b,c\}\ 2 \leftarrow\ \}$	has answer sets:	$\{a\},\ \{b\},\ \{c\},\ \{a,b\},\ \{b,c\},\ \{a,c\}$
$\{\ 1\ \{a,b,c\}\ 2 \leftarrow\ \}$	has answer sets:	$\{a\},\ \{b\},\ \{c\}$
$\{\ 1\ \{a,b\}\ \leftarrow\ \}$	has answer sets:	$\{a\},\ \{b\},\ \{a,b\}$

and so is another encoding of the 'inclusive or' $a \vee b$.

Choice rules — encoding

Choice rules are easily represented in extended logic programs. It is easiest to see with an example:

$$a \leftarrow \text{ not } a' \qquad a' \leftarrow \text{ not } a$$
$$b \leftarrow \text{ not } b' \qquad b' \leftarrow \text{ not } b$$

This program can obviously be split into two independent components: take $\{b, b'\}$ as the splitting set, say.
The 'top' component has two answer sets: $\{a\}$, $\{a'\}$.
The 'bottom' component has two answer sets: $\{b\}$, $\{b'\}$.
So the two components together (all four clauses) have answer sets:

$$\{a', b'\},\ \{a, b'\},\ \{a', b\},\ \{a, b\}$$

If we 'hide' the auxiliary atoms a' and b' in the output, we see answer sets:

$$\{\},\ \{a\},\ \{b\},\ \{a, b\}$$

Obviously this can be generalised.

A useful special case – 'a (inclusive) or b'.
As a cardinality constraint:

$$1\ \{a, b\} \leftarrow$$

equivalently, choice rule plus integrity constraint:

$$\{a, b\} \leftarrow$$
$$\leftarrow \text{ not } a,\ \text{not } b$$

can be represented:

$$a \leftarrow \text{ not } a' \qquad a' \leftarrow \text{ not } a$$
$$b \leftarrow \text{ not } b' \qquad b' \leftarrow \text{ not } b$$
$$f \leftarrow \text{ not } a,\ \text{not } b, \text{not } f$$