491 Knowledge Representation

Splitting Sets

Marek Sergot
Department of Computing
Imperial College, London

February 2003 v1.0i

Lifschitz, V. and Turner, H. Splitting a logic program. In P. van Hentenryck (ed.) *Logic Programming: Proc. Eleventh International Conference*. MIT Press 1994, 23–37.

The idea of a *splitting set* was introduced by Lifschitz and Turner as a generalisation of the stratification of logic programs.
(Splitting sets are actually defined for the more general class of *disjunctive* logic programs, but I will stick here to normal logic programs.)

Splitting sets also work in exactly the same way for extended logic programs and 'answer sets' (next topic).

Thanks to Dorian Gaertner (MEng4, 2003-04) for pointing out some typographical errors and small mistakes in an earlier version of these notes, to Timos Antonopoulos (MEng4, 2002-03) for correcting a mistake in the example at the end of the notes, and to Silvia Dobrota and Razvan Rosie (MEng4, 2013-14) for pointing out a couple of typographical errors.

Terminology

Nothing new here, but just for ease of reference:
A *normal logic program* is a set of *clauses* (also referred to as 'rules') constructed from the ground atoms of a (many-sorted) first-order language.
A *literal* is an atom or an atom preceded by the negation-as-failure symbol `not` . A *clause* is an expression of the form $A \leftarrow B_1, \ldots, B_n \quad (n \geq 0)$ where A is an atom and each B_i is a literal. The symbol $\leftarrow$ is usually omitted when $n = 0$. When B_i is an atom it is a *positive literal* of the clause and a *negative literal* of the clause otherwise. A is the *head* of the clause.
For a clause r, $head(r)$ denotes the head, and $body^+(r)$ and $body^-(r)$ the atoms in the positive and negative literals of r respectively. $atoms(r)$ is the set of atoms in r. Thus, $atoms(r) = \{head(r)\} \cup body^+(r) \cup body^-(r)$ (not necessarily disjoint).
A *program* P is a set of clauses. The set of atoms that occur in the clauses of P is denoted $atoms(P)$, which is the union of $atoms(r)$ for all clauses r in P. A *definite* clause r is one in which there are no occurrences of negation-as-failure `not` , in other words, one in which $body^-(r) = \emptyset$; a definite program is one containing definite clauses only.
A clause may contain *variables*, in which case it is regarded as shorthand for the set of all its ground instances.

Splitting sets

Definition A *splitting set* for a program P is any set of atoms U such that, for every clause $r \in P$, if $head(r) \in U$, then $atoms(r) \subseteq U$. The set U is also said to *split* P.
The set of clauses $r \in P$ such that $atoms(r) \subseteq U$ is the *bottom* of P relative to U, denoted by $bot_U(P)$. The set $top_U(P) = P - bot_U(P)$ is the *top* of P relative to U.

Note that $\emptyset$ and $atoms(P)$ are always splitting sets for a program P. But they are *trivial* splitting sets — of no interest.
Suppose we say that an atom A is *defined* by the rules r with $head(r) = A$.

		$top_U(P)$	$\Leftarrow$	there can be atoms here that are in U, but only in the *bodies* of rules, not in the heads
all atoms in here are in U	$\Rightarrow$	$bot_U(P)$	$\Leftarrow$	all atoms in U are either defined here, or are not defined anywhere in P

Examples

$$\begin{array}{l} a \leftarrow b, \texttt{not } c \\ b \leftarrow c, \texttt{not } a \\ \hline c \end{array} \qquad U = \{c\}$$

$$\begin{array}{l} c \leftarrow a \\ c \leftarrow b \\ \hline a \leftarrow \texttt{not } b \\ b \leftarrow \texttt{not } a \end{array} \qquad U = \{a, b\}$$

$$\begin{array}{l} p \leftarrow q, \texttt{not } r \\ p \leftarrow \texttt{not } q, s \\ \hline q \end{array} \qquad U = \{q, r, s\}$$

The last example has other (non-trivial) splitting sets. $\{q, r\}$, $\{q, s\}$ and $\{q\}$ are also splitting sets for the same split shown above. And $\{r, s\}$, $\{r\}$, and $\{s\}$ are splitting sets that split the program as follows:

$$\begin{array}{l} p \leftarrow q, \texttt{not } r \\ p \leftarrow \texttt{not } q, s \\ \hline q \end{array}$$

Next is an example of a program with no negative literals. (The definition of splitting set still works.)

$$\begin{array}{l} p \leftarrow q, r \\ q \\ \hline r \leftarrow s \\ s \end{array} \qquad U = \{r, s\}$$

Here is an example with several non-trivial splitting sets:

$$\begin{array}{l} a \leftarrow b, \texttt{not } c \\ b \leftarrow \texttt{not } d \\ \hline c \end{array} \qquad U = \{c\}$$

$$\begin{array}{l} a \leftarrow b, \texttt{not } c \\ b \leftarrow \texttt{not } d \\ \hline c \end{array} \qquad U = \{c, d\}$$

$$\begin{array}{l} a \leftarrow b, \texttt{not } c \\ \hline b \leftarrow \texttt{not } d \\ c \end{array} \qquad U = \{b, c, d\}$$

$$\begin{array}{l} a \leftarrow b, \texttt{not } c \\ c \\ \hline b \leftarrow \texttt{not } d \end{array} \qquad U = \{b, d\}$$

$$\begin{array}{l} b \leftarrow \texttt{not } d \\ a \leftarrow b, \texttt{not } c \\ c \\ \hline \end{array} \qquad U = \{d\}$$

(Thanks to Dorian Gaertner (MEng4, 2003-04) for pointing out some errors in the original notes.)

Splitting sets

Where a program P is split by U, the stable models of P can be characterised in terms of the stable models of the bottom part and top parts separately.

1. Find a stable model X of $bot_U(P)$.
2. Use X to simplify the clauses in the remaining clauses $top_U(P)$ by 'partially evaluating' them against X (details below); call this simplified set of clauses $e_U(top_U(P); X)$.
3. Find a stable model Y of $e_U(top_U(P); X)$.

$X \cup Y$ will be a stable model of the original program P. Moreover, all stable models of P can be obtained in this way.
Obviously, splitting sets can also be used at steps (1) and (3) above. You can split the program as many times as you like.

The 'partial evaluation'

Given a stable model X for $bot_U(P)$, the clauses in $top_U(P)$ can be simplified to eliminate all atoms occurring in U, as follows.
A clause r in $top_U(P)$ can only contribute to a stable model of P if (i) all positive literals in the body of r that are in U are also in X, and (ii) all negative literals in r whose atoms are in U are not in X.

Definition Let P be a program, and let U and X be sets of atoms. $e_U(P; X)$ is the set of clauses obtained from P as follows. r is a clause in $e_U(P; X)$ iff there is a clause r' in P such that $body^+(r') \cap U \subseteq X$ and $body^-(r') \cap U \cap X = \emptyset$, and r is the clause with $head(r) = head(r')$, $body^+(r) = body^+(r') - U$, and $body^-(r) = body^-(r') - U$.

(Thanks to Silvia Dobrota (MEng4, 2013-14) and Jo Schlemper (MEng4, 2014-15) for correcting some typos in the notation.)

Pictorially, every clause in P can be written in the following form (by re-ordering the literals in the body if necessary):

$$A \leftarrow \underbrace{B_1, \ldots, B_j}_{in\ U}, \overbrace{B_{j+1}, \ldots, B_m}^{not\ in\ U}, \underbrace{\texttt{not } C_1, \ldots, \texttt{not } C_k}_{in\ U}, \overbrace{\texttt{not } C_{k+1}, \ldots, \texttt{not } C_n}^{not\ in\ U}$$

To keep this clause after the 'partial evaluation' we need:

$$A \leftarrow \underbrace{B_1, \ldots, B_j}_{every\ B_i\ in\ X}, \overbrace{B_{j+1}, \ldots, B_m}^{not\ in\ U}, \underbrace{\texttt{not } C_1, \ldots, \texttt{not } C_k}_{no\ C_i\ in\ X}, \overbrace{\texttt{not } C_{k+1}, \ldots, \texttt{not } C_n}^{not\ in\ U}$$

And then after the 'partial evaluation' we are left with a clause with no atoms in U:

$$A \leftarrow B_{j+1}, \ldots, B_m, \texttt{not } C_{k+1}, \ldots, \texttt{not } C_n$$

Theorem [Lifschitz & Turner] Let U be a splitting set for program P. A set of atoms S is a stable model of P if and only if $S = X \cup Y$ for some X and Y such that $X \cap Y = \emptyset$, X is a stable model of $bot_U(P)$, and Y is a stable model of $e_U(top_U(P); X)$.

(Lifschitz and Turner's Splitting Set theorem actually applies to the more general case of disjunctive logic programs.)

Examples

Example

$$\begin{array}{l} a \leftarrow b, \texttt{not } c \\ b \leftarrow c, \texttt{not } a \\ \hline c \end{array} \qquad U = \{c\}$$

There is obviously one stable model for the bottom part. It is $\{c\}$. Using this to simplify the top part gives:

$$b \leftarrow \texttt{not } a$$

This has one stable model: $\{b\}$.
So there is exactly one stable model for the original program, viz. $\{c\} \cup \{b\} = \{c, b\}$.
Note: that splitting sets can also be used to find the stable model $\{b\}$ of the simplified top part $\{b \leftarrow \texttt{not } a\}$. Because that program can be split like this:

$$\begin{array}{l} b \leftarrow \texttt{not } a \\ \hline \emptyset \end{array} \qquad U' = \{a\}$$

Clearly the bottom part of this program has a single stable model: $\emptyset$. Using this to simplify $\{b \leftarrow \texttt{not } a\}$ with respect to $U' = \{a\}$ gives $\{b \leftarrow\}$, and hence the stable model $\{b\}$.

Example

$$\begin{array}{l} c \leftarrow a \\ c \leftarrow b \\ \hline a \leftarrow \texttt{not}\ b \\ b \leftarrow \texttt{not}\ a \end{array} \qquad U = \{a, b\}$$

The bottom part has two stable models: $\{a\}$ and $\{b\}$. Consider them in turn.
Simplifying the top part with $\{a\}$ gives $\{c \leftarrow\}$, which obviously has one stable model, $\{c\}$.
So one stable model for the original program is $\{a\} \cup \{c\} = \{a, c\}$.
Now the other one: Simplifying the top part with $\{b\}$ again gives $\{c \leftarrow\}$. So another stable model for the original program is $\{b\} \cup \{c\} = \{b, c\}$.

Example

$$\begin{array}{l} p \leftarrow q, \texttt{not}\ r \\ p \leftarrow \texttt{not}\ q, s \\ \hline q \end{array} \qquad U = \{q, r, s\}$$

The bottom part has one stable model: $\{q\}$.
Simplifying the top part with $\{q\}$ relative to $U = \{q, r, s\}$ gives $\{p \leftarrow\}$, which has one stable model, viz. $\{p\}$.
So there is one stable model for the original program: $\{q\} \cup \{p\}$.

You can see here the advantage of including all undefined atoms in the splitting set. Compare with this calculation using a different splitting set:

$$\begin{array}{l} p \leftarrow q, \texttt{not}\ r \\ p \leftarrow \texttt{not}\ q, s \\ \hline q \end{array} \qquad U' = \{q\}$$

Now simplifying the top part using $\{q\}$, but this time relative to the splitting set $U' = \{q\}$, gives:

$$p \leftarrow \texttt{not}\ r$$

This has one stable model, $\{p\}$. So you get the same answer eventually but with an additional step.

Example Here is an example of a program with no negative literals.

$$\begin{array}{l} p \leftarrow q, r \\ q \\ \hline r \leftarrow s \\ s \end{array} \qquad U = \{r, s\}$$

The bottom part has one stable model: $\{r, s\}$.

Using this to simplify the top part gives:

$$\begin{array}{l} p \leftarrow q \\ q \end{array}$$

This has one one stable model: $\{p, q\}$.
So the original program has a unique stable model: $\{r, s\} \cup \{p, q\}$.
(Of course, we have other ways of calculating the unique stable model of definite clause programs. And stratified programs.)

Example Here is an example with several non-trivial splitting sets:

$$\begin{array}{l} a \leftarrow b, \texttt{not}\ c \\ b \leftarrow \texttt{not}\ d \\ \hline c \end{array} \qquad U = \{c\}$$

$$\begin{array}{l} a \leftarrow b, \texttt{not}\ c \\ b \leftarrow \texttt{not}\ d \\ \hline c \end{array} \qquad U = \{c, d\}$$

$$\begin{array}{l} a \leftarrow b, \texttt{not}\ c \\ \hline b \leftarrow \texttt{not}\ d \\ c \end{array} \qquad U = \{b, c, d\}$$

$$\begin{array}{l} a \leftarrow b, \texttt{not}\ c \\ c \\ \hline b \leftarrow \texttt{not}\ d \end{array} \qquad U = \{b, d\}$$

$$\begin{array}{l} b \leftarrow \texttt{not}\ d \\ a \leftarrow b, \texttt{not}\ c \\ c \\ \hline \end{array} \qquad U = \{d\}$$

I will just show the working for the case $U = \{c, d\}$. As an exercise, check that the other splitting sets give the same answer.
The bottom part $\{c\}$ has one stable model, $\{c\}$. Simplifying the top part with $\{c\}$ relative to $U = \{c, d\}$ gives $\{b \leftarrow\}$. This has one stable model, $\{b\}$. So we have one stable model for the original program: $\{c\} \cup \{b\}$.

Example (Independent components) It's quite common that we can split a program into completely independent components such that $e_U(top_U(P); X) = top_U(P)$. For example:

$$\begin{array}{l} a_1 \leftarrow b_1, \texttt{not}\ c_1 \\ a_1 \leftarrow d_1 \\ b_1 \leftarrow \\ \hline a_2 \leftarrow \texttt{not}\ b_2, \texttt{not}\ c_2 \\ a_2 \leftarrow d_2 \\ c_2 \leftarrow \\ d_2 \leftarrow \end{array} \qquad U = \{a_2, b_2, c_2, d_2\}$$

Clearly the simplification step here is trivial, and we can work out the stable models of the two components independently.

Example (Independent components) From the tutorial sheet:

```
strong(x) ← big(x), not weak(x)
strong(x) ← small(x),muscular(x), not weak(x)
weak(x) ← small(x), not strong(x)
small(Bill)
muscular(Bill)
big(Mary)
```

Writing out the ground instances and splitting gives:

```
strong(Bill) ← big(Bill), not weak(Bill)
strong(Bill) ← small(Bill),muscular(Bill), not weak(Bill)
weak(Bill) ← small(Bill), not strong(Bill)
small(Bill)
muscular(Bill)
-------------------------------------------------------
strong(Mary) ← big(Mary), not weak(Mary)
strong(Mary) ← small(Mary),muscular(Mary), not weak(Mary)
weak(Mary) ← small(Mary), not strong(Mary)
big(Mary)
```

First, look at the component for `Bill`.

```
strong(Bill) ← big(Bill), not weak(Bill)
strong(Bill) ← small(Bill),muscular(Bill), not weak(Bill)
weak(Bill) ← small(Bill), not strong(Bill)
-------------------------------------------------------
small(Bill)          U = { small(Bill), muscular(Bill), big(Bill) }
muscular(Bill)
```

There's only one stable model {`small(Bill)`, `muscular(Bill)`} for the bottom part. Using this to simplify the top part gives:

```
strong(Bill) ← not weak(Bill)
weak(Bill) ← not strong(Bill)
```

This has two stable models, {`strong(Bill)`} and {`weak(Bill)`}.
Now for the other component (`Mary`):

```
strong(Mary) ← big(Mary), not weak(Mary)
strong(Mary) ← small(Mary),muscular(Mary), not weak(Mary)
weak(Mary) ← small(Mary), not strong(Mary)
-------------------------------------------------------
big(Mary)          U = { small(Mary), muscular(Mary), big(Mary) }
```

There's only one stable model {`big(Mary)`} for the bottom part. Using this to simplify the top part with respect to U gives:

```
strong(Mary) ← not weak(Mary)
```

This has just one stable model: {`strong(Mary)`}.
So there are two stable models for the original program:

{`small(Bill)`, `muscular(Bill)`, `big(Mary)`, `strong(Bill)`, `strong(Mary)`}
{`small(Bill)`, `muscular(Bill)`, `big(Mary)`, `weak(Bill)`, `strong(Mary)`}

Example From the first page of notes on 'Stable models':

```
q(X) ← p(X,Y), not  q(Y)
p(1,2)
```

has ground instances:

```
q(1) ← p(1,1), not  q(1)
q(1) ← p(1,2), not  q(2)
q(2) ← p(2,1), not  q(1)
q(2) ← p(2,2), not  q(2)
p(1,2)
```

But by the 'splitting set' theorem, this set of ground clauses has the same answer sets (stable models) as:

```
q(1) ← not  q(2)
p(1,2)
```

The 'grounders' used in answer set solvers (brief notes later) incorporate optimisations such as these (and others).
Similarly:

```
strong(x) ← big(x), not weak(x)
strong(x) ← small(x),muscular(x), not weak(x)
weak(x) ← small(x), not strong(x)
small(Bill)
muscular(Bill)
big(Mary)
```

When 'grounding' we only need the following equivalent set of ground instances:

```
strong(Mary) ← not weak(Mary)
strong(Bill) ← not weak(Bill)
weak(Bill) ← not strong(Bill)
small(Bill)
muscular(Bill)
big(Mary)
```

Example By the same argument, note that

```
q(X) ← p(X,Y), not  q(Y)
p(1,f(1))
```

which has infinitely many ground instances:

```
q(1) ← p(1,1), not  q(1)
q(1) ← p(1,f(1)), not  q(f(1))
q(1) ← p(1,f(f(1))), not  q(f(f(1)))
  ⋮
q(f(1)) ← p(f(1),1), not  q(1)
q(f(1)) ← p(f(1),f(1)), not  q(f(1))
q(f(1)) ← p(f(1),f(f(1))), not  q(f(f(1)))
  ⋮
p(1,f(1))
```

is equivalent to — has the same answer sets (stable models) as — the finite program:

```
q(1) ← not  q(f(1))
p(1,f(1))
```

Similarly:

```
q(X) ← p(X,Y), not  q(Y)
p(X,f(X)) ←g(X)
g(1)
```

has the same answer sets (stable models) as the finite program:

```
q(1) ← not  q(f(1))
p(1,f(1))
g(1)
```

We can see that:

- A logic program containing function symbols has an infinite Herbrand universe (set of ground terms)
- and therefore, in general, a non-finite set of ground instances of its clauses.
- However, such a program could have a finitely many finite answer sets (stable models) and thus be equivalent to a *finite* set of ground clauses.

One of the main functions of 'grounders' used with answer set solvers (brief notes later) is to detect simple syntactic conditions that guarantee this holds, and then to generate a suitable equivalent finite set of ground clauses.