Slide 1

Grounding logic programs

4.11.1999 Tommi Syrjänen

Slide 2

Terminology

A logic program is *ground* if it is variable-free.

If we substitute variables for constants in a rule, we get a *ground instance* of the rule.

`a(X) :- b(X), c(Y).` $\Longrightarrow$ `a(1) :- b(1), c(2)`

The *Herbrand universe* of a function-free logic program is the set of the constants of the program. The *Herbrand instantiation* of a logic program is the set of all possible ground instances of rules that can be constructed using terms in the Herbrand universe.

The stable models of a logic program with variables is defined to be the set of stable models of the Herbrand instantiation of the program.

Slide 3

Theoretical Background

A *grounding* transforms a general logic program into an equivalent *ground* logic program, where the equivalence is defined as having the same set of stable models. A grounding is *local* if its examines the rules one at a time while disregarding the other rules.

A naive method of grounding would use the full Herbrand instantiation of the program. However, the resulting ground program would be almost always exponential in size when compared with the original program.

Slide 4

Example 1 Let P be the set of rules

$$\begin{gathered}\{d_1(1), d_2(2),\\ a(X) \leftarrow d_1(X), \operatorname{not} b(X),\\ b(X) \leftarrow d_1(X), \operatorname{not} a(X)\}\end{gathered}$$

then the Herbrand instantiation

$$\begin{aligned}\mathbf{HI}(P) = P_G = \{&d_1(1), d_2(2),\\ &a(1) \leftarrow d_1(1), \operatorname{not} b(1),\\ &a(2) \leftarrow d_1(2), \operatorname{not} b(2),\\ &b(1) \leftarrow d_1(1), \operatorname{not} a(1),\\ &b(2) \leftarrow d_1(2), \operatorname{not} a(2)\}\end{aligned}$$

$\mathbf{HI}(P)$ has two stable models $\{d_1(1), d_2(2), a(1)\}$ and $\{d_1(1), d_2(2), b(1)\}$.

Slide 5

The atom $d_1(2)$ can't be deduced and the rules depending on it may be discarded without affecting stable models of the program.

$$\begin{aligned} P_{LG} = \{&d_1(1), d_2(2), \\ &b(1) \leftarrow d_1(1), \mathrm{not}\, a(1) \\ &a(1) \leftarrow d_1(1), \mathrm{not}\, b(1)\} \end{aligned}$$

Which has the same stable models as P_G.

Slide 6

Definition 1 *The* dependency graph $G_P = (V_P, E_P)$ *of a logic program P is constructed as follows:*

1. $V_P = \{A \mid A$ *is a predicate symbol in* $P\}$
2. $(a, b) \in E_P$ *iff there exists a rule $R \in P$ where a is the predicate symbol in head and b is a predicate symbol in body.*

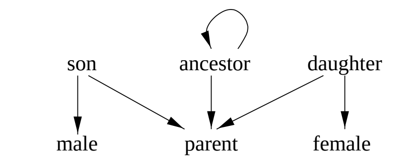
A predicate a depends *on a predicate b iff there exists a path from a to b in the dependency graph.*

Slide 7

Example 2 A small family relations database might be coded in following way:

$$\begin{aligned} ancestor(X, Y) &\leftarrow ancestor(X, Z), parent(Z, Y) \\ ancestor(X, Y) &\leftarrow parent(X, Y) \\ son(X, Y) &\leftarrow parent(Y, X), male(X) \\ daughter(X, Y) &\leftarrow parent(Y, X), female(X) \end{aligned}$$

Its dependency graph is:



Slide 8

Definition 2 *A predicate $p \in P$ is a* domain predicate *iff it holds that every path starting from node $p \in V_P$ is cycle-free.*

Definition 3 *A rule R is* strongly range restricted *if it holds that if a variable appears in the rule it also appears in a positive domain predicate in the body.*

The possible bindings for a variable in a rule are determined by the positive domain predicates in the rule body.

Slide 9

Basic strategy

1. First generate the dependency graph and calculate the extensions of domain predicates.
2. Then ground the rest of the rules one at a time by finding out all possible variable bindings by creating a natural join of all positive domain predicates in the rule body. If a variable binding makes a negative literal false in the rule body it is discarded.

Slide 10

$$P = \{d_1(a,b); d_1(c,b); d_2(a,b); d_2(b,c); d_3(a);$$
$$p(X,Z) \leftarrow d_1(X,Y), d_2(Y,Z), q(X,Y), \mathrm{not}\, d_3(X)\}$$

The join of the domain predicates is calculated by first taking all ground instances of $d_2(Y,Z)$ and then finding all matching values for $d_1(X,Y)$.

1st step			2nd step		
X	Y	Z	X	Y	Z
	a	b	a	b	c
	b	c	c	b	c

The resulting program will be

$$P_{LG} = \{d_1(a,b); d_1(c,b); d_2(a,b); d_3(a)$$
$$p(c,b) \leftarrow d_1(c,b), d_2(b,c), \mathrm{not}\, d_3(c), q(c,b)\}$$

Slide 11

Lparse Syntax

There are four different rule types, and a number of notational shortcuts.

1. Basic rule
2. Constraint rule
3. Choice rule
4. Weight rule

Slide 12

Basic rule

```
h(X) :- a1(X), a2(X), an(X), not b1(X),
        not b2(X), not bm(X).
```

If a1(X), ... , an(X) are in a stable model and b1(X), ..., bn(X) are not, then h(X) is also put in the model.

Slide 13

Constraint rule

```
h(X) :- lower { a1(X), a2(X), an(X),
                not b1(X), not b2(X),
                not bm(X) } upper.
```

If the number of true literals in the body is between lower and upper (inclusive), then the head is put in the model.

Each ground instance of a constraint rule will be transformed into three distinct rules that are outputted, for example:

```
h(a) :- 2 { d1(a), d2(a), d3(a), d4(a) } 3.
```

will be instantiated as:

```
h(a) :- int1(a), not int2(a).
int1(a) :- 2 { d1(a), d2(a), d3(a), d4(a) }.
int2(a) :- 4 { d1(a), d2(a), d3(a), d4(a) }.
```

Slide 14

Choice rule

```
lower { h1(X), h2(X), hk(X) } upper :- body.
```

If the body is true in a stable model, then the number of h1(X), ..., hk(X) which are in the model is between 'lower' and 'upper', inclusive.

If both bounds are missing there may be any number of head literals true in a model.

Internally, a choice rule with non-zero bounds will be translated into three rules:

```
{ h1(X), ..., hk(X) } :- body.
:- upper +1 { h1(X), ..., hk(X) }, body.
:- k - lower +1 { not h1(X), ..., not hk(X) }, body.
```


Slide 15

Weight rule

```
h(X) :- lower [ a1(X)=wa1, a2(X)=wa2, an(X)=wan,
                not b1(X)=wb1, not b2(X)=wb2,
                not bm(X)=wbm ] upper.
```

A weight rule behaves otherwise just like a constraint rule, but each literal may be given a different integral weight, and if the sum of the weights of satisfied literals in the rule body is between 'lower' and 'upper' (inclusive), then h(X) is put to the model. The weight expressions wa1, ..., wbm may be any strongly range restricted expressions.

Slide 16

Conditions

```
a(X) : b(X)
```

This means "all those a(X) for which also b(X) holds".

Thus

```
b(1..3).
h(Y) :- 2 { a(X,Y) : b(X) }.
```

Will be expanded to form

```
b(1..3).
h(Y) :- 2 { a(1,Y), a(2,Y), a(3,Y) }.
```

Slide 17

Functions

There is support for basic arithmetic functions in **lparse**. All functions in **lparse** affect only the grounding and they are removed before **smodels** sees them.

A function may occur in a rule in two roles:

1. As an argument to a predicate or a function ($next(X+1, X) \leftarrow t(X)$).
2. As a test literal in rule body ($p(X,Y) \leftarrow t(X), t(Y), X > Y$).

Slide 18

Declarations

There are three types of declarations:

1. Function declaration (`function foo`).

   This declares the identifier 'foo' to be used as a function throughout the program. The lparse needs to know where the object code is located, see manual for details.
2. Constant declaration (`const bar=value`).

   This declares the identifier 'bar' as being a numeric constant with value 'number'. It is also possible to define constants with command line options
3. Weight definition (`weight baz=expr`).

   This defines the global weight of the literal 'baz' to be 'expr'.

Slide 19

Ranges and multiple argument lists

A range `a(1..3)` is a notational shortcut for

```
a(X) :- int1(X).
int1(1). int1(2). int1(3).
```

A multiple argument list `foo(a;b;c)` defines the set of literals

```
foo(a), foo(b), foo(c)
```

Slide 20

Lparse Invocation

Lparse is used with **smodels**.

Normally the command line is something like:

```
% ./lparse foo | smodels
```

If you only want to see the local grounding that **lparse** produces you can use `-t` switch to do it:

```
% ./lparse -t foo
```

There are also many other command line options. The most important of them is `-c` switch that can be used to define numeric constants:

```
% ./lparse -c n=5 -c m=10 foo | smodels
```

Lparse is available at

Slide 21

http://www.tcs.hut.fi/pub/smodels/lparse.

Slide 22

```
hard_disk( hd_4GB ; hd_6GB ; hd_8GB ; hd_10GB).
monitor( m_15in ; m_17in ; m_21in).
computer(customer).
% every computer has a single monitor
1 { i_mon(X,Y) : monitor(X) } 1 :- computer(Y).
% every computer should have between 1 and 3 hard disks
1 { i_hd(X,Y) : hard_disk(X) } 3 :- computer(Y).
% a computer is a deluxe one if it has two hds and
% a large monitor
package(deluxe,Y) :- 2 { i_hd(X,Y) : hard_disk(X)},
                     1 { i_mon(m_17in, Y),i_mon(m_21in,Y)},
                     computer(Y).
% we want to find possible configurations for customer's
% machine when the monitor has to be at least 17in.
compute 0 { not i_mon(m_15in, customer) }.
```