xpanda

Table of contents

Introduction....1
Usage....2
License....2
Defining variables....3
  Range replacement....3
  Constants in ranges....3
  Multiple variables....3
  Multiple constraints....3
  Predicate constraints....4
  Multiple variables and multiple ranges....4
  Function variables....4
  Rule bodies....4
  Names defined by xpanda....5
Variables and comparision operations....6
  Variables and operators in head....6
  Variables in head....6
  Variables in body....6
  Variables of higher cardinality....6
Extending xpanda....7
  Processors....7
  Adding new processors....7
  Adding new constraint types....7

Introduction

xpanda is a simple preprocessing tool for use with a grounder/asp solver.
Its main advantage is in saving time and tedious copy and paste work. It can expand different types of variable definitions as shown below.

Usage

Using xpanda is very straightforward and simple.

It accepts input from multiple sources like unix-pipes, stdin or an input file. Called without any arguments it will point its output to stdout, infos and warnings will be written to stderr.

The possible commandline options can be aquired by calling xpanda with the argument -h or --help and are listed here:

```
xpanda.py  A simple rule expander for variable grounding

Usage:
    ./xpanda.py [options] [input-filename]
    application | ./xpanda.py [options]
    ./xpanda.py < [options] [input-filename]

Options:
    -h, --help        -- displays this help message
    -v, --verbose     -- outputs info while parsing/replacing
    --version         -- displays version
    -o<file>,         -- saves output to target file
    --outfile<file>
```

A typical call could be:

```
xpanda.py simple.lp | gringo -l | clasp 0
```

License

xpanda is released under the BSD license. Essentially this means the sourcecode is free to be modified, redistributed or changed in any other way.

Defining variables

xpanda will expand the statements.

The replacements use special names to prevent name collisions with your own predicates and facts. The full range of names used by xpanda are listed at the end.

Range replacement

Variables and their constraints can be defined.

```
#variables variable1 = 1..10.
```

```
_x_dom( variable1, 0, 1..10 ).
1{ val( variable1, X_D_Val ) : _x_dom( variable1, 0, X_D_Val ) }1.
```

This will create values from 1 to 10 for variable1.

Constants in ranges

Gringo const values can be used in range statements.

```
#variables variable1 = s..t.
```

```
_x_dom( variable1, 0, s..t ).
1{ val( variable1, X_D_Val ) : _x_dom( variable1, 0, X_D_Val ) }1.
```

This will create values from s to t for variable1.

Multiple variables

A list of variables is possible, each item separated by ','.

```
#variables variable1, variable2 = 1..10.
```

```
_x_dom( variable1, 0, 1..10 ).
1{ val( variable1, X_D_Val ) : _x_dom( variable1, 0, X_D_Val ) }1.

_x_dom( variable2, 0, 1..10 ).
1{ val( variable2, X_D_Val ) : _x_dom( variable2, 0, X_D_Val ) }1.
```

This will create values from 1 to 10 for variable1 and variable2.

Multiple constraints

A list of constraints is possible, each item separated by '|'.

```
#variables variable1 = 1..10 | 21..30.
```

```
_x_dom( variable1, 0, 1..10 ).
_x_dom( variable1, 0, 21..30 ).
1{ val( variable1, X_D_Val ) : _x_dom( variable1, 0, X_D_Val ) }1.
```

This will create values from 1 to 10 and from 21 to 30 for variable1.

Predicate constraints

Also predicate constraints can be added. These have a logic variable assigned, which will be replaced by the variable name. The first part before the colon will be used as the logical variable, the rest will be added to the body of the domain definition.

```
#variables variable1 = X : g(X) : p(X).
```

```
_x_dom( variable1, 0, X ) :- g(X) , p(X).
1{ val( variable1, X_D_Val ) : _x_dom( variable1, 0, X_D_Val ) }1.
```

These constraints can be combined with the range constraints above.

Multiple variables and multiple ranges

Multiple variables and ranges can be combined.

```
#variables variable1, variable2 = 1..10 | 20..25.
```

```
_x_dom( variable1, 0, 1..10 ).
_x_dom( variable1, 0, 20..25 ).
1{ val( variable1, X_D_Val ) : _x_dom( variable1, 0, X_D_Val ) }1.
_x_dom( variable2, 0, 1..10 ).
_x_dom( variable2, 0, 20..25 ).
1{ val( variable2, X_D_Val ) : _x_dom( variable2, 0, X_D_Val ) }1.
```

This will create values from 1 to 10 and 20 to 25 for variable1 and variable2.

Function variables

Variable definitions can be of higher cardinality.

```
#variables variable1(X) = 1..10.
```

```
_x_dom( variable1, 1, 1..10 ).
1{ val( variable1(X), X_D_Val ) : _x_dom( variable1, 1, X_D_Val ) }1.
```

This definition might not make sense without adding a rule body, as defined below.

Rule bodies

Variable definitions can have a body, which will be added to the body of the definition.

```
#variables variable1(X) = 1..10 :- p(X).
```

```
_x_dom( variable1, 1, 1..10 ).
1{ val(variable1(X), X_D_Val) : _x_dom(variable1, 0, X_D_Val) }1 :- p(X).
```

Names defined by xpanda

The following names are defined or used by xpanda, if you want to use those, be aware of possible unwanted side effect.

Predicates

`_x_dom/3`	Created by #variable definition.
`val/2`	Containing all values of variables.

Facts

`[variablename]`	Will be replaced by Val_[variablename].

Variables

`Val_[variablename]`	Replacement for [variablename]
`X_D_Val`	Used in domain definitions

`_x_dom/3` will be defined as `#hidden` for gringo.

Variables and comparision operations

After the variable definitions have been parsed and removed, all variable occurences of the variables specified are searched and then replaced following the rules below. All replacements are made inline to prevent the creation of new rules.

Variables and operators in head

```
variable1 < variable2.
```

```
:- val(variable1, Val_variable1), val(variable2, Val_variable2),
Val_variable1 >= Val_variable2.
```

```
variable2(X , Y) + 4 == 2 :- rule.
```

```
:- val(variable2(X,Y), Val_variable2), Val_variable2 + 4 != 2, rule.
```

The operators used in head are negated and the comparision is moved to the body. If no body existed before, it will be created.

Variables in head

```
f(variable1).
```

```
f(Val_variable1) :- val(variable1, Val_variable1).
```

Variables occuring in the head without a comparision operator will simply be replaced by their logical variable counterpart, which value will be aquired in the rule body (added when neccessary).

Variables in body

```
:- 3 >= variable1( X).
```

```
:- val(variable1(X), Val_variable1), 3 >= Val_variable1.
```

```
head :- variable2 != 10.
```

```
head :- val(variable2, Val_variable2), Val_variable2 != 10.
```

All variables occuring in the body will simply be replaced by their logical variable counterpart, which value will be aquired at the begin of the body.

Variables of higher cardinality

For replacing variables of higher cardinality the following occurences are all parsed the same way to prevent mistyping:

```
variable1(X,Y), variable1( X , Y ), variable1(X, Y), ...
```

Extending xpanda

xpanda was designed to be easily extensible. It is written in python. The complete api documentation can be found in the html folder.

Processors

Several processors are used by xpanda. A processor is a step in the workflow of handling a logical program. All processors extend the class `BaseProcessor`. The normal behavior of xpanda is the following: At the beginning, the logic program is checked for its consistency. Then the `VariablesConstraintExpander` searches for all variable definition rules in the program. For all variables defined there, rules are added to a buffer, the annex, which will be appended to the program at the final output. Now the `VariableOperatorReplacer` walks through all rules of the logical program structure, handling every head and body separate, replacing the variable names and making apropriate changes. At the end the `AnnexMerger` merges the annex buffer to the program.

Adding new processors

The used processors are created in xpanda.py and appended to the main instance. The processors will be called in the order they were added. To create a new processor, one has to create a new class inheriting `BaseProcessor` to xpanda/processors.py which overwrites the `processProgram` method. This method has the current source-code and the program structure as arguments. The source code can be modified in place using predefined convinience methods. Also new program parts can be added to the processors annex, which is a instance variable of `BaseProcessor`.

Adding new constraint types

All kinds of constraints can be parsed, but only those containing either a '..' (`RangeConstraint`) or a ':' (`VariableConstraint`) will be used in xpanda. The kind of constraints are defined in xpanda/parser_helpers.py. To create a new constraint, a class inheriting `Constraint` has to be created there. Then the factory method `_create` of `Constraint` needs to be changed, to detect the new kind of constraint. At last the `VariablesConstraintExpander` must be changed to handle the constraint and add the correct behaviour to the program.