xpanda: A (Simple) Preprocessor for Adding Multi-Valued Propositions to ASP

Martin Gebser, Henrik Hinrichs, Torsten Schaub*, and Sven Thiele

Universität Potsdam, Institut für Informatik, August-Bebel-Str. 89, D-14482 Potsdam, Germany

Abstract. We introduce a simple approach extending the input language of Answer Set Programming (ASP) systems by multi-valued propositions. Our approach is implemented as a (prototypical) preprocessor translating logic programs with multi-valued propositions into logic programs with Boolean propositions only. Our translation is modular and heavily benefits from the expressive input language of ASP. The resulting approach, along with its implementation, allows for solving interesting constraint satisfaction problems in ASP, showing a good performance.

1 Introduction

Boolean constraint solving technologies like Satisfiability Checking (SAT;[1]) and Answer Set Programming (ASP;[2]) have demonstrated their efficiency and robustness in many real-world applications, like planning [3, 4], model checking [5, 6], and bioinformatics [7, 8]. However, many applications are more naturally modeled by additionally using non-Boolean propositions, like resources or functions over finite domains. Unlike in SAT, however, where such language extensions are application-specific, ASP offers a rich application-independent modeling language. The high level of expressiveness allows for an easy integration of new language constructs, as demonstrated in the past by preferences [9] or aggregates [10]. Interesting examples of language extensions illustrating the utility of mixing Boolean and non-Boolean propositions can be found in [11–13], dealing with reasoning about actions.

In fact, a Boolean framework seems to offer such an elevated degree of efficiency that it becomes also increasingly attractive as a target language for non-Boolean constraint languages. This is for instance witnessed by the system Sugar [14], an award-winning SAT-based constraint solver. This motivated us to pursue a translational approach rather than an integrative one, as proposed in [15, 16] or, in more generality, in the field of SAT modulo theories.

In what follows, we expect the reader to be familiar with ASP (cf. [2]) as well as the input language of *lparse* [17, 18] or *gringo* [19, 20].

* Affiliated with the School of Computing Science at Simon Fraser University, Burnaby, Canada, and the Institute for Integrated and Intelligent Systems at Griffith University, Brisbane, Australia.

2 Approach

Our approach takes a logic program with multi-valued propositions and translates it into a standard logic program having Boolean propositions only. A multi-valued proposition is a variable taking exactly one value out of a pre-defined range. Currently, the range of multi-valued propositions is fixed to integer intervals. A multi-valued proposition is subject to two functional conditions, namely, it takes at most and at least one value. This is conveniently expressed by means of cardinality constraints. Let us make this precise by looking at the current input syntax.

For instance, the multi-valued proposition v taking values between 1 and 3 is declared as follows:

$$\#\texttt{variables}\ v = 1..3.$$

Such a declaration is translated into the following expressions:

$$\begin{array}{l} \_x\_dom(v, 0, 1..3). \\ 1\ \{val(v,\ Val) : \_x\_dom(v, 0,\ Val)\}\ 1. \end{array} \tag{1}$$

The new ternary predicate $\_x\_dom$ captures the fact that v has arity 0 and ranges over 1 to 3. This predicate is hidden in the output via $\#\texttt{hide}\ \_x\_dom(X,\ Y,\ Z)$. The value assignment to v is captured by the binary predicate val. Thus passing the logic program in (1) to an ASP system yields three answer sets, given by $\{val(v, 1)\}$, $\{val(v, 2)\}$, and $\{val(v, 3)\}$, each representing a valid assignment to variable v.

The declaration $\#\texttt{variables}$ also allows for more fine-grained specifications, like:

$$\begin{array}{l} \#\texttt{variables}\ u, v = 1..3 \mid 10..20. \\ \#\texttt{variables} f(X) = 1..10\ \texttt{:-}\ p(X). \end{array}$$

The first declaration shows how multiple variables can be specified, sharing a non-consecutive range of values. The second one shows how terms can be incorporated. For this, a term's domain must be guarded by domain predicates, like $p(X)$. These domain predicates are then added as body literals to the resulting cardinality constraint, viz.:

$$1\ \{val(f(X),\ Val) : \_x\_dom(f, 1,\ Val)\}\ 1\ \texttt{:-}\ p(X).$$

Interestingly, the mere possibility of defining multi-valued propositions opens up the possibility of specifying and solving simple constraint satisfaction problems. As an example, take two variables u, v ranging over $\{1, 2, 3\}$ and being subject to the constraint $u + v \leq 3$. This can be expressed by means of the following program:

$$\begin{array}{l} \#\texttt{variables}\ u, v = 1..3. \\ u + v \leq 3. \end{array}$$

This program is then translated as follows. While the declaration of u and v is given as in (1), the constraint $u + v \leq 3$ is expressed via an integrity constraint:

$$\texttt{:-}\ val(u,\ Val_u), val(v,\ Val_v),\ Val_u + Val_v > 3.$$

Note that the original constraint $u + v \leq 3$ appears negated as $u + v > 3$ within the integrity constraint. The idea is to exclude assignments to u and v such that $u + v > 3$. Finally, we note that passing the result of the compilation to an ASP system yields three answer sets, containing

$$\{val(u,1), val(v,1)\}, \quad \{val(u,1), val(v,2)\}, \quad \text{and} \quad \{val(u,2), val(v,1)\}.$$

The above transformation applies whenever an arithmetic expression involving multi-valued propositions appears in the head of a rule. Appearances of such expressions as body literals can be dealt with in a similar way. Unlike above, however, the original constraint is not negated. For instance, the rule within the program

$$\begin{aligned} &\#\texttt{variables}\ u, v = 1..3. \\ &p(u,v)\ \texttt{:-}\ u + v \leq 3. \end{aligned}$$

is turned into

$$p(\mathit{Val}_u, \mathit{Val}_v)\ \texttt{:-}\ val(u, \mathit{Val}_u), val(v, \mathit{Val}_v), \mathit{Val}_u + \mathit{Val}_v \leq 3.$$

The resulting program has nine answer sets, reflecting all possible value assignments to u and v. However, among them, only three contain a single instance of predicate p, namely, $p(1,1)$, $p(1,2)$, and $p(2,1)$.

Finally, our approach provides a dedicated treatment of the popular *alldistinct* constraint, expressing that all involved variables must take pairwisely different values. As before, this constraint is easily mapped onto cardinality constraints. To illustrate this, consider the program:

$$\begin{aligned} &\#\texttt{variables}\ u, v, w = 1..3. \\ &\#\texttt{alldistinct}\ u, v, w. \end{aligned}$$

The declaration of u, v, and w is dealt with as in (1). The *alldistinct* constraint yields the following program rules:

$$\begin{aligned} &\#\texttt{hide}\ \_x\_distinct\_0\_var(\mathit{Var}). \qquad \#\texttt{hide}\ \_x\_distinct\_0\_val(\mathit{Val}). \\ &\_x\_distinct\_0\_var(u). \qquad \_x\_distinct\_0\_val(\mathit{Val})\ \texttt{:-}\ \_x\_dom(u, 0, \mathit{Val}). \\ &\_x\_distinct\_0\_var(v). \qquad \_x\_distinct\_0\_val(\mathit{Val})\ \texttt{:-}\ \_x\_dom(v, 0, \mathit{Val}). \\ &\_x\_distinct\_0\_var(w). \qquad \_x\_distinct\_0\_val(\mathit{Val})\ \texttt{:-}\ \_x\_dom(w, 0, \mathit{Val}). \\ &\texttt{:-}\ \_x\_distinct\_0\_val(\mathit{Val}), 2\ \{val(\mathit{Var}, \mathit{Val}) : \_x\_distinct\_0\_var(\mathit{Var})\}. \end{aligned}$$

The predicates $\_x\_distinct\_0\_var$ and $\_x\_distinct\_0\_val$ are unique for each *alldistinct* constraint, fixing the sets of involved variables and values, respectively. The integrity constraint is violated whenever there are at least two variables sharing a value.

Our translation tool *xpanda* is written in Python and best used via Unix' pipes, e.g.:

```
cat simple.lp | xpanda.py | gringo | clasp 0.
```

A prototype version implementing a subset of the above transformations is (presently) available at http://files.mutaphysis.de/xpanda.zip. It works with ASP systems supporting the input language of *lparse* [17, 18] or *gringo* [19, 20]. Clearly, the translation of *xpanda* can easily be modified to using disjunction and explicit counting aggregates rather than cardinality constraints, and then be used by ASP systems like *dlv* [21].

3 A (Little) Case Study: *SEND+MORE=MONEY*

Let us conduct a brief case-study reflecting the scalability of our approach. To this end, we consider the *SEND+MORE=MONEY* puzzle. The task is to assign distinct numbers from $\{0,\ldots,9\}$ to the variables S, E, N, D, M, O, R, Y such that the addition of the decimal numbers $SEND$ and $MORE$ results in the decimal number $MONEY$. By convention, leading digits of decimal numbers must not be 0. This eliminates 0 from the domains of S and M. Moreover, we know that M cannot be greater than 1 because it occurs as carry. Hence, the value of M must be 1, effectively reducing the variables to S, E, N, D, O, R, Y. For clarity, however, we below use variable notation for M too.

A first and apparently compact representation of this problem is the following one:

```
#variables               m = 1.
#variables               s = 2..9.
#variables     e,n,d,o,r,y = 2..9 | 0.
#alldistinct s,e,n,d,o,r,y.

           s*1000+e*100+n*10+d
  +        m*1000+o*100+r*10+e
 == m*10000+o*1000+n*100+e*10+y.
```

The result of the compilation is given in Appendix A. Unfortunately, the grounding blows up in space because the (non-ground) integrity constraint resulting from the actual *SEND+MORE=MONEY* constraint leads to $8 * 9^6$ ground integrity constraints.

This extreme blow-up is avoided in the following representation, using column-wise addition and three carry variables to express the *SEND+MORE=MONEY* constraint:

```
#variables               m = 1.
#variables               s = 2..9.
#variables     e,n,d,o,r,y = 2..9 | 0.
#alldistinct s,e,n,d,o,r,y.
#variables        n1,e1,y1 = 0..1.

 d+e    == y+y1*10.
 n+r+y1 == e+e1*10.
 e+o+e1 == n+n1*10.
 s+m+n1 == o+ m*10.
```

The result of the compilation is given in Appendix B. Unlike a single constraint with seven variables, this formalization relies on four constraints with at most five variables. This reduces the resulting ground program to 7172 rules, which the ASP solver *clasp* (1.2.1) solves in milliseconds. The overall runtime, including *xpanda*, *gringo* (2.0.3), and *clasp*, is less than half a second when enumerating all solutions. In fact, this example has a unique solution containing:

```
 val(s,9)  val(e,5)  val(n,6) val(d,7)
 val(m,1)  val(o,0)  val(r,8)
val(n1,0) val(e1,1) val(y1,1) val(y,2).
```

4 Conclusion

We have provided a simple transformation-based approach to incorporating multi-valued propositions into ASP. Our translation is modular and heavily benefits from the expressive input language of ASP, providing variables and aggregate statements such as cardinality constraints. Once multi-valued propositions are available, it is possible to formulate and solve interesting constraint satisfaction problems by appeal to ASP technology. As with many ASP applications, the bottleneck of the approach manifests itself in grounding. We have seen that constraints involving too many variables may result in a space blow-up. This phenomenon can to some extent be controlled by the user since the number of variables remains the same in the initial specification and the resulting compilation. Of course, large domains may still be problematic.

Many open questions remain, concerning encoding optimizations, further language constructs, etc., and are subject to future research.

Acknowledgments. We are grateful to Wolfgang Faber for commenting on this paper. This work was partially funded by DFG under Grant SCHA 550/8-1 and by the GoFORSYS[1] project under Grant 0313924.

A First *SEND+MORE=MONEY* Representation: Compilation

```
:- val(s,Val_s), val(e,Val_e), val(n,Val_n), val(d,Val_d),
   val(m,Val_m), val(o,Val_o), val(r,Val_r), val(y,Val_y),
                  Val_s*1000+Val_e*100+Val_n*10+Val_d
    +             Val_m*1000+Val_o*100+Val_r*10+Val_e
   != Val_m*10000+Val_o*1000+Val_n*100+Val_e*10+Val_y.

#hide _x_distinct_0_var(X).
#hide _x_distinct_0_val(X).

_x_distinct_0_var(s). _x_distinct_0_val(Val) :- _x_dom(s,0,Val).
_x_distinct_0_var(e). _x_distinct_0_val(Val) :- _x_dom(e,0,Val).
_x_distinct_0_var(n). _x_distinct_0_val(Val) :- _x_dom(n,0,Val).
_x_distinct_0_var(d). _x_distinct_0_val(Val) :- _x_dom(d,0,Val).
_x_distinct_0_var(o). _x_distinct_0_val(Val) :- _x_dom(o,0,Val).
_x_distinct_0_var(r). _x_distinct_0_val(Val) :- _x_dom(r,0,Val).
_x_distinct_0_var(y). _x_distinct_0_val(Val) :- _x_dom(y,0,Val).

:- _x_distinct_0_val(Val),
   2{ val(Var,Val) : _x_distinct_0_var(Var) }.

#hide _x_dom(X,Y,Z).

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

[1] http://www.goforsys.org

```
_x_dom(s,0,2..9).
1{ val(s,X_D_Val) : _x_dom(s,0,X_D_Val) }1.

_x_dom(e,0,2..9). _x_dom(e,0,0).
1{ val(e,X_D_Val) : _x_dom(e,0,X_D_Val) }1.

_x_dom(n,0,2..9). _x_dom(n,0,0).
1{ val(n,X_D_Val) : _x_dom(n,0,X_D_Val) }1.

_x_dom(d,0,2..9). _x_dom(d,0,0).
1{ val(d,X_D_Val) : _x_dom(d,0,X_D_Val) }1.

_x_dom(o,0,2..9). _x_dom(o,0,0).
1{ val(o,X_D_Val) : _x_dom(o,0,X_D_Val) }1.

_x_dom(r,0,2..9). _x_dom(r,0,0).
1{ val(r,X_D_Val) : _x_dom(r,0,X_D_Val) }1.

_x_dom(y,0,2..9). _x_dom(y,0,0).
1{ val(y,X_D_Val) : _x_dom(y,0,X_D_Val) }1.
```

B Second *SEND+MORE=MONEY* Representation: Compilation

```
:- val(d,Val_d), val(e,Val_e), val(y,Val_y), val(y1,Val_y1),
                              Val_d+Val_e != Val_y+Val_y1*10.
:- val(n,Val_n), val(r,Val_r), val(y1,Val_y1), val(e,Val_e),
   val(e1,Val_e1),    Val_n+Val_r+Val_y1 != Val_e+Val_e1*10.
:- val(e,Val_e), val(o,Val_o), val(e1,Val_e1), val(n,Val_n),
   val(n1,Val_n1),    Val_e+Val_o+Val_e1 != Val_n+Val_n1*10.
:- val(s,Val_s), val(m,Val_m), val(n1,Val_n1), val(o,Val_o),
                       Val_s+Val_m+Val_n1 != Val_o+Val_m*10.

#hide _x_distinct_0_var(X).
#hide _x_distinct_0_val(X).

_x_distinct_0_var(s). _x_distinct_0_val(Val) :- _x_dom(s,0,Val).
_x_distinct_0_var(e). _x_distinct_0_val(Val) :- _x_dom(e,0,Val).
_x_distinct_0_var(n). _x_distinct_0_val(Val) :- _x_dom(n,0,Val).
_x_distinct_0_var(d). _x_distinct_0_val(Val) :- _x_dom(d,0,Val).
_x_distinct_0_var(o). _x_distinct_0_val(Val) :- _x_dom(o,0,Val).
_x_distinct_0_var(r). _x_distinct_0_val(Val) :- _x_dom(r,0,Val).
_x_distinct_0_var(y). _x_distinct_0_val(Val) :- _x_dom(y,0,Val).

:- _x_distinct_0_val(Val),
   2{ val(Var,Val) : _x_distinct_0_var(Var) }.

#hide _x_dom(X,Y,Z).

_x_dom(m,0,1).
```

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

_x_dom(s,0,2..9).
1{ val(s,X_D_Val) : _x_dom(s,0,X_D_Val) }1.

_x_dom(e,0,2..9). _x_dom(e,0,0).
1{ val(e,X_D_Val) : _x_dom(e,0,X_D_Val) }1.

_x_dom(n,0,2..9). _x_dom(n,0,0).
1{ val(n,X_D_Val) : _x_dom(n,0,X_D_Val) }1.

_x_dom(d,0,2..9). _x_dom(d,0,0).
1{ val(d,X_D_Val) : _x_dom(d,0,X_D_Val) }1.

_x_dom(o,0,2..9). _x_dom(o,0,0).
1{ val(o,X_D_Val) : _x_dom(o,0,X_D_Val) }1.

_x_dom(r,0,2..9). _x_dom(r,0,0).
1{ val(r,X_D_Val) : _x_dom(r,0,X_D_Val) }1.

_x_dom(y,0,2..9). _x_dom(y,0,0).
1{ val(y,X_D_Val) : _x_dom(y,0,X_D_Val) }1.

_x_dom(n1,0,0..1).
1{ val(n1,X_D_Val) : _x_dom(n1,0,X_D_Val) }1.

_x_dom(e1,0,0..1).
1{ val(e1,X_D_Val) : _x_dom(e1,0,X_D_Val) }1.

_x_dom(y1,0,0..1).
1{ val(y1,X_D_Val) : _x_dom(y1,0,X_D_Val) }1.
```

References

1. Biere, A., Heule, M., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. IOS Press (2009)
2. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
3. Kautz, H., Selman, B.: Planning as satisfiability. In Neumann, B., ed.: Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92), John Wiley & Sons (1992) 359–363
4. Lifschitz, V.: Answer set planning. In De Schreye, D., ed.: Proceedings of the Sixteenth International Conference on Logic Programming (ICLP'99), MIT Press (1999) 23–37
5. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design **19**(1) (2001) 7–34
6. Heljanko, K., Niemelä, I.: Bounded LTL model checking with stable models. Theory and Practice of Logic Programming **3**(4-5) (2003) 519–550
7. Lynce, I., Marques-Silva, J.: Efficient haplotype inference with Boolean satisfiability. In Gil, Y., Mooney, R., eds.: Proceedings of the Twenty-first National Conference on Artificial Intelligence (AAAI'06), AAAI Press (2006) 104–109

8. Erdem, E., Türe, F.: Efficient haplotype inference with answer set programming. In Fox, D., Gomes, C., eds.: Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI'08), AAAI Press (2008) 436–441
9. Delgrande, J., Schaub, T., Tompits, H.: Logic programs with compiled preferences. In Baral, C., Truszczyński, M., eds.: Proceedings of the Eighth International Workshop on Non-Monotonic Reasoning (NMR'00), arXiv (2000)
10. Faber, W., Pfeifer, G., Leone, N., Dell'Armi, T., Ielpa, G.: Design and implementation of aggregate functions in the DLV system. Theory and Practice of Logic Programming **8**(5-6) (2008) 545–580
11. Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H.: Nonmonotonic causal theories. Artificial Intelligence **153**(1-2) (2004) 49–104
12. Lee, J., Lifschitz, V.: Describing additive fluents in action language C+. In Gottlob, G., Walsh, T., eds.: Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03), Morgan Kaufmann Publishers (2003) 1079–1084
13. Dovier, A., Formisano, A., Pontelli, E.: Multivalued action languages with constraints in CLP(FD). In Dahl, V., Niemelä, I., eds.: Proceedings of the Twenty-third International Conference on Logic Programming (ICLP'07). Volume 4670 of Lecture Notes in Computer Science., Springer-Verlag (2007) 255–270
14. Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling finite linear CSP into SAT. In Benhamou, F., ed.: Proceedings of the Twelfth International Conference on Principles and Practice of Constraint Programming (CP'06). Volume 4204 of Lecture Notes in Computer Science., Springer-Verlag (2006) 590–603
15. Mellarkod, V., Gelfond, M.: Integrating answer set reasoning with constraint solving techniques. In Garrigue, J., Hermenegildo, M., eds.: Proceedings of the Ninth International Symposium on Functional and Logic Programming (FLOPS'08). Volume 4989 of Lecture Notes in Computer Science., Springer-Verlag (2008) 15–31
16. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In Hill, P., Warren, D., eds.: Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09). Volume 5649 of Lecture Notes in Computer Science., Springer-Verlag (2009) 235–249
17. Syrjänen, T.: Lparse 1.0 user's manual. http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz
18. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
19. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`. http://potassco.sourceforge.net
20. Gebser, M., Kaminski, R., Ostrowski, M., Schaub, T., Thiele, S.: On the input language of ASP grounder *Gringo*. In Erdem, E., Lin, F., Schaub, T., eds.: Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09). Volume 5753 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2009) 502–508
21. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic **7**(3) (2006) 499–562