

Grounding Bound Founded Answer Set Programs

Rehan Abdul Aziz, Geoffrey Chu and Peter J. Stuckey

*National ICT Australia, Victoria Laboratory,[†]
Department of Computing and Information Systems,
University of Melbourne, Australia*

Email: raziz@student.unimelb.edu.au, gchu@csse.unimelb.edu.au, pjs@csse.unimelb.edu.au

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Bound Founded Answer Set Programming (BFASP) is an extension of Answer Set Programming (ASP) that extends stable model semantics to numeric variables. While the theory of BFASP is defined on ground rules, in practice BFASP programs are written as complex non-ground expressions. Flattening of BFASP is a technique used to simplify arbitrary expressions of the language to a small and well defined set of primitive expressions. In this paper, we first show how we can flatten arbitrary BFASP rule expressions, to give equivalent BFASP programs. Next, we extend the bottom-up grounding technique and magic set transformation used by ASP to BFASP programs. Our implementation shows that for BFASP problems, these techniques can significantly reduce the ground program size, and improve subsequent solving.

KEYWORDS: Answer Set Programming, Grounding, Flattening, Constraint ASP, Magic Sets

1 Introduction

Many problems in the areas of planning or reasoning can be efficiently expressed using Answer Set Programming (ASP) (Baral 2003). ASP enforces stable model semantics (Gelfond and Lifschitz 1988) on the program, which disallows solutions representing circular reasoning. For example, given only rules $b \leftarrow a$ and $a \leftarrow b$, the assignment $a = \text{true}, b = \text{true}$ would be a solution under the logical semantics normally used by Boolean Satisfiability (SAT) (Mitchell 2005) solvers or Constraint Programming (CP) (Marriott and Stuckey 1998) solvers, but would not be a solution under the stable model semantics used by ASP solvers.

Bound Founded Answer Set Programming (BFASP) (Aziz et al. 2013) is an extension of ASP to allow *founded* integer and real variables. This makes it possible to concisely express and efficiently solve problems involving inductive definitions of numeric variables where we want to disallow circular reasoning. As an example consider the Road Construction problem (*RoadCon*). We wish to decide which roads to build such that the shortest paths between various cities are acceptable, with the minimal total cost. This can be modeled as:

A complete version of this paper that includes examples can be found at <http://arxiv.org/abs/1405.3362>
[†] NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

$$\begin{aligned}
& \text{minimize } \sum_{e \in \text{Edge}} \text{built}[e] \times \text{cost}[e] \\
& \forall y \in \text{Node} : \text{sp}[y, y] \leq 0 \\
& \forall y \in \text{Node}, e \in \text{Edge} : \text{sp}[\text{from}[e], y] \leq \text{len}[e] + \text{sp}[\text{to}[e], y] \leftarrow \text{built}[e] \\
& \forall y \in \text{Node}, e \in \text{Edge} : \text{sp}[\text{to}[e], y] \leq \text{len}[e] + \text{sp}[\text{from}[e], y] \leftarrow \text{built}[e] \\
& \forall p \in \text{Demand} : \text{sp}[\text{d_from}[p], \text{d_to}[p]] \leq \text{demand}[p]
\end{aligned}$$

The decisions are which edges e are built ($\text{built}[e]$). The aim is to minimize the total cost of the edges $\text{cost}[e]$ built. The first rule is a base case that says that shortest path from a node to itself is 0. The second constraint defines the shortest path $\text{sp}[x, y]$ from x to y : the path from x to y is no longer than from x to z along edge e if it is built plus the shortest path from z to y ; and the third constraint is similar for the other direction of the edge. The last constraint ensures that the shortest path for each of a given set of paths $p \in \text{Demand}$ is no longer than its maximal allowed distance $\text{demand}[p]$. The above model has a trivial solution with cost 0 by setting $\text{sp}[x, y] = 0$ for all x, y . In order to avoid this, we require that the sp variables are (upper-bound) *founded* variables, that is they take the largest possible justified value. The first three constraints are actually *rules* which justify upper bounds on sp , the last constraint is a restriction that needs to be met and cannot be used to justify upper bounds. Solving such a BFASP is challenging, mapping to CP models leads to inefficient solving, and hence we need a BFASP solver which can reason directly about *unfounded sets* (Van Gelder et al. 1988) of numeric assumptions. Note that *Constraint ASP* (CASP) and hybrid systems such as those given by (Mellarkod et al. 2008; Gebser et al. 2009; Drescher and Walsh 2012; Liu et al. 2012; Balduccini 2009; Aziz et al. 2013a) cannot solve the above problem without grounding the numeric domain to propositional variables and running into the grounding bottleneck. BFASP has been shown to subsume CP, ASP, CASP and Fuzzy ASP (Nieuwenborgh et al. 2006; Blondeel et al. 2013), see (Aziz et al. 2013) for details.

The above encoding for Road Construction problem is a *non-ground* BFASP since it is parametric in the data: *Node*, *Edge*, *Demand*, *cost*, *from*, *to*, *len*, *d_from*, *d_to* and *demand*. In this paper we consider how to efficiently create a ground BFASP from a non-ground BFASP given the data. This is analogous to *flattening* (Stuckey and Tack 2013) of constraint models and *grounding* (Syrjanen 2009; Gebser et al. 2007; Perri et al. 2007) of ASP programs. The contributions of this paper are: a flattening algorithm that transforms complex expressions to primitive forms while preserving the stable model semantics, a generalization of bottom-up grounding for normal logic programs to BFASPs and a generalization of the magic set transformation (Bancilhon et al. 1985; Beeri and Ramakrishnan 1991) for normal logic programs to BFASPs.

2 Preliminaries

2.1 Constraints and Answer Set Programming

We consider three types of variables: integer, real, and Boolean. Let $\mathcal{V}$ be a set of variables. A *domain* D maps each variable $x \in \mathcal{V}$ to a set of constant values $D(x)$. A *valuation* (or assignment) θ over variables $\text{vars}(\theta) \subseteq \mathcal{V}$ maps each variable $x \in \text{vars}(\theta)$ to a value $\theta(x)$. A restriction of assignment θ to variables V , $\theta|_V$, is the assignment θ' over $V \cap \text{vars}(\theta)$ where $\theta'(v) = \theta(v)$. A *constraint* c is a set of assignments over the variables $\text{vars}(c)$, representing the solutions of the constraint. A constraint c is *monotonically increasing* (resp. *decreasing*) w.r.t. a variable $y \in \text{vars}(c)$ if for all solutions θ that satisfy c , increasing (resp. decreasing) the value of y also creates a solution, that is θ' where $\theta'(y) > \theta(y)$ (resp. $\theta'(y) < \theta(y)$), and

$\theta'(x) = \theta(x), x \in \text{vars}(c) - \{y\}$, is also a solution of c . A *constraint program (CP)* is a collection of variables $\mathcal{V}$ and constraints C on those variables ($\text{vars}(c) \subseteq \mathcal{V}, c \in C$). A *positive-CP* P is a CP where each constraint is increasing in exactly one variable and decreasing in the rest. The *minimal* solution of a positive-CP is an assignment θ that satisfies P s.t. there is no other assignment θ' that also satisfies P and there exists a variable v for which $\theta'(v) < \theta(v)$. Note that for Booleans, $\text{true} > \text{false}$. A positive-CP P always has a unique minimal solution. If we have bounds consistent propagators for all the constraints in the program, then this unique minimal solution can be found simply by performing bounds propagation on all constraints until a fixed point is reached, and then setting all variables to their lowest values.

A *normal logic program* P is a collection of *rules* of the form: $b_0 \leftarrow b_1 \wedge \dots \wedge b_n \wedge \neg b'_1 \wedge \dots \wedge \neg b'_m$ where $\{b_0, b_1, \dots, b_n, b'_1, \dots, b'_m\}$ are Boolean variables. b_0 is the *head* of the rule while the RHS of the reverse implication is the *body* of the rule. A rule without any negative literals is a *positive rule*. A *positive program* is a collection of positive rules. The *least model* of a positive program is an assignment θ that assigns *true* to the minimum number of variables. The *reduct* of P w.r.t. an assignment θ is written P^θ and is a positive program obtained by transforming each rule r of P as follows: if there exists an i for which $\theta(b'_i) = \text{true}$, discard the rule, otherwise, discard all negative literals $\{b'_1, \dots, b'_m\}$ from the rule. The stable models of P are all assignments θ for which the least model of P^θ is equal to θ . Note that if we consider a logic program as a constraint program, then a positive program is a positive-CP and the least model of that program is equivalent to the minimal solution defined above.

2.2 Bound Founded Answer Set Programs (BFASP)

BFASP is an extension of ASP that extends its semantics over integer and real variables. In BFASP, the set of variables is a union of two disjoint sets: standard $\mathcal{S}$ and *founded* variables $\mathcal{F}$.¹ A rule r is a pair (c, y) where c is a constraint, $y \in \mathcal{F}$ is the head of the rule and it is increasing in c . A bound founded answer set program (BFASP) P is a tuple $(\mathcal{S}, \mathcal{F}, C, R)$ where C and R are sets of constraints and rules respectively (also accessed as $\text{constraints}(P)$ and $\text{rules}(P)$ resp.). Given a variable $y \in \mathcal{F}$, $\text{rules}(y)$ is the set of rules with y as their heads. Each standard variable s is associated with a lower and an upper bound, written $lb(s)$ and $ub(s)$ respectively.

The reduct of a BFASP P w.r.t. an assignment θ is a positive-CP made from each rule $r = (c, y)$ by replacing in c every variable $x \in \text{vars}(c) - \{y\}$ s.t. x is a standard variable or c is not decreasing in x , by its value $\theta(x)$ to create a positive-CP constraint c' . Let r^θ denote this constraint. If r^θ is not a tautology, it is included in the reduct P^θ . An assignment θ is a stable solution of P iff i) it satisfies all the constraints in P and ii) it is the minimal solution that satisfies P^θ . For a variable $y \in \mathcal{F}$, the *unconditionally justified bound* of y , written $ujb(y)$, is a value that is unconditionally justified by the rules of the program regardless of what the standard variables are fixed to. E.g. if we have a rule: $(y \geq 3 + x, y)$ where x is a standard variable with domain $[0, 10]$, then we can set $ujb(y) = 3$. For any Boolean, we assume that ujb is fixed to *false*.

The focus of this paper is BFASPs where every rule is written in the form $(y \geq f(x_1, \dots, x_n), y)$. Recall that we consider the domains of Boolean variables to be ordered such that $\text{true} > \text{false}$. So for example, an ASP rule such as $a \leftarrow b \wedge c$ can equivalently be written as: $a \geq f(b, c)$ where

¹ For the rest of this paper we only consider *lower bound* founded variables, analogous to founded Booleans. Upper bound founded variables can be implemented as negated lower bound founded variables, e.g. replace $sp[x, y]$ in the Road Construction example by $\neg nsp[x, y]$ where $nsp[x, y]$ is lower bound founded.

f is a Boolean that returns the value of $b \wedge c$. $f(x_1, \dots, x_n)$ is essentially an expression tree where the leaf nodes are the variables $x_1, \dots, x_n$.

The *local dependency graph* for a BFASP P is defined over founded variables. For each rule $r = (y \geq f(x_1, \dots, x_n), y)$, there is an edge from y to all founded x_i . Each edge is marked increasing, decreasing, or non-monotonic, depending on whether f is increasing, decreasing, or non-monotonic in x_i . A BFASP is *locally valid* iff no edge within an SCC is marked non-monotonic. A program is *locally stratified* if all the edges between any two nodes in the same component are marked increasing.

2.3 Non-ground BFASPs

A *non-ground BFASP* is a BFASP where sets of variables are grouped together in variable arrays, and sets of ground rules are represented by non-ground rules via universal quantification over index variables. For example, if we have arrays of variables a, b, c , then we can represent the ground rules: $(a[1] \geq b[1] + c[1], a[1]), (a[2] \geq b[2] + c[2], a[2]), (a[3] \geq b[3] + c[3], a[3])$ by $\forall i \in [1, 3] : (a[i] \geq b[i] + c[i], a[i])$. Variables can be grouped together in arrays of any dimension and non-ground BFASP rules have the following form: $\forall \bar{i} \in \bar{D}$ where $con(\bar{i}) : (y[l_0(\bar{i})] \geq f(x_1[l_1(\bar{i})], \dots, x_n[l_n(\bar{i})], y[l_0(\bar{i})])$, where $\bar{i}$ is a set of index variables $i_1, \dots, i_m$, $\bar{D}$ is a set of domains $D_1, \dots, D_m$, con is a *constraint* over the index variables which constrains these variables, $l_0, \dots, l_n$ are functions over the index variables which return a tuple of array indices, $y, x_1, \dots, x_n$ are arrays of variables and f is a function over the x_i variables. Let $gen(r) \equiv \bar{i} \in \bar{D} \wedge con(\bar{i})$ denote the *generator constraint* for a non-ground rule r . Note that we require the generator constraint in each rule to constrain the index variables so that f is always defined.

Variable arrays can contain either founded variables, standard variables, or parameters (which can simply be considered fixed standard variables), although all variables in a variable array must be of the same type. Note that the array names in our notation correspond to predicate names in standard ASP syntax, and our index variables correspond to ASP “local variables.” Given a non-ground rule r , let $grnd(r)$ be the set of ground rules obtained by substituting all possible values of the index variables that satisfy $gen(r)$ into the quantified expression. Similarly given a non-ground BFASP P , let $grnd(P)$ be the grounded BFASP that contains the grounding of all its rules and constraints. The *predicate dependency graph*, validity and stratification are defined similarly for array variables and non-ground rules as the local dependency graph, local validity and local stratification respectively are defined for ground variables and ground rules. All our subsequent discussion is restricted to valid BFASPs.

3 Flattening

A ground BFASP may contain constraints and rules whose expressions are not *flat*, i.e., they are expression trees with height greater than one. Such expressions are not supported by constraint solvers and we need to flatten these expressions to primitive forms. We omit consideration of flattening constraints since this is the same as in standard CP (Stuckey and Tack 2013). It can be shown that the standard CP flattening approach in which a subexpression is replaced with a standard variable and a constraint is added that equates the introduced variable with the subexpression, does *not* preserve stable model semantics. To preserve the stable model semantics, it is necessary to use introduced *founded* variables to represent subexpressions containing founded variables. We now describe the central result used in our flattening algorithm.

```

flat(P)
  Pflat := ∅
  R := rules(P)
  T := constraints(P)
  for(r ∈ R)
    R := R \ {r}
    flatRule(r, R, T)
    r := simplify(r)
    Pflat ∪= {r}
  for(c ∈ T) Pflat ∪= cp.flat(c)
  return Pflat

flatRule(r = (y ≥ f(e1, ..., en), y), R, T)
  for(each non-terminal ei)
    if(ei does not contain founded vars)
      replace ei with standard var y' in r
      T ∪= {y' = ei}
    elif(f is increasing in ei)
      replace ei with founded var y' in r
      R ∪= {(y' ≥ ei, y')}
    elif(f is decreasing in ei)
      replace ei with founded var -y' in r
      R ∪= {(y' ≥ -ei, y')}

```

Theorem 1

Let P be a BFASP containing a rule $r = (y \geq f_1(x_1, \dots, x_k, f_2(x_{k+1}, \dots, x_n)), y)$ where f_1 is increasing in the argument where f_2 appears, and where if a variable occurs among both $x_1, \dots, x_k$ and $x_{k+1}, \dots, x_n$, then f_1 and f_2 have the same monotonicity w.r.t. it. Let P' be P with r replaced by the two rules: $r_1 = (y \geq f_1(x_1, \dots, x_k, y'), y)$ and $r_2 = (y' \geq f_2(x_{k+1}, \dots, x_n), y')$ where y' is an introduced founded variable. Then the stable solutions of P' restricted to the variables of P are equivalent to the stable solutions of P .

As a corollary, if f_1 is decreasing in the argument where f_2 appears, we can replace f_2 by a founded variable $-y'$ and add the rule $(y' \geq -f_2(x_k, \dots, x_n), y')$ instead. Not all valid rule forms are supported by Theorem 1, because we require that multiple occurrences of the same variable in the expression must have the same monotonicity w.r.t. the root expression. Note that if a subexpression does not contain any founded variables at all, i.e., only contains standard variables, parameters or constants, then a standard CP flattening step is sufficient. Let us now describe our flattening algorithm `flat` for ground BFASPs and later extend it to non-ground BFASPs. We put all the rules and constraints of the program in sets R and T respectively. For every rule $r = (y \geq f(e_1, \dots, e_n), y) \in R$, where f is the top level function in that rule, and $e_1, \dots, e_n$ are the expressions which form f 's arguments, we call `flatRule` which works as follows. If there is some e_i which is not a terminal, i.e., not a constant, parameter or variable, then we have two cases. If e_i does not contain any founded variables, we simply replace it with standard variable y' and add the constraint $y' = e_i$ to T . Otherwise, we apply the transformation described in Theorem 1. After `flatRule`, we simplify r as much as possible through the subroutine `simplify`, e.g., by getting rid of double negations, pushing negations inside the expressions as much as possible etc. Finally, we flatten all the constraints in T using the standard CP flattening algorithm `cp_flat` as described in (Stuckey and Tack 2013). Since we replace all decreasing subexpressions by negated introduced variables and simplify expressions by pushing negations towards the variables, we handle negation through simple rule forms like $(y \geq -x, y)$, $(y \geq \neg x, y)$ etc.

The algorithm can be extended to non-ground rules by defining the index set of the introduced variables to be equal to the domain of index variables as given in the generator of the rule in which they replace an expression. Moreover, the generator expression of an intermediate rule stays the same as that of the original rule from which it is derived.

4 Grounding

ASP grounders keep track of variables that have been created and instantiate further rules based on that. For example, if the variables b and c have been created, then the rule $a \leftarrow b \wedge c$ justifies a bound on a and therefore, must be included in the final program. The justification of all positive literals in a rule potentially justify its head. However, for a rule, if any one positive variable in its body does not have any rule supporting it, then that rule can safely be ignored until a justification

c	ϕ_r
$y \geq \text{sum}(x_1, \dots, x_n)$	$(\sum_i \text{ujb}(x_i) > \text{ujb}(y)) \vee$ $((\wedge_i \text{ujb}(x_i) > -\infty \vee \text{cr}(x_i)) \wedge (\vee_i \text{cr}(x_i)))$
$y \geq \text{max}(x_1, \dots, x_n)$	$\vee_i (\text{ujb}(x_i) > \text{ujb}(y) \vee \text{cr}(x_i))$
$y \geq \text{min}(x_1, \dots, x_n)$	$\wedge_i (\text{ujb}(x_i) > \text{ujb}(y) \vee \text{cr}(x_i))$
$y \geq \text{product}(x_1, \dots, x_n)$ where $\wedge_i x_i > 0$	$\prod_i \text{ujb}(x_i) > \text{ujb}(y) \vee (\vee_i \text{cr}(x_i))$
$y \geq x \leftarrow r$	$\text{cr}(r) \wedge (\text{ujb}(x) > \text{ujb}(y) \vee \text{cr}(x))$
$y \leftarrow x \geq 0$	$\text{ujb}(x) \geq 0 \vee \text{cr}(x)$
$y \leftarrow \wedge_i x_i$	$\wedge_i \text{cr}(x_i)$
$y \leftarrow \vee_i x_i$	$\vee_i \text{cr}(x_i)$
$y \geq -x$	$-\text{ub}(x) > \text{ujb}(y)$
$y \leftarrow \neg x$	true
$y \geq 1/x$ where $x > 0$	$1 / -\text{ub}(x) > \text{ujb}(y)$

Table 1. Grounding conditions for rule $r = (c, y)$

for that variable has been found. In case a justification is never found for that variable, then the rule is *useless*, i.e., excluding the rule from the program does not change its stable solutions.

We propose a simple grounding algorithm for non-ground BFASPs which can be implemented by simply maintaining a set of ground rules and variables as done in ASP grounders, but which may generate useless rules in addition to all the useful ones. The idea is that for each variable v , we only keep track of whether v can potentially be justified above its ujb value, rather than keeping track of whether it can be justified above each value in its domain. If it can be justified above its ujb , then when v appears in the body of a rule, we assume that v can be justified to any possible bound for the purpose of calculating what bound can be justified on the head.

We refer to a variable x as being *created*, written $\text{cr}(x)$, if it can go above its ujb value. More formally, $\text{cr}(x)$ is a founded Boolean with a rule: $\text{cr}(x) \leftarrow x > \text{ujb}(x)$. While that is how we define $\text{cr}(x)$, we do not explicitly have a variable $\text{cr}(x)$ or the above rule in our implementation. Instead, we implement it by maintaining a set Q of variables that have been created. Initially, Q is empty. We recursively look at each non-ground rule to see if the newly created variables make it possible for more head variables to be justified above their ujb values. If so, we create those variables and add them to Q . In order to do this, we need to find necessary conditions under which the head variable can be justified above its ujb . In order to simplify the presentation, we are going to define ujb for constants, standard variables and parameters as well. For a constant x , we define $\text{ujb}(x)$ to be the value of x . For parameters and standard variables x , we define $\text{ujb}(x) = \text{ub}(x)$.⁴ Note that for soundness, the ujb values of founded variables only have to be correct (e.g. $-\infty$ for all variables) although tighter ujb values can improve the efficiency of our algorithm. Table 1 gives a non-exhaustive list of necessary conditions for the head variable to be justified above its ujb value for different rule forms.

Let us now make a few observations about the conditions given in Table 1. A key point is that for many rule forms ϕ_r can evaluate to *true*, even without any variable in the body getting created. All such rules that evaluate to true give us a starting point for initializing Q in our implementation. The linear case (*sum*) deserves some explanation. It is made up of two disjuncts, the first of which is an evaluation of the initial condition, i.e., whether the sum of ujb values of all variables is greater than the ujb of the head. If this condition is true, then the rule needs to be grounded unconditionally. If this is false, then the second disjunct becomes important. The second disjunct itself is a conjunction of two more conditions. The first one says that all variables must be greater than $-\infty$ in order for the rule to justify a finite value on the head. In the case

⁴ Upper and lower bounds for a parametric array can be established by simply parsing the array.

```

createCPs(P)
  for( $r \in \text{rules}(P) : \phi_r = \bigwedge_{i=1}^n \text{cr}(x_i[\bar{l}_i])$ )
     $\text{cp}[r] := \text{true}$  % new constraint program
     $\text{cp}[r] := \text{cp}[r] \wedge \text{gen}(r)$ 
    for( $i \in 1 \dots n$ )
       $\text{set}[r, i] := \emptyset$ 
       $\text{cp}[r] := \text{cp}[r] \wedge \bar{l}_i \in \ll \text{set}[r, i] \gg$ 
  for( $r \in \text{rules}(P) : \phi_r = \bigvee_{i=1}^n \text{cr}(x_i[\bar{l}_i])$ )
    for( $i \in 1 \dots n$ )
       $\text{cp}[r, i] := \text{true}$  % new constraint program
       $\text{cp}[r, i] := \text{cp}[r, i] \wedge \text{gen}(r)$ 
       $\text{set}[r, i] := \emptyset$ 
       $\text{cp}[r, i] := \text{cp}[r, i] \wedge \bar{l}_i \in \ll \text{set}[r, i] \gg$ 

ground(P)
   $C := \{\text{groundAll}(c) : c \in \text{constraints}(P)\}$ 
   $R' := \{\text{groundAll}(r) : r \in \text{rules}(P) : \phi_r = \text{true}\}$ 
  while( $R' \neq \emptyset$ )
     $H := \text{heads}(R')$ 
     $Q \cup= H$ 
     $R' := \emptyset$ 
    for( $r \in \text{rules}(P) : H \cap \text{vars}(\phi_r) \neq \emptyset$ )
      if( $\phi_r = \bigwedge_{i=1}^n \text{cr}(x_i[\bar{l}_i]) \vee \phi_r = \bigvee_{i=1}^n \text{cr}(x_i[\bar{l}_i])$ )
        for( $i \in 1 \dots n$ )
           $\text{dom} := \{\bar{m} \mid x[\bar{m}] \in Q\}$ 
           $\text{set}[r, i] := \text{dom} \setminus \text{set}[r, i]$ 
          if( $\phi_r$  is conj)  $R' \cup= \text{search}(\text{cp}[r]) \setminus R$ 
          if( $\phi_r$  is disj)  $R' \cup= \text{search}(\text{cp}[r, i]) \setminus R$ 
           $R \cup= R'$ 
           $\text{set}[r, i] := \text{dom}$ 
  for( $y \in \text{vars}(R) \cap \mathcal{F}$ )  $R \cup= (y \geq \text{ujb}(y), y)$ 

```

where all variables already have a finite ujb , the second conjunct says that at least one of them must be created for the rule to be grounded. Finally, observe that after plugging all values of ujb , all conditions given in the table simplify to one of the following four forms: *true*, *false*, $\bigvee_i \text{cr}(x_i)$ or $\bigwedge_i \text{cr}(x_i)$. Note that the grounding conditions are significantly more sophisticated than the simple conjunctive condition for normal rules. More specifically, after simplification, we can get a disjunctive condition which has no analog in ASP.

We are now ready to present the main bottom-up grounding algorithm. Logically, our grounding algorithm starts with $\text{ujb}(x)$ for all x , adds $(x \geq \text{ujb}(x), x)$ to the program and then finds all the ground rules that are not made redundant by these rules. **createCPs** is a preprocessing step that creates constraint programs for rules in a BFASP P whose conditions are either conjunctions or disjunctions. For a rule with a conjunctive condition, it only creates one program, while for one with a disjunctive condition, it creates one constraint program for each variable in the condition. Each program is initialized with the $\text{gen}(r)$ which defines the variables and some initial constraints given in the where clause in the generator of non-ground rule. Furthermore, for each array literal in ϕ_r , a constraint is posted on its literal (which is a function of index variables in the rule), to be in the domain given by the *current* value of the *set* variable (the reason for the Quine quotes) which is initially set to empty. **ground** is called after preprocessing. Q and R are sets of ground variables and rules respectively. **groundAll** is a function that grounds a non-ground rule or constraint completely, and returns the set of all rules and constraints respectively. Initially, we ground all constraints in P and rules for which ϕ_r evaluates to true. R' is a temporary variable that represents the set of new ground rules from the last iteration. In each iteration, we only look for non-ground rules that have some variable in their conditions that is created in the previous iteration. **heads** takes a set of ground rules as its input and returns their heads. In each iteration, through Q , we manipulate the *set* constraint to get new rule instantiations. For each variable in the clause, we make *set* equal to the new index values created for that variable. For both the conjunctive and the disjunctive case, this optimization only tries out new values of recently created variables to instantiate new rules. **search** takes a constraint program as its input, finds all its solutions, instantiates the non-ground rule for each solution, and returns the set of these ground rules. After creating new rules due to the new values in *set*, we make it equal to all values of the variable in Q . The fixed point calculation stops when no new rules are created. Finally, for every founded variable y , we add $(y \geq \text{ujb}(y), y)$ as a rule so that if the ujb relied on some rules that were ignored during grounding, then this ensures that $\text{ujb}(y)$ is always justified.

5 Magic set transformation

Let us first define the *query* of a BFASP. To build the query Q for a BFASP P , we ground all its constraints and its objective function, and put all the variables that appear in them in Q .⁵ Note that our query does not have any free variables and only contains ground variables. Therefore, we do not need adornment strings to propagate binding information as in the original magic set technique. The original magic set technique has three stages: adorn, generate and modify. For the reason described above, we only describe the latter two.

The purpose of the magic set technique is to simulate a top-down computation through bottom-up grounding. For every variable a in the original program, we create a *magic variable* m_a that represents whether we care about a . Additionally, there are *magic rules* that specify when a magic variable should be created. Consider a simple rule $(a \geq b + c, a)$ where ujb of all variables is equal to $-\infty$. Suppose we are interested in computing a , we model this by setting m_a to true. Since b is required to compute the value of a , we add a magic rule $m_b \leftarrow m_a$. We do not care about c until a finite bound on b is justified (until b is created), so we generate a tighter magic rule for c : $m_c \leftarrow m_a \wedge cr(b)$.

We can utilize the necessary conditions for a useful grounding of a rule r as given by ϕ_r . Recall that after evaluating the initial conditions, ϕ_r reduces to true, false, a conjunction or a disjunction. The above generation of magic rules for the rule $(a \geq b + c, a)$ is an example of the conjunctive case. For a disjunction, the magic rules are even simpler. For every $cr(x)$ in the disjunction, we create the magic rule $m_x \leftarrow m_a$. Note that not all variables in the original rule appear in the condition; some might get removed in the simplification or not be included in the original condition at all. We can ignore them for grounding, but we are interested in their values as soon as we know that the rule can be useful. Therefore, as soon as the magic variable for the head is created, and ϕ_r is satisfied, we are interested in all the variables in the rule that do not appear in ϕ_r . Finally, we define the modification step for a rule $r = (y \geq f(\bar{x}), y)$, written $\text{modify}(r)$, as changing it to $r = (y \geq f(\bar{x}) \leftarrow m_y, y)$. The pseudo-code for generation of magic rules and modification of the original rule is given as the function `magic` that takes a rule as its input. It adds magic rules for a rule to a set P . The first two if conditions handle the disjunctive and conjunctive case respectively. The **for** loop that follows generates magic rules for variables that are not in ϕ_r .

The entire bottom-up calculation with magic sets is as follows. First, create magic variables for all the variables in the program and call `magic` for every rule in the program. If the magic rules generated and/or the original rule after modification are not primitive expressions, flatten them. Then, call `ground` on the resulting program. While grounding the constraints, build the query by including m_v in Q for every ground variable v that is in some ground constraint. After grounding, filter all the magic variables from Q , and magic rules from R .

If a given BFASP program is unstratified, then the algorithm described above is not sound. There might be parts of the program that are unreachable from the founded atoms appearing in the query but are inconsistent. We refer the reader to (Faber et al. 2007) for further details. We overcome this by including in the query all ground magic variables of all array variables that are part of a component in the dependency graph in which there is some decreasing (negative) edge between any two of its nodes. The following result establishes correctness of our approach.

⁵ Technically if the problem has output variables, whose value will be printed, they too need to be added to Q .

```

magic(r)
  a := head(r)
  if( $\phi_r = \bigvee_{i=1}^n cr(x_i)$ )
    for( $i \in 1 \dots n$ )  $P \sqcup= gen(r) : (m.x_i \leftarrow m.a, m.x_i)$ 
  if( $\phi_r = \bigwedge_{i=1}^n cr(x_i)$ )
    for( $i \in 1 \dots n$ )
      b := m.a
       $P \sqcup= gen(r) : (m.x_i \leftarrow b, m.x_i)$ 
      b := b  $\wedge$  cr( $x_i$ )
  for( $v \in vars(r) \setminus (vars(\phi_r) \cup \{a\})$ )
     $P \sqcup= gen(r) : (m.v \leftarrow m.a \wedge \phi_r, m.v)$ 
  modify(r)

```

Theorem 2

Given a BFASP P , let G be equal to $grnd(P)$ and M be a ground BFASP produced by running the magic set transformation after including the unstratified parts of the program in the initial query for a given non-ground BFASP P . The stable solutions of G restricted to the variables $vars(M)$ are equivalent to the stable solutions of M . That is, if θ' is a stable solution of G , then $\theta'|_{vars(M)}$ is a stable solution of M and if θ is a stable solution of M , then there exists θ' s.t. θ' is a stable solution of G and $\theta'|_{vars(M)} = \theta$.

6 Experiments

We show the benefits of bottom-up grounding and magic sets for computing with BFASPs on a number of benchmarks: *RoadCon*, *UtilPol* and *CompanyCon*.⁷ In utilitarian policies (*UtilPol*), a government decides a set of policies to enact while minimizing the cost. Additionally, there are different citizens and each citizen's happiness depends on the enacted policies and happiness of other citizens. There is a citizen t whose happiness should be above a given value. Company controls (*CompanyCon*) is a problem related to stock markets. The parameters of the problem are the number of companies, each company's ownership of stocks in other companies, and a source company that wants to *control* a destination company. The decision variables are the number of stocks that the source company buys in every other company. A company c controls a company d if the number of stocks that c owns in d plus the number of stocks that other companies that c controls own in d is greater than 50 percent of total number of stocks of company d . The objective is to minimize the total cost of stocks bought. All experiments were performed on a machine running Ubuntu 12.04.1 LTS with 8 GB of physical memory and Intel(R) Core(TM) i7-2600 3.4 GHz processor. Our implementation extends MiniZinc 2.0 (LIBMZN) and uses the solver CHUFFED extended with founded variables and rules as described in our previous work (Aziz et al. 2013). Each time in the tables is the median time in seconds of 10 different instances.

Table 2 shows the results for *RoadCon*. N is the number of nodes, and SCCs is the minimum number of strongly connected components in the graph. We compare exhaustive grounding (simply creating $grnd(P)$) against bottom-up grounding, and bottom-up grounding with magic set transformation. A — represents either the flattener/solver did not finish in 10 minutes or that it ran out of memory. Using bottom-up grounding, the founded variables representing shortest paths between two nodes that are not in the same SCC and the corresponding useless rules are not created. Clearly bottom-up grounding is far superior to naively grounding everything, and magic sets substantially improves on this. Tables 3 and 4 show the results for utilitarian policies

⁷ All problem encodings and instances can be found at: www.cs.mu.oz.au/~pjs/bound_founded/

N	SCCs	Exhaustive		Bottom-up		Magic	
		Flat	Solve	Flat	Solve	Flat	Solve
100	5	4.25	3.34	1.37	.64	.27	.04
300	15	39.02	—	4.19	1.25	.41	.07
600	20	237.97	—	19.70	22.56	.83	.96
900	30	—	—	30.44	127.90	1.17	4.74
1400	45	—	—	56.99	398.29	1.79	25.66

Table 2. *Road Construction RoadCon*

Instance				Bottom-up		Magic		Instance		Bottom-up		Magic	
C	P	C_r	P_r	Flat	Solve	Flat	Solve	C	C_r	Flat	Solve	Flat	Solve
50	100	5	5	2.02	1.90	.48	.01	1000	15	24.27	5.20	.79	.70
100	300	10	30	16.62	91.66	2.97	.07	1500	25	53.66	17.52	1.39	2.07
100	500	10	30	24.78	—	4.39	.09	2000	35	94.38	66.81	2.31	8.75
250	350	105	105	83.45	—	35.16	18.40	3000	50	209.70	86.35	1.71	17.87
250	400	110	110	88.61	—	39.32	452.17	3500	60	—	—	5.58	19.18
300	400	125	150	140.36	—	57.09	—	5000	80	—	—	9.63	51.45

Table 3. *UtilPol*Table 4. *CompanyCon*

and company controls respectively. The running time for exhaustive and bottom-up for these benchmark are similar, therefore, the comparison is only given for bottom-up vs. magic sets. For *UtilPol*, C and P represent the number of citizens and policies respectively, C_r represents the maximum number of relevant citizens on which the happiness of t directly or indirectly depends and P_r is the maximum number of policies on which the happiness of t and other citizens in C_r depends. This is the part of the instance that is relevant to the query and the rest is ignored when magic sets are enabled. It can be seen that magic sets outperform regular bottom-up grounding, especially when the relevant part of the instance is small compared to the entire instance. Note that when P_r is small, the flattening time for magic sets is greater than the solving time since the resulting set of rules is actually simple. This changes, however, as P_r is increased. For *CompanyCon*, C is the number of total companies while C_r is the maximum number of companies reachable from the destination in the given ownership graph. The table shows that if C_r is small compared to C , magic sets can give significant advantages.

7 Conclusion

Bound Founded Answer Set Programming extends ASP to disallow circular reasoning over numeric entities. In this paper, we show how we can flatten and ground a non-ground BFASP while preserving its semantics, thus creating an executable specification of the BFASP problem. We show that using bottom-up grounding and magic sets transformation we can significantly improve the efficiency of computing BFASPs. The existing magic set techniques are only defined for the normal rule form, involving only founded Boolean variables. We have extended magic sets to BFASP, a formalism that has significantly more sophisticated rule forms and has both standard and founded variables, that can moreover be Boolean or numeric.

References

- AZIZ, R. A., CHU, G., AND STUCKEY, P. J. 2013. Stable model semantics for founded bounds. *Theory and Practice of Logic Programming* 13, 4–5, 517–532. Proceedings of the 29th International Conference on Logic Programming.
- AZIZ, R. A., STUCKEY, P. J., AND SOMOGYI, Z. 2013a. Inductive definitions in constraint programming. In *Proceedings of the Thirty-Sixth Australasian Computer Science Conference*, B. Thomas, Ed. CRPIT, vol. 135. ACS, 41–50.
- BALDUCCINI, M. 2009. Representing constraint satisfaction problems in answer set programming. In *In ICLP09 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP09)*.
- BANCILHON, F., MAIER, D., SAGIV, Y., AND ULLMAN, J. D. 1985. Magic sets and other strange ways to implement logic programs. In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*. ACM, 1–15.
- BARAL, C. 2003. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press.
- BEERI, C. AND RAMAKRISHNAN, R. 1991. On the power of magic. *The Journal of Logic Programming* 10, 255 – 299.
- BLONDEEL, M., SCHOCKAERT, S., VERMEIR, D., AND DE COCK, M. 2013. Fuzzy answer set programming: An introduction. In *Soft Computing: State of the Art Theory and Novel Applications*. Springer, 209–222.
- DRESCHER, C. AND WALSH, T. 2012. Answer set solving with lazy nogood generation. In *Technical Communications of the 28th International Conference on Logic Programming*. 188–200.
- FABER, W., GRECO, G., AND LEONE, N. 2007. Magic sets and their application to data integration. *Journal of Computer and System Sciences* 73, 4, 584–609.
- GEBSER, M., OSTROWSKI, M., AND SCHAUB, T. 2009. Constraint answer set solving. In *Proceedings of the 25th International Conference on Logic Programming*. Springer, 235–249.
- GEBSER, M., SCHAUB, T., AND THIELE, S. 2007. Gringo : A new grounder for answer set programming. In *LPNMR*. 266–271.
- GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference on Logic Programming*. MIT Press, 1070–1080.
- LIU, G., JANHUNEN, T., AND NIEMELA, I. 2012. Answer set programming via mixed integer programming. In *Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning*. AAAI Press, 32–42.
- MARRIOTT, K. AND STUCKEY, P. 1998. *Programming with Constraints: an Introduction*. MIT Press.
- MELLARKOD, V. S., GELFOND, M., AND ZHANG, Y. 2008. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence* 53, 1-4, 251–287.
- MITCHELL, D. G. 2005. A SAT solver primer. *Bulletin of the EATCS* 85, 112–132.
- NIEUWENBORGH, D. V., COCK, M. D., AND VERMEIR, D. 2006. Fuzzy answer set programming. In *Proceedings of Logics in Artificial Intelligence, 10th European Conference, JELIA 2006*. Springer Berlin Heidelberg, 359–372.
- PERRI, S., SCARCELLO, F., CATALANO, G., AND LEONE, N. 2007. Enhancing DLV instantiator by backjumping techniques. *Annals of Mathematics and Artificial Intelligence* 51, 2-4, 195–228.
- STUCKEY, P. J. AND TACK, G. 2013. Minizinc with functions. In *Proceedings of the 10th International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming*. Number 7874 in LNCS. Springer, 268–283.
- SYRJANEN, T. 2009. Logic programs and cardinality constraints – theory and practice. Ph.D. thesis, Faculty of Information and Natural Sciences, Aalto University.
- VAN GELDER, A., ROSS, K. A., AND SCHLIPF, J. S. 1988. Unfounded sets and well-founded semantics for general logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*. ACM, 221–230.

Appendix A Proofs of theorems

Theorem 1

Let P be a BFASP containing a rule $r = (y \geq f_1(x_1, \dots, x_k, f_2(x_{k+1}, \dots, x_n)), y)$ where f_1 is increasing in the argument where f_2 appears, and where if a variable occurs among both $x_1, \dots, x_k$ and $x_{k+1}, \dots, x_n$, then f_1 and f_2 have the same monotonicity w.r.t. it. Let P' be P with r replaced by the two rules: $r_1 = (y \geq f_1(x_1, \dots, x_k, y'), y)$ and $r_2 = (y' \geq f_2(x_{k+1}, \dots, x_n), y')$ where y' is an introduced founded variable. Then the stable solutions of P' restricted to the variables of P are equivalent to the stable solutions of P .

Proof

For a rule $s = (c, \text{head})$, let $\text{con}(s) = c$. By construction, $\text{con}(r) \Leftrightarrow \exists_{y'}(\text{con}(r_1) \wedge \text{con}(r_2))$ and all the other constraints in P and P' are identical. Also, given any assignment θ' of P' , since f_1 is increasing in the argument where f_2 appears, y' will be left as a variable in $r_1^{\theta'}$. Consider an assignment θ' over $\text{vars}(P')$, and let $\theta = \theta'|_{\text{vars}(P)}$. Recall that the reduct of a program with respect to an assignment replaces all the standard variables and founded variables that are not decreasing in any rule's constraint with its value in that assignment. Since f_1 and f_2 have the same monotonicity w.r.t. any variable common in $\{x_1, \dots, x_k\}$ and $\{x_{k+1}, \dots, x_n\}$, it will either be replaced by its assignment value in both f_1 and f_2 or not be replaced at all. Therefore, the relation $r^\theta \Leftrightarrow \exists_{y'}(r_1^{\theta'} \wedge r_2^{\theta'})$ is also valid. Furthermore, all other constraints in P^θ and $P'^{\theta'}$ are identical.

Suppose θ is a stable solution of P . Let θ' be the extension of θ to variable y' s.t. $\theta'(y') = f_2(\theta(x_k), \dots, \theta(x_n))$. Clearly, this choice of $\theta'(y')$ allows θ' to satisfy all the constraints of P' and allows $\theta'|_{\text{vars}(P'^{\theta'})}$ to satisfy all the constraints of $P'^{\theta'}$. To prove that θ' is a stable solution of P' , we just need to show that there is no smaller solution of $P'^{\theta'}$ than $\theta'|_{\text{vars}(P'^{\theta'})}$. Since $r^\theta \Leftrightarrow \exists_{y'}(r_1^{\theta'} \wedge r_2^{\theta'})$ and all other constraints in P^θ and $P'^{\theta'}$ are identical, $P'^{\theta'}$ must force the same lower bounds on the variables in $\text{vars}(P^\theta)$ as P^θ does. Hence, none of those values can go any lower. Also, $r_2^{\theta'}$ forces $y' \geq f_2(\theta(x_k), \dots, \theta(x_n))$, and so $f_2(\theta(x_k), \dots, \theta(x_n))$ is the lowest possible value for y' . Hence $\theta'|_{\text{vars}(P'^{\theta'})}$ is the minimal solution of $P'^{\theta'}$ and θ' is a stable solution of P' .

Suppose θ' is a stable solution of P' . Let $\theta = \theta'|_{\text{vars}(P)}$. Since $\text{con}(r) \Leftrightarrow \exists_{y'}(\text{con}(r_1) \wedge \text{con}(r_2))$ and all the other constraints in P and P' are identical, θ satisfies all the constraints in P . Since $r^\theta \Leftrightarrow \exists_{y'}(r_1^{\theta'} \wedge r_2^{\theta'})$ and all other constraints in P^θ and $P'^{\theta'}$ are identical, $\theta|_{\text{vars}(P^\theta)}$ satisfies all the constraints in P^θ . To prove that θ is a stable solution of P , we just need to show that there is no smaller solution of P^θ than $\theta|_{\text{vars}(P^\theta)}$. Since $r^\theta \Leftrightarrow \exists_{y'}(r_1^{\theta'} \wedge r_2^{\theta'})$ and all other constraints in P^θ and $P'^{\theta'}$ are identical, P^θ must force the same lower bounds on the variables in $\text{vars}(P^\theta)$ as $P'^{\theta'}$ does. Hence, none of those values can go any lower, $\theta|_{\text{vars}(P^\theta)}$ is the minimal solution of P^θ and θ is a stable solution of P . $\square$

Theorem 2

Given a BFASP P , let G be equal to $\text{grnd}(P)$ and M be a ground BFASP produced by running the magic set transformation after including the unstratified parts of the program in the initial query for a given non-ground BFASP P . The stable solutions of G restricted to the variables $\text{vars}(M)$ are equivalent to the stable solutions of M . That is, if θ' is a stable solution of G , then $\theta'|_{\text{vars}(M)}$ is a stable solution of M and if θ is a stable solution of M , then there exists θ' s.t. θ' is a stable solution of G and $\theta'|_{\text{vars}(M)} = \theta$.

Proof

Let us first argue about the correctness of our grounding approach presented in Section 4. We can analyze each row in Table 1 and reason that until the condition is satisfied, the rule can be ignored without changing the stable solutions of the program. We only provide a brief sketch and do not analyze each case in the table. Say, e.g. for $y \geq \max(x_1, \dots, x_n)$, if the condition is not satisfied, this means that no x_i has a rule in the program that justifies a value higher than its ujb , and no x_i initially justifies a bound on y that is greater than $ujb(y)$. If we include a ground version of this rule in the program, then after taking the reduct w.r.t. any assignment, the rule can never justify any bound on the head, and hence can safely be eliminated.

Let P_i be part of $grnd(P)$ that is not included in M . It can be seen from the description of magic set transformation that any variable in P_i either cannot be reached from any variable in M in the dependency graph of P , or can only be reached through useless rules. Since useless rules can be eliminated as argued above, we conclude that no variable in M can reach any variable in P_i in the dependency graph. This obviously also holds for dependency graph of respective reduced program w.r.t. some assignment. This means that for a given assignment θ' , the minimal order computation can first be performed on $M^{\theta'}$ which fixes all the variables in $vars(M)$, and then on $P_i^{\theta'}$ which fixes all the remaining variables, i.e., variables in $vars(P) - vars(M)$. Combining both the minimal solutions would be the same as computing the minimal solution for $G^{\theta'}$. This proves the first result.

For the second result, since all unstratified parts in P are included in M , all the intra-component edges in the dependency graph of P_i are marked increasing (positive). It can be shown that for such a program, once we fix all the standard variables appearing in any rule in P_i , there is a unique stable solution that can be computed as the *iterated least fixpoint* of P_i . This is similar to the well known result for logic programs that states that for a stratified program, the unique stable solution can be computed as the iterated least fixpoint of the program (Corollary 2 in (Gelfond and Lifschitz 1988)). Therefore, if we are given a stable solution θ for M , we can extend it to θ' by fixing all the unfixed standard variables to any value, and then computing the iterated least fixpoint, which will extend θ' over founded variables of P_i , and will be a unique stable solution given the values of all standard variables. $\square$