

GASP: Answer Set Programming with Lazy Grounding

A. Dal Palù¹, A. Dovier², E. Pontelli³, and G. Rossi¹

¹ Dip. Matematica, Univ. Parma,

{`alessandro.dalpalu|gianfranco.rossi`}@unipr.it

² Dip. Matematica e Informatica, Univ. Udine, `dovier@dimi.uniud.it`

³ Dept. Computer Science, New Mexico State Univ., `epontell@cs.nmsu.edu`

Abstract. In this paper we present a novel methodology to compute stable models in Answer Set Programming. The process is performed with a bottom-up approach that does not require the preprocessing of the typical grounding phase. The implementation is completely in Prolog and Constraint Logic Programming over finite domains. The code is very simple and can be used for didactic purposes.

1 Introduction

In recent years, we have witnessed a significant increase of interest towards the *Answer Set Programming* (briefly, ASP) paradigm [13, 14]. The growth of the field has been sparked by two essential activities:

- The development of effective implementations (e.g., SMOBELS [15], DLV [6], CMOBELS [1], CLASP [7])
- The continuous development of *knowledge building blocks* (e.g., [2]) enabling the application of ASP to various problem domains.

The majority of ASP systems rely on a two-stage computation model. The actual computation of the answer set is performed only on propositional programs—either directly (as in SMOBELS and DLV and CLASP) or appealing to the use of a SAT solver (as in ASSAT and CMOBELS). On the other hand, the convenience of ASP programming vitally builds on the use of first-order constructs. This introduces the need of a *grounding* phase, typically performed by a separate grounding program (e.g., LPARSE or GRINGO, or the grounding module of DLV).

The development of sophisticated applications of ASP in real-world domains (e.g., planning [10], phylogenetic inference [3]) has highlighted the strengths and weaknesses of this paradigm. The high expressive power enables the compact and elegant encoding of complex forms of knowledge (e.g., common-sense knowledge, defaults). At the same time, the technology underlying the execution of ASP is still lagging behind and it is often unable to keep up with the demand of complex applications. This has been, for example, highlighted in a recent study of use of ASP to address complex planning problems (drawn from the recent international planning competitions) [17]. A problem like Pipeline (from International

Planning Competition n. 5, IPC-5), whose first 9 instances can be effectively solved by state-of-the-art planners like FF [9], can be solved only for instance 1 using ASP. Using SMOBELS, instances 2 through 4 do not terminate within several hours of execution, while instance 5 generates a ground image that is beyond the input capabilities of SMOBELS.

In this manuscript, we propose a novel implementation of ASP—hereafter named *GASP* (*Grounding-lazy ASP*). The spirit of our effort can be summarized as follows:

- The execution model relies on a novel bottom-up scheme;
- The bottom-up execution model does not require preliminary grounding of the program;
- The internal representation of the program and the computation make use of constraint logic programming over finite domains [4].

This combination of ideas provides a novel system with significant potentials. In particular:

- It enables the simple integration of new features in the solver, such as constraints and aggregates. If preliminary grounding was required, these features would have to be encoded as ground programs, thus reducing the capability to devise general strategies to optimize the search, and often further growing the size of the ground program.
- The adoption of a non-ground search allows the system to control the search process effectively at a higher level, enabling the adoption of Prolog-level implementations of search strategies and the use of static analysis techniques. While in theorem proving-based ASP solvers the search is driven by literals (i.e., the branching in the search tree is generated by alternatively trying to prove p and **not** p), here the search is “*rule-driven*” in the sense that an applicable rule (possibly not ground) is selected and applied.
- It reduces the impact of grounding the whole program before execution, as observed in some domains (e.g., [17]). Grounding is lazily applied to the rules being considered during construction of an answer set, and the ground rules are not kept beyond their needed use.

Given a ASP program P , the key ingredients of the proposed system are:

1. an efficient implementation of the immediate consequence operator T_P (for definite and normal programs);
2. an implementation of an alternating fixpoint procedure for the computation of well-founded models of a (non-ground) program;
3. a mechanism for nondeterministic selection and application of a ground rule to a partial model.

GASP has been developed into a prototype, completely implemented in Prolog and available from <http://www.dimi.uniud.it/dovier/CLPASP>. For efficiency, the representation of predicates is mapped to *finite domain sets* (*FDSETS*), and

techniques are developed to implement a permutation-free search, which limits the risk of repeatedly reconstructing the same answer sets.

The prototype is aimed at demonstrating the feasibility of the proposed approach; at the current stage the system is slower than state-of-the-art ASP solvers. Performance improvements could be gained by using low level data structures that allow constant time w.r.t. linear time access to rules and models. Moreover, as shown in Section 6, our implementation can benefit from *Finite Domain (FD)* and *Finite Domain Set (FDSET)* constraint primitives to significantly speed-up the answer sets search. However, the code is rather short (less than 700 lines, including comments and auxiliary I/O predicates) and it can be used for pedagogical purposes at three levels: for the T_P computation of definite programs, for the computation of well-founded models, and, finally, for the computation of stable models. The first two levels have already performances comparable to other systems, in spite of the simplicity and compactness of the Prolog code.

2 Preliminaries

Let us consider a logic language composed of a collection of atoms \mathcal{A} . An ASP rule has the form:

$$p \leftarrow p_0, \dots, p_n, \mathbf{not} p_{n+1}, \dots, \mathbf{not} p_m$$

where $\{p, p_0, \dots, p_n, p_{n+1}, \dots, p_m\} \subseteq \mathcal{A}$. A program is a collection of ASP rules. Given a rule $a \leftarrow body$, let us denote with $body^+$ the collection of positive literals in $body$ and with $body^-$ the atoms that appear in negative literals in $body$. We also refer to a as $head(a \leftarrow body)$.

We view an interpretation I as a subset of the set of atoms $I \subseteq \mathcal{A}$. I satisfies an atom p if $p \in I$ (denoted by $I \models p$). The interpretation I satisfies the literal $\mathbf{not} p$ if $p \notin I$ (denoted by $I \models \mathbf{not} p$). The notion of entailment can be generalized to conjunctions of literals in the obvious way. An interpretation I is a *model* of a program P if for every rule $p \leftarrow p_0, \dots, p_n, \mathbf{not} p_{n+1}, \dots, \mathbf{not} p_m$ of P , if $I \models p_0 \wedge \dots \wedge p_n \wedge \mathbf{not} p_{n+1}, \dots, \mathbf{not} p_m$ then $I \models p$. We refer to these classical interpretations as to *2-interpretations*.

The traditional immediate consequence operator T_P is generalized to the case of ASP programs as follows:

$$T_P(I) = \{a \in \mathcal{A} \mid (a \leftarrow body) \in P, I \models body\} \quad (1)$$

Any program admits the trivial model \mathcal{A} (w.l.o.g., let us assume that $\mathcal{A} = B_P$, the Herbrand base of the program P). However, this model does not reflect the meaning of the program. It is widely accepted that the notion of *stable model* [8] is a necessary and sufficient condition for being an intended model of a program. A model I of P is stable if I is the least fixpoint of the operator T_{P^I} of the definite clause program P^I obtained adding to the definite clauses in P the

rules $p \leftarrow p_0, \dots, p_n$ such that $p \leftarrow p_0, \dots, p_n$, **not** p_{n+1}, \dots , **not** p_m is in P and $p_{n+1} \notin I, \dots, p_m \notin I$. Stable models are also known as *answer sets*.

Unfortunately, stating the existence of a stable model is NP-complete. When looking for stable models it appears clear (for some atoms) that they must be present in all stable models and (for some other atoms) that they cannot be present in any stable model. This suggested a 3-valued representation of interpretations.

A *3-interpretation* I is a pair $\langle I^+, I^- \rangle$ such that $I^+ \cup I^- \subseteq \mathcal{A}$ and $I^+ \cap I^- = \emptyset$. Intuitively, I^+ denotes the atoms that are known to be true while I^- denotes those atoms that are known to be false. Let us observe that it is not required that $I^+ \cup I^- = \mathcal{A}$. If $I^+ \cup I^- = \mathcal{A}$, then the interpretation I is said to be *complete*.

Given two 3-interpretations I, J , we introduce the relation $I \subseteq J$ to denote the fact that $I^+ \subseteq J^+$ and $I^- \subseteq J^-$. The notion of entailment for 3-interpretations can be defined as follows. If $p \in \mathcal{A}$, then $I \models p$ iff $p \in I^+$; $I \models \mathbf{not} p$ iff $p \in I^-$.

The *well-founded model* [18] of a general program P is a 3-interpretation denoted as $\mathbf{wf}(P)$. Intuitively, the well-founded model of P contains only (possibly not all) the literals that are necessarily true and the ones that are necessarily false in all stable models of P . The remaining literals are undefined, because they depend on loops of dependent literals in P and thus there is no unique interpretation for them. It is well-known that a general program P has a unique well-founded model $\mathbf{wf}(P)$ [18]. If $\mathbf{wf}(P)$ is complete then it is also a stable model (and it is the unique stable model of P).

A well-founded model can be computed deterministically using the idea of alternating fixpoint [19]. This technique uses pairs of 2-interpretations (denoted by I and J) for building the 3-interpretation $\mathbf{wf}(P)$. The immediate consequence operator is extended, with the introduction of another interpretation J :

$$T_{P,J}(I) = \{a \in \mathcal{A} \mid (a \leftarrow bd^+, \mathbf{not} bd^-) \in P, I \models bd^+, (\forall p \in bd^-)(J \not\models p)\} \quad (2)$$

With this extension, the computation of the well-founded model of P is obtained as follows:

$$\begin{cases} K_0 = \text{lfp}(T_{P^+, \emptyset}) & U_0 = \text{lfp}(T_{P, K_0}) \\ K_i = \text{lfp}(T_{P, U_{i-1}}) & U_i = \text{lfp}(T_{P, K_i}) \quad i > 0 \end{cases}$$

where P^+ , used for computing K_0 , is the subset of P composed by definite clauses (facts and rules without negation in body).

When $(K_i, U_i) = (K_{i+1}, U_{i+1})$, the fixpoint is reached and the well-founded (possibly partial) model is the 3-interpretation:

$$\mathbf{wf}(P) = \langle K_i, B_P \setminus U_i \rangle$$

where B_P is the Herbrand base of the program P .

3 Computation-based characterization of stable models

The computation model adopted in GASP has been derived from recent investigations about alternative models to characterize answer set semantics for various extensions of ASP—e.g., programs with *abstract constraint atoms* [16].

3.1 Computations and Answer Sets

The work described in [11] provides a *computation-based* characterization of answer sets for programs with abstract constraints. One of the side-effects of that research is the development of a computation-based view of answer sets for general logic programs. The original definition of answer sets [8] requires guessing an interpretation and successively validating it—through the notion of reduct (P^I) and the ability to compute minimal models of a definite program (e.g., via repeated iterations of the immediate consequence operator [12]).

The characterization of answer sets derived from [11] does not require the initial guessing of a complete interpretation; instead it combines the guessing process with the construction of the answer set.

Let us present this alternative characterization in the case of propositional programs.

Definition 1 (Computation). *A computation of a program P is a sequence of interpretations I_0, I_1, I_2, \dots that satisfies the following conditions:*

- $I_0 = \emptyset$
- $I_i \subseteq I_{i+1}$ for all $i \geq 0$ (Persistence of Beliefs)
- $\bigcup_{i=0}^{\infty} I_i$ is a model of P (Convergence)
- $I_{i+1} \subseteq T_P(I_i)$ for all $i \geq 0$ (Revision)
- if $a \in I_{i+1} \setminus I_i$ then there is a rule $a \leftarrow \text{body}$ in P such that $I_j \models \text{body}$ for each $j \geq i$ (Persistence of Reason).

The results presented in [11] imply the following theorem.

Theorem 1. *Given a program P and a 2-interpretation I , I is an answer set of P if and only if there exists a computation that converges to I .*

The notion of computation characterizes answer sets through an incremental construction process, where the choices are performed at the level of what rules are actually applied to extend the partial answer set.

3.2 A Refined View of Computation

This original notion of computation can be refined in various ways:

- The Persistence of Beliefs rule, together with the convergence rule, indicate that all elements that have a uniquely determined truth value at any stage of the computation can be safely added.

- The notion of computation can be made more specific by enabling the application of only one rule at each step (instead of an arbitrary subset of the applicable rules).

These two observations allow us to rephrase the notion of computation as follows, in the context of 3-interpretations. Given a rule $a \leftarrow body$ and an interpretation I , we say that the rule is *applicable* w.r.t. I if

$$body^+ \subseteq I^+ \text{ and } body^- \cap I^+ = \emptyset.$$

We extend the definition of applicable to a non ground rule R w.r.t. I iff there exists a grounding r of R that is applicable w.r.t. I .

For the sake of simplicity we also overload the \cup operation between interpretations. Given two interpretations I, J , we denote $I \cup J = \langle I^+ \cup J^+, I^- \cup J^- \rangle$. Additionally, given a program P and an interpretation I , we denote with $P \cup I$ the program

$$P \cup I = (P \setminus \{r \in P \mid head(r) \in I^-\}) \cup I^+.$$

Intuitively, $P \cup I$ is the program P modified in such a way to guarantee that all elements in I^+ are true and all elements of I^- are false.

Definition 2 (GASP Computation). *A GASP computation of a program P is a sequence of 3-interpretations I_0, I_1, I_2, \dots that satisfies the following properties:*

- $I_0 = \text{wf}(P)$
- $I_i \subseteq I_{i+1}$ (Persistence of Beliefs)
- if $I = \bigcup_{i=0}^{\infty} I_i$, then $\langle I^+, \mathcal{A} \setminus I^+ \rangle$ is a model of P (Convergence)
- for each $i \geq 0$ there exists a rule $a \leftarrow body$ in P that is applicable w.r.t. I_i and $I_{i+1} = \text{wf}(P \cup I_i \cup \langle body^+, body^- \rangle)$ (Revision)
- if $a \in I_{i+1} \setminus I_i$ then there is a rule $a \leftarrow body$ in P which is applicable w.r.t. I_j , for each $j \geq i$ (Persistence of Reason).

The following result holds:

Theorem 2. *Given a program P , a 3-interpretation I is an answer set of P if and only if there exists a GASP computation that converges to I .*

Proof Sketch. One direction is quite simple, by observing that the computations described in Def. 2 are special cases of the computations of Def. 1 (thus they produce answer sets). The other direction builds on the observation that each computation as in Def. 1 can be rewritten as a computation as in Def. 2, thanks to the Revision and Persistence of Reason properties. \square

4 Computing models using FDSETs

In this section we show how to encode and handle interpretations and answer sets in Prolog using FDSETs. In particular, we show how to represent an interpretation and how to compute the operator T_P and the well-founded model.

FDSETs are a data structure available in the `clpfd` library of SICStus Prolog that allows to efficiently store and compute on sets of integer numbers. Basically, a set $\{a_1, a_2, \dots, a_n\}$ is recognized as the union of a set of intervals $[a_{b_1}..a_{e_1}], \dots, [a_{b_k}..a_{e_k}]$ and stored consequently as $[[a_{b_1}|a_{e_1}], \dots, [a_{b_k}|a_{e_k}]]$. A library of built-in predicates for dealing with this data structure is made available.

4.1 Representation of Interpretations

Most existing front-ends to ASP systems allow the programmer to express programs using a first-order notation. Program atoms are expressed in the form $p(t_1, \dots, t_n)$, where each t_i is either a constant or a variable. Each rule represents a syntactic sugar for the collection of its ground instances. Languages like those supported by the SMODELS system impose syntactic restrictions to facilitate the grounding process and ensure finiteness of the collection of ground clauses. In particular, SMODELS requires each rule in the program to be *range restricted*, i.e., all variables in the rule should occur in *body*⁺. Furthermore, SMODELS requires all variables to appear in at least one atom built using a *domain predicate*—i.e., a predicate that is not recursively defined. Domain predicates play for variables the same role as types in traditional programming languages.⁴

In the scheme proposed here, the instances of a predicate that are true and false within an interpretation are encoded as sets of tuples, and handled using FD techniques.

We identify with p^n a predicate p with arity n . In the program, a predicate p^n appears as $p(X_1, \dots, X_n)$ where, in place of some variables, a constant can occur (e.g., $p(a, X, Y, d)$). The interpretation of the predicate p^n can be modeled as a set of tuples (a_1, a_2, \dots, a_n) , where $a_i \in \text{Consts}(P)$ —where $\text{Consts}(P)$ denotes the set of constants in the language used by the program P . The explicit representation of the set has the maximal cardinality $|\text{Consts}(P)|^n$. The idea is to use a more compact representation based on FDSETs, after a mapping of tuples to integers. Without loss of generality, we assume that $\text{Consts}(P) \subseteq \mathbb{N}$. Each tuple $\mathbf{a} = (a_0, \dots, a_{n-1})$ is mapped to the *unique* number $\text{map}(\mathbf{a}) = \sum_{i \in [0..n-1]} a_i \mathbb{M}^i$, where \mathbb{M} is a “big number”, $\mathbb{M} \geq |\text{Consts}(P)|$. We also extend the `map` function to the case of FD variables. In this case, if $X_i \in \text{Vars}(P)$ then $\text{map}(\mathbf{X})$ is a new FD variable constrained to be equal to the sum defined above, using FD operators. In case of predicates without arguments (predicates of arity 0), for the empty tuple $()$ we set $\text{map}() = 0$.

A 3-interpretation $\langle I^+, I^- \rangle$ can be represented by a set of 4-tuples

$$(p, n, \text{Pos}_{p,n}, \text{Neg}_{p,n}),$$

one for each predicate symbol, where p is the predicate name, n its arity, and

$$\begin{aligned} \text{Pos}_{p,n} &= \{\text{map}(\mathbf{x}) : I^+ \models p(\mathbf{x})\} \\ \text{Neg}_{p,n} &= \{\text{map}(\mathbf{x}) : I^- \models \text{not } p(\mathbf{x})\} \end{aligned}$$

⁴ Some of these restrictions have been relaxed in other systems, e.g., DLV.

The sets Pos and Neg are represented and handled efficiently, by using FD-SETS (see e.g., SICStus Prolog manual). For instance, if

$$Pos_{p,3} = \{\text{map}(0, 0, 1), \text{map}(0, 0, 2), \text{map}(0, 0, 3), \text{map}(0, 0, 8), \\ \text{map}(0, 0, 9), \text{map}(0, 1, 0), \text{map}(0, 1, 1), \text{map}(0, 1, 2)\}$$

and $M = 10$, then its representation as FDSET is simply: $[[1|3], [8|12]]$, in other words, the disjunction of two intervals.

4.2 Minimum model computation

```
(1) apply_def_rule(rule([H],PBody,NBody),I,[atom(F,AR,NEWPDOM,NEG)|PARTI]):-
(2)   copy_term([H,PBody,NBody],[H1,PBody1,NBody1]),
(3)   term_variables([H1,PBody1],VARS),
(4)   bigM(M),M1 is M-1,domain(VARS,0,M1),
(5)   build_constraint(PBody1,I,C3,pos),
(6)   build_constraint(NBody1,I,C4,negknown),
(7)   H1 =.. [F|ARGS],
(8)   build_arity(ARGS,VAR,AR),
(9)   select(atom(F,AR,OLD,NEG),I,PARTI),
(10)  nin_set(AR,VAR,OLD,C1),
(11)  nin_set(AR,VAR,NEG,C2),
(12)  findall(X,(C1+C2+C3+C4 #>= 4, X #= VAR,labeling([ff],VARS)),LIST),
(13)  list_to_fdset(LIST,SET),
(14)  fdset_union(OLD,SET,NEWPDOM).
(15) build_constraint([R|Rs],I,C,Sign):-
(16)   R =.. [F|ARGS],
(17)   builtin(F),!,
(18)   expression(F,ARGS,C2),
(19)   C #<=> C2 #/\ C1,
(20)   build_constraint(Rs,I,C1,Sign).
(21) build_constraint([R|Rs],I,C,Sign):-
(22)   R =.. [F|ARGS],!,
(23)   build_arity(ARGS,VAR,ARITY),
(24)   member(atom(F,ARITY,DOMF,NDOMF),I),
(25)   (Sign=neg,!, nin_set(ARITY,VAR,DOMF,C2);
(26)   Sign=pos,!, C2 #<=> VAR in_set DOMF;
(27)   Sign=negknown, C2 #<=> VAR in_set NDOMF ),
(28)   build_constraint(Rs,I,C1,Sign),
(29)   C #<=> C1 #/\ C2.
(30) build_constraint([],_,1,_).
(31) build_arity(ARGS,VAL,ARITY):-
(32)   (ARGS = [],!, VAL=0, ARITY=0;
(33)   ARGS = [VAL],!, ARITY=1;
(34)   ARGS = [Arg1,Arg2],!, bigM(M), ARITY=2, VAL #= M*Arg1+Arg2;
(35)   ARGS = [A1,A2,A3], bigM(M), ARITY=3, VAL #= M*M*A1+M*A2+A3).
```

Fig. 1. T_P computation

We start showing how the computation of the T_P (see equation (1)) can be implemented using finite domains and FDSET. Actually, the T_P implemented is slightly more complex than the one needed for definite programs, but it has the advantage to be used also during the answer set computation of general programs, when negative information is also known.

We define a simple fixpoint procedure that calls recursively `apply_def_rule` until the model is no longer modifiable. The definition of this predicate is reported in Figure 1. Observe in line 1 that `rule([H],PBody,NBody)` is the internal representation of the rule $H \leftarrow PBody, \text{not} NBody$, while `atom(F,AR,PDOM,NDOM)` is the internal representation of the tuple for the current interpretation of F as described in Section 4.1. In line 2, the variables of the (possibly non-ground) rule $H \leftarrow PBody, \text{not} NBody$ are renamed with fresh variables. These variables are assigned to the finite set domain $0..M1$ where $M1=M-1$ (and the “big number” M introduced in the previous subsection is defined by a fact `bigM(M)`). Constraints on the atoms of the body are set in lines 5 and 6. The predicate `build_constraint` sets membership (`in_set`) and non membership (`nin_set`) constraints on the variables of the atoms of the positive part (resp. negative part) of the body. The idea is that $C3 = 1$ iff the positive body is satisfied by the model I and $C4 = 1$ iff the negative body is satisfied by I . The predicate `build_arity` converts a tuple into a unique FD value according to the function `map` defined in the previous section. In lines (10) and (11), we add the constraints expressing the fact that the head is not yet in the positive part of the model (useless rule application) and it is not in the negative part of the model (inconsistent rule application). The variables C_1 and C_2 take care of these constraints.

In line (12), we collect all the ground instantiations of the rule that lead to new positive introductions of instances of the head. The list of new values for the atom F (see line (7)) is converted to the positive domain and added to the previous values. Observe that `build_constraint` takes care also of built-in predicates (e.g., equalities) calling the auxiliary predicate `expression` that parses the terms in the built-in atom. The additional parameter will be used later to support the computation of the well-founded model.

Observe that (local) grounding is performed in line (12), making sure that useless or redundant grounding are avoided a priori.

4.3 Well-founded model computation

Computing a well-founded model is a deterministic step during GASP computation. As done for T_P in the previous section, the implementation is based on FD constraint programming and FDSETs representation of interpretations. The idea of alternating fixpoint [19] is realized in Prolog.

The implementation boils down to controlling the alternating fixpoint computation and to realizing the $T_{P,J}$ operator (2). We present here the core procedure that analyzes a clause and computes the head predicates to be included in the resulting interpretation.

Let us consider a clause of P :

$$p^n(\mathbf{Y}_0) \leftarrow p_1^{n_1}(\mathbf{Y}_1), \dots, p_k^{n_k}(\mathbf{Y}_k), \text{not } p_{k+1}^{n_{k+1}}(\mathbf{Y}_{k+1}), \dots, \text{not } p_j^{n_j}(\mathbf{Y}_j)$$

with $|\mathbf{Y}_i| = n_i$ and $\mathbf{Y}_i \in (Vars(P) \cup Consts(P))^{n_i}$. Note that \mathbf{Y}_i may contain repeated variables and it can share variables with other literals in the clause.

Given two interpretations I and J , the application of $T_{P,J}$ to I considers each clause such that $I \models body^+$, $J \not\models body^-$. For these clauses, a set of new head

predicates is produced and added to the resulting interpretation. Instead of a generate and test approach, in which the new tuples in the head are generated using unification, we adopt a constraint-based approach. For each clause and interpretation I , we build a CSP that characterizes the atoms $p^n(\mathbf{X})$ derivable from the body.

For each predicate $p_i^{n_i}$ we introduce a FD variable V_i and we create the constraint $V_i = \mathbf{map}(\mathbf{Y}_i)$. Recall that this constraint allows us to connect the individual variables in \mathbf{Y}_i to the variable V_i representing the possible tuples associated to $p_i^{n_i}$. The domain of V_i is Pos , where $(p_i^{n_i}, n_i, Pos, Neg)$ is part of the current interpretation I . For each literal **not** $p_j^{n_j}$, we use the same scheme, except for the fact that the domain of V_j is disjoint from the set Pos' , with $(p_i^{n_i}, n_i, Pos', Neg') \in J$. For the head of the rule, we define a FD variable V_0 similarly. These FD constraints allow us to link the tuples assignments, according to the variables occurrences.

An additional constraint is posted to ensure that the tuples in V_0 (the ones that are to be added to the interpretation) are not already present in the model and are not causing contradictions with the known negative tuples. According to the semantics of $T_{P,J}$, we use a call to the predicate `build_constraint(NBody, J, C4, neg)` in Figure 1. The difference between this constraint and the corresponding one used in the computation of T_P computation is that, by definition, the negative part of the rule may not appear in J^+ .

Finally, a labeling phase for the variables occurring in \mathbf{X} allows us to produce the set of ground instances of \mathbf{X} that satisfy the CSP. The atoms $p^n(\mathbf{X})$ are added to the interpretation, exploiting FDSETs operations.

Example 1. Let us consider an example of the application of $T_{P,\emptyset}(I)$ using a clause and the FD representation of domains. In particular, let us consider the clause

$$p(X) \leftarrow q(X, Y), \mathbf{not} r(Y)$$

where $I = \langle \{q(1, 1), q(1, 2), q(2, 2), p(1)\}, \emptyset \rangle$. Let $\mathbb{M} = 3$, then I is represented as

$$(p, 1, [[1|1], []]), (q, 2, [[4|4], [7|8]], []), (r, 1, [], []).$$

The CSP induced is: $V_0 = X, V_1 = X + 3Y, V_2 = Y$, where the initial domains for V_0 and V_2 is the set $\{0, \dots, \mathbb{M} - 1\} = \{0, 1, 2\}$, while $V_1 \in \{4, 7, 8\}$. Using constraint propagation, the other domains can be restricted to $V_0 \in \{1, 2\}$ and $V_2 \in \{1, 2\}$. Moreover, the predicate p is constrained to be different from the values already known: $V_0 \notin \{1\}$; the predicate p is also constrained not to be in contradiction to its negative facts: in this case no constraint is added. The solution to this CSP is $X \in \{2\}$ and thus the fact $p(2)$ can be added to the interpretation.

4.4 Computing answer sets

The complete answer set enumeration is based on the well-founded model computation, alternated to a non deterministic choice phase. Basically, we collect

the well-founded model I of P . If the model is complete, we return the result. Otherwise, if there are some unknown literals, we start the stable model computation with I as initial interpretation. The call to $\text{wf}(P)$ save the first can detect inconsistent interpretations (failed I). In Figure 2, we summarize in pseudocode the algorithm.

```

(1)  rec_search(I)
(2)    R = applicable_rules(I)
(3)    if (R = ∅ and I is a model) output: I is a stable model
(4)    else select a ← body+, body- ∈ R
(5)      I = wf(P ∪ {body-} ∪ I)
(6)      if (I not failed) rec_search(I)

```

Fig. 2. The answer set computation

Each applicable rule represents a non deterministic choice in the computation of a stable model. The stable model computation explores each of these choices (line 4), and computes I_{i+1} using the wf operator starting for $P \cup \{\text{body}^-\} \cup I_i$ (line 5), as defined in the GASP computation. Note that a is immediately inserted into the model. This step requires the local grounding of each applicable rule in P , according to the interpretation I_i . The local grounding phase is repeated several times during computation, however it should be noted that each ground rule is produced only once along each branch, due to the constraints introduced. However, every time the local grounding in invoked, a CSP is built. We believe that the enhancement of this step (e.g., building CSPs less often) could reduce the search time significantly.

The process may encounter a contradiction while adding new facts to the interpretation, and consequently the computation may encounter failures. Whenever there are no more applicable rules, a leaf in the search tree is reached (line 3) and the corresponding stable model is obtained (convergence property).

The applicable rules w.r.t. an interpretation I_i are determined (line 2) as defined in GASP computation, i.e., solving the CSP using FD and FDSETs with similar techniques to the ones described in the previous sections. Here, the constraints for the negative part are slightly different from the well-founded rule applicability, since the negative part of a clause may not appear in the current I^+ , while in wf we used the additional interpretation J^+ .

In order to separate failure nodes from leaf nodes (stable models with no applicable rules), we organized the expansion as follows. We first collect every applicable rule at a node of the computation (line 2), then we apply the rule and add the new heads to the next interpretation. If the head produces a literal in contradiction to the interpretation, a failure is raised by a consistency check and the search backtracks (line 6). Note that I can be failed, since the well-founded computation is based on a new program that could introduce some contradictions (line 5).

From the implementation point of view, we verified that computing well-founded models at every non deterministic application of a rule is time consuming. In particular, the extraction of the extension of P with new facts from the positive interpretation is inefficient.

To gain efficiency, we substituted the call to the well-founded computation with a variant of the T_P operator. The extension of T_P to ASP considers rules where $body^+ \in I^+$ and $body^- \in I^-$ (actually, the same implemented in the code in Figure 1). The T_P operator adds new positive atoms as stated by the head of the rule. Other consequences that a call to the well-founded solutions could detect in one call will be detected in the successive part of computation (we are not loosing correctness or completeness).

5 A permutation-free labeling

When enumerating stable models with a bottom up tree-based search, special care is needed in order to avoid producing repeated models. In fact, the concept of applicable rules and their non deterministic applications, allow the exploration of equivalent branches, where the order of rule applications is swapped, while the interpretation converges to the same set.

Let us abstract the process of search over a tree as follows. Every node u in the search tree is related to a set of possible choices $C(u)$ to be tried, where $C(u) \subseteq R$ and R is a set of applicable rules. The applicable rules at u depend on the interpretation $I(u)$ present at that node. Each non deterministic choice $r \in C(u)$ at u opens a branch, labeled by r , and it defines a child node of u in the search tree.

We now show a simple example of the combinatorial explosion of the number of branches due to all permutations. Let us assume that a simple search tree has a root u with $C(u) = \{a, b, c\}$. The expansion of the tree produces 6 different leaves, according to the possible permutations in the application of available rules. The unique interpretation $I = \{a, b, c\}$ is explored under every possible order of application of the facts in the program.

Let us introduce a transitive relation $\prec: R \times R \cup \{\perp\}$. Given $A = a_1 \prec a_2 \prec \dots \prec a_n$ and $B = b_1 \prec b_2 \prec \dots \prec b_m$, the *extension* of A w.r.t. B is the new order $a_1 \prec a_2 \prec \dots \prec a_n \prec c_1 \prec \dots \prec c_k$, where c_1, \dots, c_k are the elements in $B \setminus A$ retrieved in the same order as in B .

For example, let $a, b, c \in R$ and $a \prec b \prec c$. The order can be used to guide the nodes expansion, in particular to force the backtracking whenever the order is not respected along a branch. In the previous example, assuming to expand the rule b at the root, the applicable rules at the next node are $\{a, c\}$. The application of a constructs a branch in which b is applied before a and thus that branch fails. The application of c is allowed, and produces a new node in which only a is applicable. However the application of a generates a branch $b \prec c \not\prec a$ which results in a failure. The complete exploration of the search tree leads to a single success leaf, which corresponds to the application of rules in the order $a \prec b \prec c$.

Given a node u , for each applicable rule $r \in C(u)$ at u , a child of u' is expanded. We denote with $r = rule(u')$ the rule that caused the generation of u' . Note that all applicable rules must be collected at a node. When expanding the corresponding child, a test on the order is performed and a backtrack is invoked in case of a failure. This strategy allows us to distinguish between a success node (no applicable rules) and a failure node (every applicable rule violates the order). In fact, filtering out rules that are applicable but violate the ordering would produce an ambiguity on the type of node at hand.

When computing stable models, there is no a-priori order which is suitable to avoid permutations. In fact, the order is built dynamically, while exploring the nodes. Basically, a partial ordering is defined at each node, starting with the empty order at the root. Every time a node u is considered, the order $Ord(u)$ is extended with the applicable rules $C(u)$ of the node. Any expanded child v inherits the new order. To detect the order violation, it is sufficient to test if $r(u) \prec r(v)$ using the order $Ord(v)$. If the test fails, then backtracking is forced, since there is a left branch that contains the expansion $u \prec v$.

6 Experiments

The prototype implementing the ideas described above, as well as the tests described in this section, is available at www.dimi.uniud.it/dovier/CLPASP. The prototype has been developed using SICStus Prolog 4 (<http://www.sics.se/isl/sicstuswww/site/>), chosen for its rich library of FDSET primitives.

We performed some preliminary experiments, using different classes of ASP programs, and we report execution times in Table 1. All the experiments have been performed on an AMD Opteron 2GHz Windows XP machine.

In the set of experiments (Definite), we define a recursive binary predicate p , with the semantics $p(1, 2), p(2, 3), \dots, p(N - 1, N)$, and a predicate h which represents the transitive closure of p . The growth of the running time is quadratic in N , and the ratio between Smodels execution and GASP execution is a (low) constant. Let us observe that Smodels was unable to execute the instance $N = 256$, due to **Error in input** caused by the large size of the ground program. $N = 228$ is the largest instance handled by lparse+Smodels.⁵ We indicate with ∞ the computations that required more than a day to terminate. We also report the grounding times required by Lparse.

We then add to the definite program the following clause defining the predicate r :

$$r(X, Y) :- h(X, Y), \mathbf{not} p(X, Y).$$

The whole program admits a well-founded and stable model. The same complexity observations made for definite programs continue to hold.

Finally, we run GASP and Smodels on a naive ASP modeling of the Schur number problem (see, e.g., [5] for a description).

`number(1..n).`

⁵ smodels 2.26 under Windows XP/cygwin.

```

part(1..p).
1 { inpart(X,P):part(P) } 1 :- number(X).
:- number(X), number(Y), number(Z), part(P),
   inpart(X,P), inpart(Y,P), inpart(Z,P),
   X < Y+1, Z = X+Y.

```

In this case the behavior of GASP (looking for one solution) is worse than in the above simpler cases. This is probably due to the amount of backtracking to be handled at a meta-interpreter level.

The current GASP implementation admits a natural and effective extension if cardinality constraints are used in the ASP program. We focus here on a particular case of a *function* constraint, namely an ASP rule of the form:

```
1 { function(X,Y): range(Y) } 1 :- domain(X).
```

typically used in encoding constraint satisfaction problems [5]. Let us assume that the predicate `range` is defined by the facts `range(a1)...`, `range(an)`. Then, the above rule can be encoded as:

```

function(X,a1) :- domain(X),
                 not function(X,a2), ..., not function(X,an).
...
function(X,an) :- domain(X),
                 not function(X,a1), ..., not function(X,an-1).

```

and GASP will be able to find the correct stable models.

We can anticipate the calling of the `fixpoint` procedure by a non-deterministic, constraint-based, generation of the values of the functions that satisfy ASP *constraints* in the ASP code we are reasoning on. In the implementation available on internet, we have implemented this idea. Basically, we look for pairs (X, Y) to be assigned to the predicate `inpart` that represents *functions* and fulfilling the following constraints generated by unfolding the ASP constraint modeling Schur:

- (1) forall X: number(X) forall Y:number(Y) forall Z:number(Z)
- (2) forall P1: part(P1) forall P2:part(P2) forall P3:part(P3)
- (3) if inpart(X,P1) \wedge inpart(Y,P2) \wedge inpart(Z,P3) \wedge X+Y = Z
- (4) add the constraint $P1 = P2 \Rightarrow P1 \neq P3$

As reported in Table 1 (column GASP-fun) this leads to a dramatic speed-up of the running time and this is encouraging for extensions of the naive implementation.

7 Conclusions

In this paper, we provided the foundation for a bottom-up construction of stable models of a program P without preliminary program grounding. The notion of

	N (n,p)	Lparse	Smodels	GASP	GASP/Smodels	GASP-fun
Definite	32	0.06	0.03	0.14	4.6	0.14
	64	0.09	0.12	0.45	3.7	0.45
	128	0.26	0.79	1.89	2.4	1.89
	228	0.75	1.95	6.78	3.5	6.78
	256	0.65	Error	8.81	-	8.81
Well Founded	32	0.06	0.08	0.98	12.2	0.98
	64	0.11	0.23	3.58	27.8	3.58
	128	0.32	1.07	15.38	14.4	15.38
	228	0.90	3.39	58.70	17.3	58.70
	256	0.68	Error	78.34	-	78.34
Schur	(6,3)	0.04	0.09	1.64	18.2	0.04
	(7,3)	0.05	0.18	6.59	36.6	0.06
	(8,3)	0.05	0.25	27.43	109.7	0.06
	(9,3)	0.06	0.48	113.59	236.65	0.06
	(10,3)	0.06	0.19	480.85	2530.8	0.09
	(30,4)	0.09	0.03	∞	-	0.36
	(35,4)	0.10	0.09	∞	-	0.44
	(40,4)	0.11	77.31	∞	-	0.50
	(45,4)	0.15	∞	∞	-	8315

Table 1. Timings (expressed in seconds). ∞ means ≥ 1 day.

GASP computation has been introduced; this model does not rely on the explicit grounding of the program. Instead, the grounding is local and performed on-demand during the computation of the answer sets. We believe this approach can provide an effective avenue to achieve greater efficiency in space and time w.r.t. a complete program grounding.

We illustrated a preliminary implementation of GASP using CSP on FD variables and FDSETS. We showed how to design T_P , well-founded and stable models computation based on CSPs. This allowed us to encode the entire process in Prolog. Interestingly, the running times for T_P and the well-founded computation are comparable to Smodels. Some ASP programs run slower, due to Prolog overheads and the limited efficiency of some (naive) data structures used.

We plan to investigate how to extend the model to enable the integration of other language features commonly encountered in ASP languages, and how to effectively use such features as constraints to guide the construction of the FDSET search space. We will also explore how global properties of the program and of the partial model can be used by the GASP implementation to improve efficiency.

Acknowledgments

The work is partially supported by MUR FIRB RBNE03B8KK and PRIN projects, and NSF grants HRD0420407 and CNS0220590. We really thank Andrea Formisano for the several useful discussions.

References

1. Y. Babovich and M. Maratea. Cmodels-2: SAT-based Answer Sets Solver Enhanced to Non-tight Programs. <http://www.cs.utexas.edu/users/yuliya>, 2003.
2. C. Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, 2003.
3. D. Brooks, E. Erdem, S. Erdogan, J. Minett, and D. Ringe. Inferring Phylogenetic Trees Using Answer Set Programming. *Journal of Automated Reasoning*, 39(4):471–511, 2007.
4. P. Codognet and D. Diaz. A Minimal Extension of the WAM for clp(fd). In *International Conference on Logic Programming*. MIT Press, 1993.
5. A. Dovier, A. Formisano, and E. Pontelli. A Comparison of CLP(FD) and ASP Solutions to NP-Complete Problems. In *International Conference on Logic Programming*, pages 67–82. Springer Verlag, 2005.
6. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR System dl_v: Progress Report, Comparisons, and Benchmarks. In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 406–417, 1998.
7. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: a Conflict-driven Answer Set Solver. In *Logic Programming and Non-Monotonic Reasoning*, pages 260–265. Springer Verlag, 2007.
8. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programs. In *International Symposium on Logic Programming*, pages 1070–1080. MIT Press, 1988.
9. J. Hoffmann and B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.
10. V. Lifschitz. Answer Set Planning. In *Logic Programming and Non-monotonic Reasoning*, pages 373–374. Springer Verlag, 1999.
11. L. Liu, E. Pontelli, S. Tran, and M. Truszczynski. Logic Programs with Abstract Constraint Atoms: the Role of Computations. In *International Conference on Logic Programming*, pages 286–301. Springer Verlag, 2007.
12. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Heidelberg, 1987.
13. V.W. Marek and M. Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In K.R. Apt, V.W. Marek, M. Truszczyński, and D. S. Warren, editors, *The Logic Programming Paradigm*. Springer Verlag, 1999.
14. I. Niemela. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *Annals of Mathematics and AI*, (to appear).
15. I. Niemela and P. Simons. Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal LP. In *Logic Programming and Non-monotonic Reasoning*, pages 421–430. Springer Verlag, 1997.
16. T. Son and E. Pontelli. Set Constraints in Logic Programming. In *Logic Programming and Non-Monotonic Reasoning*, pages 167–179. Springer Verlag, 2004.
17. T. Son and E. Pontelli. Planning for Biochemical Pathways: a Case Study of Answer Set Planning in Large Planning Problem Instances. In *First International Workshop on Software Engineering for Answer Set Programming*, pages 116–130, 2007.
18. A. Van Gelder, K.A. Ross, and J.S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.
19. U. Zukowski, B. Freitag, and S. Brass. Improving the Alternating Fixpoint: The Transformation Approach. Proc. of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 4–59, 1997.