GASP: Answer Set Programming with Lazy Grounding

Alessandro Dal Palù
Dipartimento di Matematica, University of Parma
alessandro.dalpalu@unipr.it

Agostino Dovier
Dipartimento di Matematica e Informatica, University of Udine
dovier@dimi.uniud.it

Enrico Pontelli
Dept. Computer Science, New Mexico State University
epontell@cs.nmsu.edu

Gianfranco Rossi
Dipartimento di Matematica, University of Parma
gianfranco.rossi@unipr.it

December 18, 2009

Abstract

In recent years, Answer Set Programming has gained popularity as a viable paradigm for applications in knowledge representation and reasoning. This paper presents a novel methodology to compute answer sets of an answer set program. The proposed methodology maintains a bottom-up approach to the computation of answer sets (as in existing systems), but it makes use of a novel structuring of the computation, that originates from the non-ground version of the program. Grounding is lazily performed during the computation of the answer sets. The implementation has been realized using Constraint Logic Programming over finite domains.

1 Introduction

In recent years, we have witnessed a significant increase of interest towards *Answer Set Programming* (briefly, ASP) [17, 19]. ASP is a logic programming paradigm, whose syntax corresponds to that of *normal logic program*—i.e., logic programs with negation as failure—and whose semantics is given by the *stable model semantics* [10]. The growth of the field has been sparked by two key contributions:

- The development of effective implementations (e.g., SMODELS [20], DLV [13], ASSAT [15], CMODELS [1], CLASP [9])
- The creation 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 SMODELS, DLV, and CLASP) or via translation to a SAT solving problem (as in ASSAT and CMODELS). 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, GRINGO, or the grounding module of DLV).

The development of complex applications of ASP in real-world domains (e.g., planning [14], phylogenetic inference [4]) 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 unable to keep up with the demand of the more complex applications. This has been, for example, highlighted in a recent study concerning the use of ASP to solve complex planning problems (drawn from recent international planning competitions) [23]. A problem like Pipeline (from IPC-5), whose first 9 instances can be effectively solved by state-of-the-art planners like FF [11], can be solved only in its first instance using ASP. Using LPARSE+SMODELS, instances 2 through 4 do not terminate within several hours of execution, while instance 5 leads LPARSE to generate a ground image that is beyond the input capabilities of SMODELS. Moreover, in [8] it is shown that constraint logic programming over finite domains is a technology that can improve running time of ASP solvers over ASP programs encoding planning problems.

In this manuscript, we propose a novel implementation of ASP—hereafter named *Grounding-lazy ASP (GASP)*. The spirit of our proposal 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 [5].

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 leading to exponential growth in the size of the ground program.
- The adoption of a non-ground search allows the system to effectively control the search process 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 is generated by alternatively trying to prove p and **not** p), in GASP the search is "*rule-driven*"—in the sense that an applicable rule (possibly not ground) is selected and applied.
- It reduces the negative impact of grounding the whole program before execution (e.g., [23]). Grounding is lazily applied to the rules being considered during the construction of an answer set, and the ground rules are not kept beyond their needed use.

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

1. A fast 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 program rule to a partial model;
4. An optimized search procedure for a family of ASP programs based on constraint-based mechanisms.

GASP has been implemented in a complete prototype developed in Prolog and available at `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 avoids the repeated reconstruction of the same answer sets. As we will show in Section 5, our implementation benefits from *Finite Domain (FD)* and *Finite Domain Set (FDSET)* constraint primitives to speed-up answer set computation. The prototype demonstrates the feasibility of the proposed approach; at the current stage, the system outperforms state-of-the-art ASP solvers on some problems instances. Moreover, we show that some problems that generate a non-tractable ground instance are effectively solved by GASP.

Related work: This work is the extended version of a previously presented preliminary version [6]. Here we introduce propagation techniques and indexing for a faster computation. New results are computed and compared to the state-of-the-art solvers.

The experimental section provides comparisons with another system with similar characteristics (*ASPeRiX*). ASPeRiX [12] has been concurrently and independently developed. Both GASP and ASPeRiX have their theoretical roots in the same notion of computation-based characterization of answer sets [16]. ASPeRiX is implemented in C++ and develops heuristics aimed at enhancing the choice of the rules when more of them are applicable. The idea is based on the analysis of the dependency graph of the predicates in the program.

Models for non-ground computation based on alternative execution schemes (e.g., top-down computations) have also been recently proposed (e.g., [3]).

2 Preliminaries

Let us consider a logic language composed of a collection of propositional atoms $\mathcal{A}$. An ASP rule has the form: $p \leftarrow p_1, \ldots, p_n, \textbf{not}\, p_{n+1}, \ldots, \textbf{not}\, p_m$ where $\{p, p_1, \ldots, p_n, p_{n+1}, \ldots, p_m\} \subseteq \mathcal{A}$. A program is a collection of ASP rules. We also refer to these programs as *propositional* programs. If $m = n = 0$, then we simply write p. An ASP constraint is an ASP rule without head: $\leftarrow p_1, \ldots, p_n, \textbf{not}\, p_{n+1}, \ldots, \textbf{not}\, p_m$. An ASP constraint is a syntactic sugar for the ASP rule $p \leftarrow \textbf{not}\, p, p_1, \ldots, p_n, \textbf{not}\, p_{n+1}, \ldots, \textbf{not}\, p_m$, where p is a new propositional atom. ASP constraints are used to denote propositional formulae $p_1 \wedge \ldots \wedge p_n \wedge \neg p_{n+1} \wedge \ldots \wedge \neg p_m$ that should not be entailed.

ASP programs can be expressed using first-order atoms, variables, and a finite set of constants. Moreover, a finite set of arithmetic constants and operators are commonly allowed. A non-ground rule is a shorthand for the collection of all its ground instances. In turn, every ground atom, with abuse of notation, can be seen as a propositional atom of the set $\mathcal{A}$. The *grounding* process is the process that replaces each non-ground rule in a program with the corresponding finite set of ground rules. For instance, the non-ground program containing the rules $p(a)$, $p(b)$ and $r(X) \leftarrow q(X, Y)$ correspond to the ground rules

$$p(a) \quad p(b) \quad r(a) \leftarrow q(a,a) \quad r(a) \leftarrow q(a,b) \quad r(b) \leftarrow q(b,a) \quad r(b) \leftarrow q(b,b).$$

Given a rule $a \leftarrow \mathit{body}$, let us denote with body^+ the collection of positive literals in *body* and with body^- the collection of atoms that appear in negative literals in *body*. Hence, the rule $a \leftarrow \mathit{body}$ can be represented as $a \leftarrow \mathit{body}^+, \textbf{not}\, \mathit{body}^-$. Given $a \leftarrow \mathit{body}$, let $\mathit{head}(a \leftarrow \mathit{body})$ denote a. A rule is *definite* if body^- is empty.

We view an interpretation (or *2-interpretation*) I as a subset of the set of atoms $\mathcal{A}$, i.e., $I \subseteq \mathcal{A}$. I satisfies an atom p if $p \in I$ (denoted by $I \models p$). The interpretation I satisfies the literal $\textbf{not}\, p$ if $p \notin I$ (denoted by $I \models \textbf{not}\, p$). 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 $p \leftarrow p_1, \ldots, p_n, \textbf{not}\, p_{n+1}, \ldots, \textbf{not}\, p_m$ of P, if $I \models p_1 \wedge \cdots \wedge p_n \wedge \textbf{not}\, p_{n+1} \wedge \cdots \wedge \textbf{not}\, p_m$ then $I \models p$.

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

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

We will use the standard notation $T_P \uparrow 0(I) = I, T_P \uparrow (n+1)(I) = T_P(T_P \uparrow n(I)), T_P \uparrow \omega(I) = \bigcup_{n \in \mathbb{N}} T_P \uparrow n(I)$. If I is omitted then it is assumed $I = \emptyset$. If T_P is monotone and continuous, then $T_P \uparrow \omega$ is the least fixpoint and it is denoted also as $\mathsf{lfp}(T_P)$.

Given an interpretation I, we say that I is a *supported model* of a program P if I is a model of P and for each $a \in I$ there exists a rule $a \leftarrow \mathit{body}$ in P such that $I \models \mathit{body}$.

Under the answer set semantics, the intended meaning is given by the collection of *answer sets* [10] of the program. Let P^I be the program obtained by adding to the definite clauses of P the rules $p \leftarrow p_1, \ldots, p_n$ such that $p \leftarrow p_1, \ldots, p_n, \textbf{not}\, p_{n+1}, \ldots, \textbf{not}\, p_m$ is in P and $p_{n+1} \notin I, \ldots, p_m \notin I$. A model I of P is an *answer set* if I is the least fixpoint of the operator T_{P^I}. P^I is known as the *reduct* of P w.r.t. I.

When looking for answer sets, there exists a set of atoms shared by all answer sets, while other atoms cannot appear in any answer sets. This suggests the use of 3-valued 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. Observe that in general $I^+ \cup I^- \neq \mathcal{A}$. If $I^+ \cup I^- = \mathcal{A}$, then the 3-interpretation I is said to be *complete*.

Given two 3-interpretations I, J, we use $I \subseteq J$ 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 $I \models p$ if and only if $p \in I^+$; $I \models \textbf{not}\, p$ if and only if $p \in I^-$. $I \models \ell_1 \wedge \cdots \wedge \ell_n$ iff $I \models \ell_i$ for all $i \in \{1, \ldots, n\}$. Given a 3-interpretation I and a set of atoms M, we say that I *agrees* with M if $I^+ \subseteq M$ and $M \cap I^- = \emptyset$.

The immediate consequence operator T_P can be generalized in the case of 3-interpretations as follows

$$T^3_P(I) \quad = \quad \langle \{a \in \mathcal{A} \mid (a \leftarrow \mathit{body}) \in P, I \models \mathit{body}\} \cup I^+, I^- \rangle \tag{2}$$

Iterations on T^3_P inherit the notation from the 2-valued ones. Moreover, T^3_P is monotone and continuous and therefore given an interpretation I, a deterministic extension $T^3_P \uparrow \omega(I)$ can be computed in a countable number of iterations.

The *well-founded model* [24] of a program P (denoted as $\mathsf{wf}(P)$) is a 3-interpretation. 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 answer sets of P. It is well-known that a general program P has a unique well-founded model $\mathsf{wf}(P)$ [24]. If $\mathsf{wf}(P)$ is complete then it is also an answer set (and it is the unique answer set of P).

A well-founded model can be computed deterministically using the idea of alternating fixpoints (as described in [25]). This technique uses pairs of 2-interpretations (denoted by I and J) for building the 3-interpretation $\mathsf{wf}(P)$. The immediate consequence operator is extended with the introduction of an interpretation J:

$$T_{P,J}(I) \quad = \quad \left\{ a \in \mathcal{A} \, : \, \left(\begin{array}{l} (a \leftarrow \mathit{body}^+, \textbf{not}\, \mathit{body}^-) \in P, \\ I \models \mathit{body}^+, \\ (\forall p \in \mathit{body}^-)(J \not\models p) \end{array} \right) \right\} \tag{3}$$

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

$$\left\{ \begin{array}{lll} K_0 = \mathsf{lfp}(T_{P^+,\emptyset}) & U_0 = \mathsf{lfp}(T_{P,K_0}) & \\ K_i = \mathsf{lfp}(T_{P,U_{i-1}}) & U_i = \mathsf{lfp}(T_{P,K_i}) & i > 0 \end{array} \right.$$

where P^+, used for computing K_0, is the subset of P composed of definite clauses, and $\mathsf{lfp}()$ is the least fixpoint operator. When $(K_i, U_i) = (K_{i+1}, U_{i+1})$, the fixpoint is reached and the well-founded model is the 3-interpretation: $\mathsf{wf}(P) = \langle K_i, \mathcal{A} \setminus U_i \rangle$.

3 Computation-based characterization of answer sets

The computation model of GASP is the result of recent investigations about alternative models to characterize semantics for various extensions of ASP—e.g., programs with *abstract constraint atoms* [22].

3.1 Computations and Answer Sets

The work described in [16] provides a *computation-based* characterization of answer sets for programs with abstract constraints. One of the outcomes of that research is the development of a computation-based view of answer sets for logic programs. As recalled in Section 2, the original definition of answer sets [10] requires guessing an interpretation and successively validating it, through the notion of reduct (P^I) and the computation of the minimum model of a definite program.

The characterization of answer sets derived from [16] does not require the initial guessing of a complete interpretation; instead it combines the guessing process with the construction of the answer set. 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. Let us present this alternative characterization in the case of propositional programs.

Definition 3.1 (Computation) *A* computation *of a program P is a sequence of 2-interpretations $I_0 = \emptyset, I_1, I_2, \ldots$ satisfying the following conditions:*

- $I_i \subseteq I_{i+1}$ *for all* $i \geq 0$ *(*Persistence of Beliefs*)*
- $I_\infty = \bigcup_{i=0}^{\infty} I_i$ *is such that* $T_P(I_\infty) = I_\infty$ *(*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* $a \leftarrow body$ *in* P *such that* $I_j \models body$ *for each* $j \geq i$ *(*Persistence of Reason*).*

We say that a computation $I_0, I_1, \ldots$ converges to I if $I = \bigcup_{i=0}^{\infty} I_i$. The results presented in [16] imply the following theorem.

Theorem 3.2 *Given a ground 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.* □

3.2 A Refined View of Computation

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

- The Persistence of Beliefs rule and 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—as defined next).

These observations are in the same spirit as the *expand* operation used by SMODELS and the transformation-based computation of the well-founded semantics investigated in [25]. These two observations allow us to rephrase the notion of computation in the context of 3-interpretations, leading to the introduction of the notion of GASP-computation, as follows.

Definition 3.3 (applicable) *Given a ground rule $a \leftarrow body$ and a 3-interpretation I, we say that the rule is* applicable *w.r.t. I if*

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

A non-ground rule R is applicable *w.r.t. I if and only if there is a grounding r of R that is applicable w.r.t. I.*

We wish to point out that the requirement is weaker than requiring that $I \models body$. Moreover, observe that the applicability is not a monotone property. Consider for instance the 3-interpretations $I = \langle \{q\}, \emptyset \rangle$ and $J = \langle \{q, s\}, \emptyset \rangle$. Then, $I \subseteq J$ and the rule $p \leftarrow q, \textbf{not}\ s$ is applicable w.r.t. I but not w.r.t. J.

Let us also introduce here the notion of *declarative closure*: given a program P and a 3-interpretation I, the declarative closure $DCL_P(I)$ of P w.r.t. I is defined as

$$DCL_P(I) \quad = \quad T_P^3 \uparrow \omega(I) \tag{4}$$

where T_P^3 is defined in equation (2). $DCL_P(I)$ is the smallest extension of the 3-interpretation I that includes all positive atoms that are implied by the application of the rules in P according to I.

The refined notion of computation adds inference power to the non-deterministic expansion step. This inference is delegated to an operator defined as follows:

Definition 3.4 *Let us define an the operator Φ_P, from 3-interpretations to 3-interpretations. Φ_P is an* expanding operator *if it satisfies the following properties:*

- *Φ_P is monotone, i.e., $I \subseteq I'$ implies $\Phi_P(I) \subseteq \Phi_P(I')$;*
- *Φ_P introduces at least the positive atoms of the declarative closure, namely $DCL_P(I) \subseteq \Phi_P(I)$;*
- *Φ_P is answer set preserving, i.e.,*

$$\left\{ X \,\middle|\, \begin{array}{l} X \text{ answer set of } P, \\ I^+ \subseteq X, I^- \cap X = \emptyset \end{array} \right\} = \left\{ X \,\middle|\, \begin{array}{l} X \text{ answer set of } P, \\ \Phi_P(I)^+ \subseteq X, \Phi_P(I)^- \cap X = \emptyset \end{array} \right\}$$

Let us observe that $I^- \cap I^+ = \emptyset$ and $\Phi_P(I)^+ \cap \Phi_P(I)^- = \emptyset$. Moreover, since $DCL_P(I) \subseteq \Phi_P(I)$ and $I \subseteq DCL_P(I)$ then $I \subseteq \Phi_P(I)$.

Example 3.5 *We give here two possible choices for an expanding operator.*

The first one is $\Phi_P = DCL_P$. In this case Φ_P allows only the expansion of the interpretation of the positive atoms.

A second, more complex, example can be constructed by modifying the program transformations of [25]. In this case, Φ_P can expand both positive and negative components of the interpretation. Given a 3-interpretation I, let us introduce the transformation *of the program P according to I as*

$$\begin{array}{rcl} \mathcal{RED}_P(I) & = & \left\{ a \leftarrow body' \,\middle|\, \begin{array}{l} (a \leftarrow body) \in P, \\ body^+ \cap I^- = \emptyset, body^- \cap I^+ = \emptyset, \\ body'^+ = body^+ \setminus I^+, body'^- = body^- \setminus I^- \end{array} \right\} \\ \mathcal{TR}_P(I) & = & \{ a \leftarrow body \mid (a \leftarrow body) \in \mathcal{RED}_P(I), a \notin \mathcal{U}(\mathcal{RED}_P(I)) \} \end{array}$$

where $\mathcal{U}(P)$ denotes the largest unfounded set of P [24]. Then we use these notions for defining the operator Φ_P. Given a 3-interpretation I, let us define the following mapping between interpretations:

$$\begin{array}{rcl} \Psi_P^I(J) & = & I \cup \langle \{a \mid (a \leftarrow) \in \mathcal{TR}_P(J)\}, \{a \mid \forall (r \in \mathcal{TR}_P(J)).\, (head(r) \neq a)\} \rangle \\ \Phi_P(I) & = & \Psi_P^I \uparrow \infty \end{array}$$

Let us observe that, in the case of $I = \langle \emptyset, \emptyset \rangle$, we have that $\Phi_P(\langle \emptyset, \emptyset \rangle) = \mathsf{wf}(P)$ [25].

Definition 3.6 (GASP Computation) *Let Φ be an expanding operator. A* GASP computation *of a program P is a sequence of 3-interpretations $I_0, I_1, I_2, \ldots$ that satisfies the following properties:*

- *$I_0 = \langle \emptyset, \emptyset \rangle$*
- *$I_i \subseteq I_{i+1}$ for all $i \geq 0$ (*Persistence of Beliefs*)*
- *if $I = \bigcup_{i=0}^{\infty} I_i$, then $T_P^3(\langle I^+, \mathcal{A} \setminus I^+ \rangle) = \langle I^+, \mathcal{A} \setminus I^+ \rangle$ and $\langle I^+, \mathcal{A} \setminus I^+ \rangle$ is a model of P (*Convergence*)*
- *for each $i \geq 0$ there is a set of rules $Q \subseteq \{(a \leftarrow body) \in P \mid a \leftarrow body$ is applicable w.r.t. $I_i\}$ such that $I_{i+1} = \Phi_P(I_i \cup \langle \bigcup_{a \leftarrow body \in Q} body^+, \bigcup_{a \leftarrow body \in Q} body^- \rangle)$ (*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*).*

*A sequence of 3-interpretations fulfilling the points above save for the fact that $I_0 = A$, where A is a possibly non-empty 3-interpretation, is said to be a A-*GASP computation.

Note that the notion of Convergence implies that the computation converges to a supported model of the program. Let us draw the proofs of correctness and completeness of GASP-computation w.r.t. the answer sets of a program P.

Theorem 3.7 (correctness) *Given a program P, if there exists a GASP computation that converges to a 3-interpretation I, then I^+ is an answer set of P.*

Proof.

Let us consider a GASP computation $I_0 = \langle \emptyset, \emptyset \rangle, I_1, I_2, \ldots$ and let us denote with I the convergence point of the computation. For the sake of readability, let us denote with $Q = P^{I^+}$ the reduct of the program P with respect to I^+. We show that $\mathsf{lfp}(T_Q) = I^+$.

1. First, let us prove by induction on i that, for all $i \geq 0$, $I_i^+ \subseteq T_Q \uparrow i$. By proving this we will be able to infer that $I^+ \subseteq \mathsf{lfp}(T_Q)$.

   For $i = 0$, $I_i^+ = \emptyset$ and therefore the result is obvious.

   Let us assume that $I_i^+ \subseteq T_Q \uparrow i$ for a given i, and let $a \in I_{i+1}^+$.

- If a is also in I_i^+ then $a \in T_Q \uparrow i$, and hence, for monotonicity of T_Q, $a \in T_Q \uparrow (i+1)$.
- Let us now consider $a \in I_{i+1}^+ \setminus I_i^+$. Because of the persistence of reason, there exists a rule

$$a \leftarrow \mathit{body}^+, \textbf{not}\ \mathit{body}^-$$

such that for each $k \geq i$ we have $I_k \models \mathit{body}^+$ and $I_k \models \textbf{not}\,\mathit{body}^-$. Since the sequence I_k is monotone, then we can easily see that $I \models \mathit{body}^+$ and $I \models \textbf{not}\ \mathit{body}^-$. Since the reduct Q is computed using I, we obtain that

$$a \leftarrow \mathit{body}^+$$

is a rule in Q. Since $I_k \models \mathit{body}^+$ for all $k \geq i$ in particular $I_i \models \mathit{body}^+$. Since $I_i^+ \subseteq T_Q \uparrow i$ (inductive hypothesis) $\mathit{body}^+ \subseteq T_Q \uparrow i$. Therefore $a \in T_Q \uparrow (i+1)$. This allows us to conclude that $I_{i+1}^+ \subseteq T_Q \uparrow (i+1)$.

Thus, for all i, we have that $I_i^+ \subseteq T_Q \uparrow \omega$. Therefore $I^+ = \bigcup_{i=0}^{\infty} I_i^+ \subseteq T_Q \uparrow \infty = \mathsf{lfp}(T_Q)$.

2. Let us now prove that $\mathsf{lfp}(T_Q) \subseteq I^+$. Because of the Convergence condition, we know that I^+ is a model of P. Since models of a program are also models of the reduct, then I^+ is a model of Q; being $\mathsf{lfp}(T_Q)$ the minimal model of Q, then $\mathsf{lfp}(T_Q) \subseteq I^+$. □

The proof of completeness of the GASP computation can be derived as follows. We use the more general notion of A-GASP computation where $I_0 = A$ (instead of $I_0 = \langle \emptyset, \emptyset \rangle$—see Def. 3.6).

Lemma 3.8 *Let M be an answer set of P and let A a 3-interpretation that agrees with M; then $\Phi_P(A)$ agrees with M.*

Proof. The result is an immediate consequence of the answer set preservation property of Φ_P. □

Lemma 3.9 *Let M be an answer set of P and let A be a partial 3-interpretation that agrees with M. There exists an A-GASP computation that converges to M.*

Proof. By hypothesis, A agrees with M. Therefore, $A^+ \subseteq M$ (and $M \cap A^- = \emptyset$). Let us prove the result by induction on the size of the difference set $|M \setminus A^+|$.
The base case is $|M \setminus A^+| = 0$, hence $M = A^+$. If we consider the computation composed of $A = I_0 = I_1 = I_2 = \cdots$ we have a correct A-GASP computation that converges to M. In fact

- it has persistence of beliefs, since $I_i = I_{i+1}$
- the convervenge point is $\langle M, \mathcal{A} \setminus M \rangle$, which is clearly a model since M is an answer set of P
- the revision principle is satisfied by simply taking $Q = \emptyset$ at each step (and from the fact that Φ_P is answer set preserving)
- persistence of reason derives from the fact that each answer set of a program is a supported model.

Let us now consider the inductive step: assume that $A^+ \subset M$. We would like to argue that there exists a rule that is applicable in A, it will persist in a computation that converges to M, and the rule has head which appears in $M \setminus A^+$. Let us consider the computation for an answer set induced by the 2-valued immediate consequence operator T_{P^M}, where P^M is the reduct of P w.r.t. M. Precisely, let us consider the sequence of sets of atoms $X_0, X_1, \ldots$ where $X_i = T_{P^M} \uparrow i$. Let us also define $\mathit{ADD}_i = (X_i \setminus X_{i-1}) \cap (M \setminus A^+)$; these are the atoms added in the i-th step of the basic computation (i.e., the computation obtained by iterating the immediate consequence operator of the reduct of the program) which are not covered by A.

Since $A^+ \subset M$ there must be an index j such that $\mathit{ADD}_j \neq \emptyset$. Let j be the smallest index with such property, and let us consider an atom $a \in \mathit{ADD}_j$.

Since $a \in \mathit{ADD}_j$, then $a \in X_j \setminus X_{j-1}$, which in turn implies that there exists a rule $a \leftarrow \mathit{body}^+$ in P^M such that $\mathit{body}^+ \subseteq X_{i-1}$ and therefore a rule

$$a \leftarrow \mathit{body}^+, \textbf{not}\ \mathit{body}^-$$

in P such that (1) $\mathit{body}^+ \subseteq X_{i-1}$ and (2) $\mathit{body}^- \cap M = \emptyset$.

Observe that, because of the minimality of the index j, we must have that $\mathit{body}^+ \subseteq A^+$. Note that it is impossible to have $\mathit{body}^- \cap A^+ \neq \emptyset$—if this was the case, then there would be an atom $b \in A^+ \cap \mathit{body}^-$, and since $A^+ \subseteq M$, then we would also have $M \cap \mathit{body}^- \neq \emptyset$.

This means that the above rule is applicable in A.

Let us construct the first step of the desired A-computation, by stating $I_0 = A$ and $I_1 = \Phi_P(A \cup \langle \mathit{body}^+, \mathit{body}^- \rangle)$. It holds that $A \cup \langle \mathit{body}^+, \mathit{body}^- \rangle$ agrees with M. As a matter of fact, $A^+ \cup \mathit{body}^+ \subseteq M$ since $\mathit{body}^+ \subseteq A^+$ and $A^+ \subseteq M$. Furthermore, the hypothesis above about the choice of the rule states that $\mathit{body}^- \cap M = \emptyset$; since $A^- \cap M = \emptyset$, then $(A^- \cup \mathit{body}^-) \cap M = \emptyset$.

From Lemma 3.8 we can conclude that also I_1 agrees with M, which allows us to invoke the inductive hypothesis and construct the rest of the computation. Note in particular that the rule chosen above will remain applicable throughout the computation because of the persistence of belief property of computations.□

Theorem 3.10 (completeness) *Given a program P and an answer set M of P, there exists a GASP computation that converges to M.*

Proof. Immediate from lemma 3.9, since the empty 3-interpretation $\langle \emptyset, \emptyset \rangle$ agrees with M. □

Corollary 3.11 *Given a program P and an answer set M of P, there exists a* $\mathsf{wf}(P)$*-GASP computation that converges to M.*

Proof. Immediate from lemma 3.9, since the well-founded model of a program P agrees with any stable model M (see, e.g., [2]). □

Corollary 3.12 *Given a program P and an answer set M of P, there exists a GASP computation that converges to M where the cardinality of the sets Q in the Revision steps are such that $|Q| \leq 1$.*

Proof. This is obvious from the proof of Lemma 3.9. □

4 Computing models using Constraint Logic Programming

In this section we show how to encode and handle interpretations and answer sets in Prolog using Constraint Logic Programming over finite domains (briefly, FD).

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 $p(t_1, \ldots, t_n)$, 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 LPARSE+SMODELS system impose syntactic restrictions to facilitate the grounding process and ensure finiteness of the collection of ground clauses. In particular, LPARSE requires each rule in the program to be *range restricted*, i.e., all variables in the rule should occur in body^+. Furthermore, LPARSE 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.[1]

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 $p(X_1, \ldots, X_n)$ 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 $(a_1, a_2, \ldots, a_n)$, where $a_i \in \mathit{Consts}(P)$—$\mathit{Consts}(P)$ denotes the set of constants in the language used by the program P.

The explicit representation of the set of tuples has the maximal cardinality $|\mathit{Consts}(P)|^n$. The idea is to use a more compact representation based on the constraint-based data structure FDSETs, after a mapping of tuples to integers. We describe below the details of this idea.

Without loss of generality, we assume that $\mathit{Consts}(P) \subseteq \mathbb{N}$. Each tuple $\vec{a} = (a_0, \ldots, a_{n-1})$ is mapped to the *unique* number

$$\mathsf{map}(\vec{a}) = \sum_{i \in [0..n-1]} a_i \mathbb{M}^i \tag{5}$$

where $\mathbb{M}$ is a "big number", $\mathbb{M} \geq |\mathit{Consts}(P)|$. In case of predicates without arguments (predicates of arity 0), for the empty tuple () we set $\mathsf{map}(()) = 0$. We also extend the map function to the case of non-ground tuples, using FD variables. If $\vec{Y} = (y_1, y_2, \ldots, y_n)$, where $y_i \in \mathit{Vars}(P) \cup \mathit{Consts}(P)$, then $\mathsf{map}(\vec{Y})$ is the FD constraint that represent the sum defined above. For instance, if $\vec{Y} = (3, X, 1, Z)$ and $\mathbb{M} = 10$, then $\mathsf{map}(\vec{Y}) = 3 + X * 10 + 1 * 10^2 + Z * 10^3$. Moreover, all variables possibly occurring in $\vec{Y}$ are constrained to have domain $0..\mathbb{M} - 1$.

[1]Some of these restrictions have been relaxed in other systems, e.g., DLV.

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

$$(p, n, \texttt{POS}_{p,n}, \texttt{NEG}_{p,n}),$$

one for each predicate symbol, where p is the predicate name, n its arity, and $\texttt{POS}_{p,n}$ and $\texttt{NEG}_{p,n}$ are sets of integers defined in the following way:

$$\texttt{POS}_{p,n} = \{\mathsf{map}(\vec{Y}) : I \models p(\vec{Y})\} \qquad \texttt{NEG}_{p,n} = \{\mathsf{map}(\vec{Y}) : I \models \mathbf{not}\, p(\vec{Y})\}$$

Sets of integer numbers can be efficiently stored and handled using the data structure FDSET provided by the clpfd library of SICStus Prolog. A set $\{a_1, a_2, \ldots, a_n\}$, where $a_1, a_2, \ldots, a_n$ are integer constants, is interpreted as the union of a set of disjoint intervals. Precisely, the i-th interval can be identified by the lower bound a_{b_i} and by the upper bound a_{e_i}. Therefore, the interval set can be represented by $[a_{b_1}, a_{e_1}], \ldots, [a_{b_k}, a_{e_k}]$ (where $a_{e_i} < a_{b_{i+1}} + 1$) and stored consequently as $[[a_{b_1}|a_{e_1}], \ldots, [a_{b_k}|a_{e_k}]]$. clpfd provides a library of built-in predicates for dealing with this data structure. Using FDSETs, the complexity of constraint propagation depends on the number of intervals constituting the set, instead than on the size of the sets.

In our implementation, we exploit FDSETs to represent the sets $\texttt{POS}_{p,n}$ and $\texttt{NEG}_{p,n}$ used to store interpretations for each predicate p^n. This will help us in an efficient implementation of the various immediate consequence operators.

The mapping of tuples to integers and then to FDSETs is illustrated by the following simple example.

Example 4.1 *Let* $\{(0,0,1), (0,0,2), (0,0,3), (0,0,8), (0,0,9), (0,1,0), (0,1,1), (0,1,2)\}$ *be the set of tuples that represent the positive part of the interpretation of a ternary predicate p. If* $\mathbb{M} = 10$*, then*

$$\begin{aligned}\texttt{POS}_{p,3} &= \{\mathsf{map}(0,0,1), \mathsf{map}(0,0,2), \mathsf{map}(0,0,3), \mathsf{map}(0,0,8),\\ &\quad\ \mathsf{map}(0,0,9), \mathsf{map}(0,1,0), \mathsf{map}(0,1,1), \mathsf{map}(0,1,2)\}\\ &= \{1,2,3,8,9,10,11,12\}\end{aligned}$$

Using FDSETs, this set is represented compactly by the set of two intervals

$$[[1|3], [8|12]]$$

4.2 Minimal Model Computation

We start by showing how the computation of T^3_P (see equation (2)) can be implemented using finite domain constraints. 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 rule and interpretation I, we build a CSP that characterizes the atoms $p^n(\vec{X})$ derivable from the rule body.

More precisely, let r:

$$p_0(\vec{Y_0}) \leftarrow p_1(\vec{Y_1}), \ldots, p_k(\vec{Y_k}), \mathbf{not}\, p_{k+1}(\vec{Y}_{k+1}), \ldots, \mathbf{not}\, p_j(\vec{Y_j})$$

be the selected rule of P, with $|\vec{Y_i}| = n_i$ be the arity of the predicate p_i and $\vec{Y_i} \in (\mathit{Vars}(P) \cup \mathit{Consts}(P))^{n_i}$. Note that $\vec{Y_i}$ may contain repeated variables and it can share variables with other literals in the clause.

All variables occurring in the rule are assigned to the finite domain $0..\mathbb{M}-1$. Moreover, for each (positive) atom p_i we introduce a FD variable V_i to represent the possible tuples associated with p_i. The domain of V_i is $\texttt{POS}_{p_i,n_i}$, where $(p_i, n_i, \texttt{POS}_{p_i,n_i}, \texttt{NEG}_{p_i,n_i})$ is part of the current interpretation I. In addition, V_i is set equal to $\mathsf{map}(\vec{Y_i})$ in order to establish a connection between the individual variables in $\vec{Y_i}$ and the variable V_i. The added constraint is:

$$V_i \in \texttt{POS}_{p_i,n_i} \wedge V_i = \mathsf{map}(\vec{Y_i})$$

For each literal **not** p_j, we use the same scheme, but the domain of the FD variables is $\texttt{NEG}_{p_i,n_i}$.

Finally, for the head of the rule $p_0(\vec{Y_0})$, we similarly define a FD variable V_0, and we add the constraints expressing the fact that the head is not yet in the positive part of the interpretation I being built (to avoid unnecessary rule applications) and it is not in the negative part of I (to avoid inconsistent rule applications):

$$V_0 = \mathsf{map}(\vec{Y_0}) \wedge V_0 \notin \texttt{POS}_{p_0,n_0} \wedge V_0 \notin \texttt{NEG}_{p_0,n_0}.$$

If C_r is the total constraint associated with the selected rule r, and $\mathit{Vars}(r)$ is the set of variables possibly occurring in r, then the new interpretation for the head predicate p_0 of r is

$$\langle p_0, n_0, \texttt{POS}'_{p_0,n_0}, \texttt{NEG}_{p_0,n_0} \rangle$$

where

$$\texttt{POS}'_{p_0,n_0} = \texttt{POS}_{p_0,n_0} \cup Sol(C_r)|_{V_0}$$

where $Sol(C_r)$ denotes the set of all the solutions of the constraint C_r, namely the variable assignments that satisfy the constraint C_r. This set is projected on the variable V_0 storing the tuple of the rule head.

The application of $T^3_P(I)$ using the FD constraint technique described above is illustrated by the following simple example.

Example 4.2 *Let*

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

be the selected rule and assume the current interpretation I is

$$\langle\{q(1,1), q(1,2), q(2,2), p(1)\}, \emptyset\rangle.$$

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 set $\{0,\ldots,\mathbb{M}-1\} = \{0,1,2\}$ is the initial domain for V_0 and V_2, while $V_1 \in \{4,7,8\}$. Using constraint propagation, the 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.

In the current prototype, the computation of T^3_P is implemented by a fixpoint procedure that calls recursively a procedure `apply_def_rule` until the interpretation cannot be further modified. The procedure `apply_def_rule` associates the CSP with the selected rule in P, and then it solves this problem by using the FD solver over FDSETs. As discussed above, the sets `POS` and `NEG` are implemented by using FDSETs. Membership constraints are handled using the `in_set` predicate provided by the `clpfd` library of SICStus Prolog. The addition of the new instances of the head predicate to the current interpretation is implemented via library operations on FDSETs (namely, `fdset_union`). Finally, the set denoted by $Sol(C_r)$ is computed by using the standard `findall` built-in predicate of Prolog and the `labeling` facility of `clpfd`. Precisely, the set `DeltaPOS` of new tuples can be computed as:

$$\texttt{findall}(X, (X = V_0, C_r, \texttt{labeling}(Vars(r))), \texttt{DeltaPOS})$$

`findall` collects all the ground instances of $\vec{Y_0}$ that satisfy the CSP and, therefore, represent new positive instances of the head that must be added to the current interpretation. Grounding is performed locally within the `findall` computation, making sure that useless or redundant groundings are avoided a priori.

Observe that this method explicitly produces all solutions, and this could be inefficient in the case of large domains. An implicit construction of the set of all solutions could improve this situation but it would require features not supported by current solvers.

In section 5.1 we experimentally show the effectiveness of using a compact representation of tuples (FDSETs) and the importance of building the CSP domains from them. In particular we compare this approach with other reasonable ones.

4.3 Well-founded Model Computation

Computing a well-founded model is a deterministic step during GASP computation. As done for T^3_P in the previous section, the implementation is based on FD constraint programming and the FDSET representation of interpretations.

The implementation boils down to controlling the alternating fixpoint computation [25] and to encode the $T_{P,J}$ operator (see equation 3).

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

As for the T^3_P, we associate a FD constraint with each rule and we obtain the set of the new tuples in the rule head which are derivable from the rule body by solving the constraint. The difference between this constraint and the corresponding one used in the computation of T^3_P is that, by definition of $T_{P,J}$, the negative part of the rule may not appear in J^+.

This is obtained by requiring that, for each literal $\mathbf{not}\, p_j^{n_j}$, $V_j \notin Pos'$, with $(p_i^{n_i}, n_i, \texttt{POS}', \texttt{NEG}') \in J$.

4.4 Computing Answer Sets

The complete answer set enumeration is based on the well-founded model computation, alternated with a non-deterministic choice phase. The computation starts by determining the well-founded model I of P. This is justified by Corollary 3.11. If the model is complete, we return the result. Otherwise, if there are some unknown literals, we proceed with the answer set computation, using I as the initial interpretation. We use the generic notion of expanding operator Φ_P (Def 3.4). Each call to Φ_P can detect inconsistent interpretations (failed I). In Figure 1, we provide the pseudocode of the algorithm.

```
(1)    GASP_computation(P)
(2)       I = wf(P)
(3)       rec_search(P,I)

(4)    rec_search(P,I)
(5)       $R$ = applicable_rules(I)
(6)       if ($R = \emptyset$ and I is a model) then return I        { I is an answer set }
(7)       else select $(a \leftarrow body^+, \textbf{not}\, body^-) \in R$
(8)          I = $\Phi_P(I \cup \langle \emptyset, \{body^-\} \rangle)$
(9)          if ($I^+ \cap I^- \neq \emptyset$) then rec_search(P,I)
(10)         else return fail
```

Figure 1: The answer set computation

Each applicable rule represents a non-deterministic choice in the computation of an answer set. The answer set computation explores each of these choices (line 7), and computes I_{i+1} using the Φ_P operator applied to $I_i \cup \langle \emptyset, \{body^-\} \rangle$ (line 8), as defined in the GASP computation. Since for any 3-interpretation I, $DCL_P(I) \subseteq \Phi_P(I)$, we know that a is 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, but it should be noted that each ground rule is produced only once along each branch, due to the constraints introduced. Let us observe that every time the local grounding in invoked, a CSP is built.

The process may encounter a contradiction while adding a new predicate $a \in I^-$ to I^+ or, vice versa, $a \in I^+$ to I^-. In this case we report failure. Whenever there are no more applicable rules, a leaf in the search tree is reached (line 6) and the corresponding answer set is obtained (convergence property).

The applicable rules w.r.t. an interpretation I_i are determined (line 5) as defined in the GASP computation, i.e., solving the CSP using FD and FDSETs with similar techniques to the ones described in the previous sections. In particular, we have chosen as Φ_P the fixpoint of T^3_P (namely we compute the declarative closure $DCL_P(I)$).

Let us briefly show that the notion of GASP computation (precisely, of wf(P)-GASP computation) is implemented by this algorithm.

Theorem 4.3 *The procedure* `GASP_computation(P)` *in Figure 1 returns* `I` *if and only if I is an answer set of* `P`.

Proof Sketch. Thanks to Corollary 3.11 and Theorem 3.7, it is sufficient to show that the procedure in Figure 1 generates all and only the wf(P)-GASP computations that select at most one applicable rule at each Revision step (see Corollary 3.12).

First of all, let us argue that the procedure indeed generates wf(P)-GASP computations of the desired form:

- The Persistence of Beliefs holds, since Φ_P is monotone. As soon as an atom is introduced (either positively or negatively) in a model in line (8), it will persist in the rest of the computation.
- The rule r selected from R is applicable (i.e., $body^+ \in I^+$). Since $body^-$ is added to I^- in line (8) and the persistence of beliefs holds, then r will be applicable in the rest of the computation.
- The Persistence of Reason holds. Let us consider two cases. If a is the head of the selected rule, then it follows from the previous points. If, instead, a is introduced by the Φ_P computation, namely, in our implementation, by the computation $DCL_P(I) = T^3_P \uparrow \omega(I)$ then a has been introduced as head of an applicable rule in the iteration fixpoint process. The persistence of beliefs ensures that the same rule will be applicable in the rest of the computation.
- The Convergence Property holds; when $R = \emptyset$, this implies that the computation will continue from that point on with identical interpretations.

Finally, let us note that the non-deterministic choice in Line `(7)` enables us to explore all possible choices of applicable rules. □

Example 4.4 *In Figure 2 we depict the evolution of the computation for stable models for the program P reported in the first box.*

In the box I_0 we show the internal representation of the interpretation after the application of `wf` *operator for P. Note that* `r(1)` *and* `s(1)` *are the only two atoms that are undefined. Observe that, for the convenience of the implementation, the representation of interpretations may actually covers atoms that do not actually appear in the original program (as indicated by the presence of* `maxint` *as the end-point of the intervals). This does not endanger the correctness of the implementation, since all atoms not appearing in the program are immediately assigned false truth value. For the sake of readability, when building the CSPs, the indexes of the variables are related to the respective predicate names, instead than to consecutive integers. Moreover, we have written directly X rather than* $\mathsf{map}(X)$*.*

At the next step, two rules are recognized as applicable, leading to a non-deterministic choice—and the figure shows the two CSPs used to identify the applicable ground rules.

The boxes labeled I_1 describe the internal representation of the interpretations obtained after applying one of the rules and performing the DCL_P computation. These are the two answer sets of the program.

4.5 Search heuristics

When enumerating answer sets 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 allows the exploration of equivalent branches, employing the same rules, applied in different orders, while the interpretation converges to the same set.

In [6], we explicitly avoid the computation of identical answer sets in distinct branches. The search is stopped when the list of applied rules along the current branch is a subsequence of a permutation of the applied rules in a previously visited branch. However, this approach is inefficient, since a permutation is detected by maintaining an ordering of previously applied rules. Moreover, this approach is based on a simple test on permutations, that does not infer any extra information on the answer sets being computed.

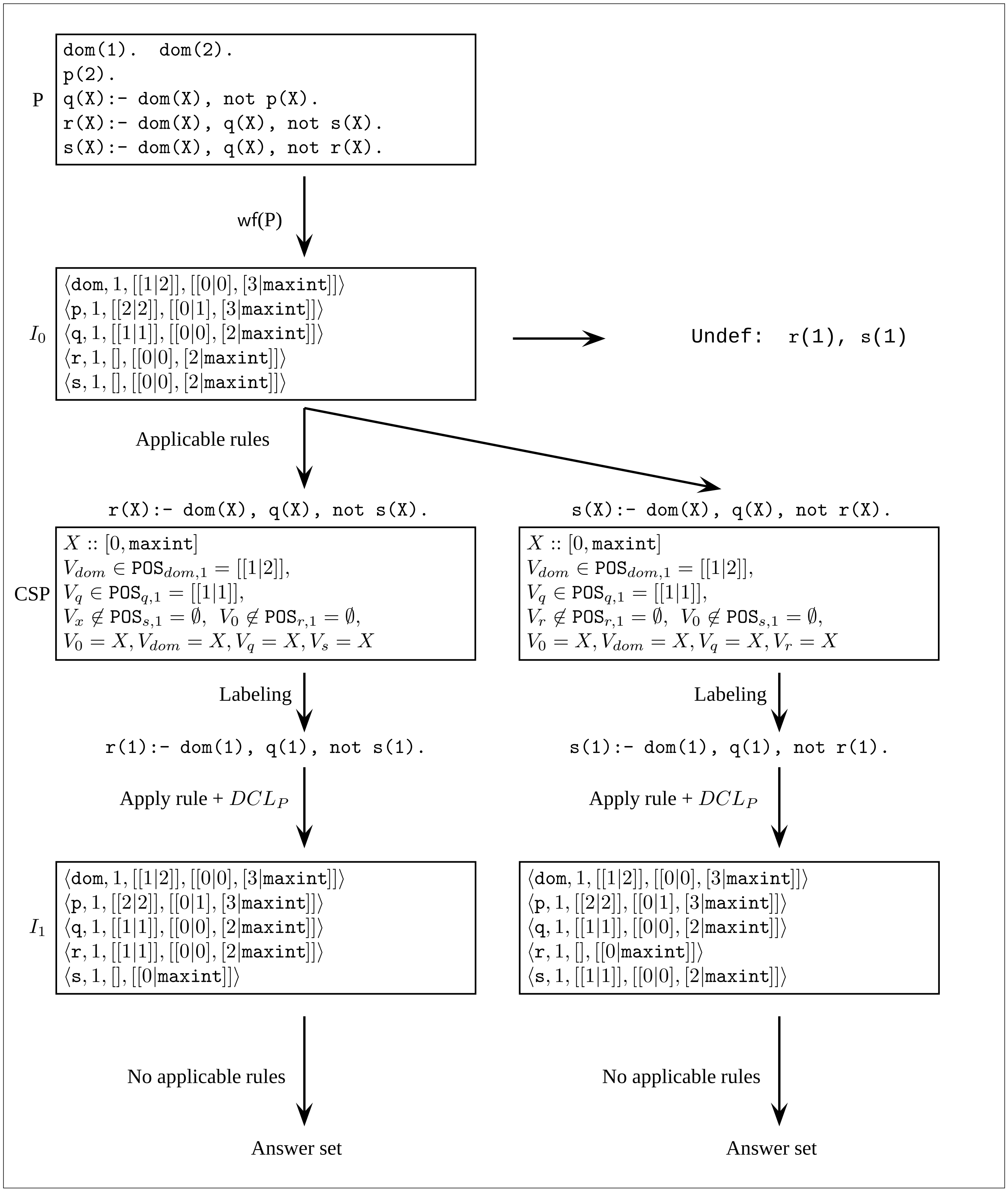
In the current version of GASP, we adopt a strategy that is very similar to the one used in [12]. When a rule R is applicable at a choice point n, there are two choices: either R is applied or R is not applied. The choice opens two distinct branches from n, where R is applied in the left branch, while the right branch contains a subtree where R may not be applied. Practically, when the right branch is opened, a new rule R' is added to the program. The rule R' is a constraint, stating that at least one negative literal in R is falsified by the interpretation being constructed (note that the positive body is verified, since R is applicable). Formally, R' is the constraint $\leftarrow$ **not** $\mathit{body}^{-}(R)$. This additional rule is processed together with the program during the search in the right subtree, and it can propagate some information using the propagators for constraint rules (see Section 4.6).

In our implementation, when a choice point n is reached, we compute the complete set of applicable rules at n. Let us assume that there are $k > 0$ rules (if $k = 0$ then the computation will halt). We create k branches from node n. We associate to the branch number i the application of rule R_i in the set. In order to avoid redundancies, each subtree of a branch i may not contain any applicable rule $R_{j<i}$, since that combination would have been already explored before. To realize this, each branch i extends the current program with a set of new constraints ($R'_{j<i}$) that forbid the application of rules $R_{j<i}$ in the subtree.

4.6 Dependency Graph and Propagation

In order to speed-up the computation of the fixpoints of the various T_P, we introduced several optimizations. At each step of the application of the T_P operator, the rules of the program P are checked and, when their bodies are in the model, the corresponding heads are added to the model. Instead of testing every rule in the program P, the subset of the potential rules that could add new literals to the answer set can be computed, based on the literals introduced at the previous iteration. This subset can be considerably smaller than P, and thus can reduce the number of operations to reach the fixpoint. From the implementation point of view, it is sufficient to precompute a dependency graph of the predicates in the program and to collect the rules that contain the literals added during the previous application of T_P operator. This simple idea improves the running time of [6] of more than 30%.

In order to prune the search and to reduce the number of choice points, we introduced two propagators that are able to infer deterministically some negative literals that can not be produced by T_P and well founded computations. The ideas presented below represent a generalization of some of the techniques that drive the search in SMODELS. In particular, we deal with non ground rules and therefore we introduce a CSP-based



```
dom(1).  dom(2).
p(2).
q(X):- dom(X), not p(X).
r(X):- dom(X), q(X), not s(X).
s(X):- dom(X), q(X), not r(X).
```

$\langle \texttt{dom}, 1, [[1|2]], [[0|0], [3|\texttt{maxint}]]\rangle$
$\langle \texttt{p}, 1, [[2|2]], [[0|1], [3|\texttt{maxint}]]\rangle$
$\langle \texttt{q}, 1, [[1|1]], [[0|0], [2|\texttt{maxint}]]\rangle$
$\langle \texttt{r}, 1, [], [[0|0], [2|\texttt{maxint}]]\rangle$
$\langle \texttt{s}, 1, [], [[0|0], [2|\texttt{maxint}]]\rangle$

```
Undef:  r(1), s(1)
```

```
r(X):- dom(X), q(X), not s(X).
```

$X :: [0, \texttt{maxint}]$
$V_{dom} \in \texttt{POS}_{dom,1} = [[1|2]],$
$V_q \in \texttt{POS}_{q,1} = [[1|1]],$
$V_x \notin \texttt{POS}_{s,1} = \emptyset,\ V_0 \notin \texttt{POS}_{r,1} = \emptyset,$
$V_0 = X, V_{dom} = X, V_q = X, V_s = X$

```
s(X):- dom(X), q(X), not r(X).
```

$X :: [0, \texttt{maxint}]$
$V_{dom} \in \texttt{POS}_{dom,1} = [[1|2]],$
$V_q \in \texttt{POS}_{q,1} = [[1|1]],$
$V_r \notin \texttt{POS}_{r,1} = \emptyset,\ V_0 \notin \texttt{POS}_{s,1} = \emptyset,$
$V_0 = X, V_{dom} = X, V_q = X, V_r = X$

```
r(1):- dom(1), q(1), not s(1).
```

```
s(1):- dom(1), q(1), not r(1).
```

$\langle \texttt{dom}, 1, [[1|2]], [[0|0], [3|\texttt{maxint}]]\rangle$
$\langle \texttt{p}, 1, [[2|2]], [[0|1], [3|\texttt{maxint}]]\rangle$
$\langle \texttt{q}, 1, [[1|1]], [[0|0], [2|\texttt{maxint}]]\rangle$
$\langle \texttt{r}, 1, [[1|1]], [[0|0], [2|\texttt{maxint}]]\rangle$
$\langle \texttt{s}, 1, [], [[0|\texttt{maxint}]]\rangle$

$\langle \texttt{dom}, 1, [[1|2]], [[0|0], [3|\texttt{maxint}]]\rangle$
$\langle \texttt{p}, 1, [[2|2]], [[0|1], [3|\texttt{maxint}]]\rangle$
$\langle \texttt{q}, 1, [[1|1]], [[0|0], [2|\texttt{maxint}]]\rangle$
$\langle \texttt{r}, 1, [], [[0|\texttt{maxint}]]\rangle$
$\langle \texttt{s}, 1, [[1|1]], [[0|0], [2|\texttt{maxint}]]\rangle$

Figure 2: Example of GASP computation

analysis similar to the one used to compute the applicability of rules. The resolution of the CSP is designed to avoid the complete grounding of the rules involved. We address two settings where negative literals can be deduced: inferring a literal that appears (i) in the body and (ii) in the head.

Let R be a non-ground ASP rule of the form $\mathit{head}(R) : -\mathit{body}^+(R), \textbf{not}\ \mathit{body}^-(R)$, $I = \langle I^+, I^- \rangle$ be the current model and R' a grounding of R.

The case (i) applies when there exists a particular grounding R' of R such that $\mathit{head}(R') \in I^-$. In this case the rule R' should not become applicable, otherwise $\mathit{head}(R')$ would be added to I^+ and generate a failure. We consider the specific situation in which $\mathit{body}(R')$ is completely satisfied except for exactly one undetermined literal $\ell \in \mathit{body}^+ \setminus (I^+ \cup I^-)$. To prevent the rule R' to fire, the literal ℓ should be set to false in every model computed from I and thus ℓ is added to I^-.

The case (ii) applies when it is possible to deduce that a not yet determined literal $\ell \notin I^+ \cup I^-$ may not be introduced in I^+ in any subsequent computation. The (ground) literal ℓ can be introduced in I^+ only if there is at least one (potentially) applicable rule R' such that $\mathit{head}(R') = \ell$. If some literals $p \in \mathit{body}(R')$ are undetermined, we assume that they can potentially contribute to satisfy the body: i.e., if $p \in \mathit{body}^+(R')$ then p is assumed to be true and if $p \in \mathit{body}^-(R')$ then p is assumed to be false. If there is no such rule R' then the literal ℓ can be safely added to I^-.

During the search, the propagators are executed after every application of a rule. Since they may lead to the addition of new false literals to the answer set, the solver computes a fixpoint that involves the T_P^3 iterations and the propagation phases.

5 Experiments

The prototype implementing the ideas described above and all the tests described in this section, are available at `www.dimi.uniud.it/dovier/CLPASP`. The prototype has been developed using SICStus Prolog 4.0.7,[2] chosen for its rich library of FDSET primitives.

5.1 Testing FDSETs

In order to infer the capabilities of FDSETs for encoding predicates, we compare the computation of the least fixpoint on the two simple definite programs of Figure 3, using different encodings of the models representation. In particular we implemented:

1. **TP tuples**: in this version the model is represented as a list of tuples associated to each predicate. In particular, during the computation of the ground rules that fire the head, the retrieval of a tuple is performed by using `member`. The only CLP(FD) feature used is the computation of arithmetic expressions of rules, after every other atom is grounded.

2. **TP int**: this version differs from the previous one in the representation of tuples. Here each tuple is converted into an integer as described in Equation 5. The values represented the tuples are stored again into lists of integers and `member` calls are used to retrieve the values. This version aims at showing that the additional CLP(FD) operations for the conversion between tuple and integers do not affect the computational time and space.

3. **TP FD_interval**: this version uses a union of FD intervals for the representation of the integers associated to the tuples of a predicate. The representation is written in Prolog and it has the form $A_1..B_1 \vee \ldots \vee A_n..B_n$, where $A_i..B_i$ represents the integers k, with $A_i \leq k \leq B_i$. The intervals are mapped to CLP(FD) domains in order to set up a CSP for the computation of ground rules that can fire the head.

4. **TP GASP**: this is the version used by GASP implementation, as described in Section 4.2. It substitutes the Prolog representation of union of integers intervals by the built-in FDSETs of SICStus Prolog.

We test the four versions of the fixpoint computation on two graph problems reported in Figure 3.

```
q(1).                                    q(1).
q(X):-q(Y),X=Y+1,X<51.                   q(X):-q(Y),X=Y+2,X<100.
p(X,Y) :- q(X),q(Y),Y = X + 1.           p(X,Y) :- q(X),q(Y),Y = X + 2.
h(X,Y) :- p(X,Y).                        h(X,Y) :- p(X,Y).
h(X,Y) :- p(X,Z),h(Z,Y),q(Y).            h(X,Y) :- p(X,Z),h(Z,Y),q(Y).
```

Figure 3: The programs used to test the FDSETdata structure (in both of them, $N = 50$). The leftmost is the "contiguous", the rightmost is the "all singleton".

[2] `www.sics.se/isl/sicstuswww/site/`

N	Contiguous		Singleton		N	Contiguous		Singleton	
	Time	Memory	Time	Memory		Time	Memory	Time	Memory
1: TP tuples					2: TP int				
50	1.68	1.2	1.79	1.1	50	1.45	1.2	1.49	1.1
75	12.2	3.3	12.7	3.3	75	9.86	3.3	9.79	3.3
100	49.2	3.5	51.7	3.1	100	37.6	3.4	39.45	3.4
3: TP FD_interval					4: TP GASP				
50	0.13	1.2	3.86	6.5	50	0.08	1.2	0.16	3.2
75	0.32	3.3	21.64	15.1	75	0.14	1.3	0.46	11.6
100	0.81	3.4	73.17	51.5	100	0.19	3.4	1.21	15.0

Table 1: Running time (in seconds) and memory consumption (in MB) of different encodings of transitive closure

In both programs we compute the transitive closure of a graph consisting, initially, of a unique path (predicate p). In the "contiguous" version we use consecutive node numbers and the edges are $p(1,2), p(2,3), p(3,4), \dots, p(N-1,N)$; in the "all singleton" version, instead p has the semantics $p(1,3), p(3,5), p(5,7), \dots, p(2N-3, 2N-1)$. Although isomorphic, these two programs behave differently under the FDSET encoding. Let us consider, for instance, the case of $N = 5$. In the first case, the set of values $\{1,2,3,4,5\}$ for the predicate `q` is represented by the compact FDSET $[[1|5]]$. In the second case, instead, the set of values $\{1,3,5,7,9\}$ for the predicate `q` is represented by the sparse FDSET $[[1|1],[3|3],[5|5],[7|7],[9|9]]$. The situation for the predicates `p` and `h` is similar.

The tests aim at comparing and evaluating the costs, both in time and space, of the four different approaches listed above. Running time and memory consumptions of the two approaches are sensibly different, as shown in Table 1. The tests were made on a Intel Core Duo 2.66GHz, 3MB cache Linux machine.

It is interesting to note that the conversion of tuples in lists of integers (case 2) pays off the cost of the conversions between the two representations (w.r.t. case 1). The choice of converting tuples into integers is efficient, even if, with large arity tuples, it may limit the choice of a suitable $\mathbb{M}$.

The use of compact intervals (case 3 and 4), works at best with the contiguous problem, where the computational times are sensibly reduced at no extra memory costs. However for case 3, we note that our Prolog implementation of intervals is rather inefficient in terms of intervals operations and memory usage for singleton problems.

The use of built-in FDSET library shows the best compromise, both in time and space, with a significant gain w.r.t. case 1. Even in the singleton case, where each node is represented by a distinct singleton interval, the representation allows to build the CSPs more efficiently. The drawback, in this case, is a penalty of a 5 times in extra memory consumption.

These tests highlight the importance of a compact representation and its direct mapping to domains of the CSP associated to a rule (SICStus built-in `in_set`). We plan, as future work, to investigate more in detail other possible options (e.g. OBDDs). In this version of GASP, we make use of FDSETs since they represents the best trade off between coding-time and efficiency.

5.2 Comparing GASP to other systems

We performed some preliminary experiments, using different classes of ASP programs, and we report the execution times in Table 2. All the experiments have been performed on an Intel Core Duo 2.66GHz, 3MB cache Linux machine. For the ASP tests, we used Lparse 1.1.1 and Smodels 2.33[3]. We also tested some benchmarks using ASPeRiX 0.2[4]. The column with the label time GASP / Lparse + smodels, indicates the ratio between the execution times of GASP and Lparse + Smodels. The last column indicates, when applicable, the times for equivalent programs solved by ASPeRiX.

We used ten benchmark suites. The first suite of experiments (test0) aims at testing the practical running time of computing lfp(T_P) for definite programs. It is exactly the same program (the "contiguous" one) described in the previous section that computes the transitive closure of a graph. The growth of the running time appears to be quadratic in the value N. GASP and Lparse+Smodels behave similarly: their running times differ by a constant ratio. Let us observe that the size of the ground file with $N = 512$ is 3.9MB.

The second suite of programs (test1) is obtained by adding to the previous one the following clause defining the predicate r: $r(X,Y)$:- $h(X,Y), \textbf{not}\, p(X,Y)$. The whole program admits a complete well-founded model and thus a single answer set. We still observe a quadratic growth in execution time, but the number of calls to T_P is now greater, and in this case Lparse+Smodels outperforms GASP. The size of the ground file with $N = 512$ is 8MB. In fact, the alternating fixpoint techniques used by GASP for computing the well-founded set reduces to five calls to the fixpoint of T_P for this program. ASPeRiX, instead, uses the fact that the program

[3]`www.tcs.hut.fi/Software/smodels/`
[4]`http://www.info.univ-angers.fr/pub/claire/asperix/`

is stratified and therefore, in practice, runs as if the fixpoint procedure is called only once. We are planning to improve our code by extending the more general rewriting technique of [25] to the case of non-ground programs.

The third suite of benchmarks (`test2`) is based on graphs obtained modifying the above example. The programs admit two different answer sets. In this case, the preliminary computation of the well-founded model returns a partial answer set, and the non deterministic part of the GASP computation procedure is used. The grounding time (and size of the program—with $N = 160$ the file is 224MB) is not negligible. In these examples, we show that GASP outperforms LPARSE+SMODELS, since the grounding of the program can be avoided by GASP. Compared to ASPeRiX, GASP behaves similarly: there is a rather constant ratio between the two execution times.

We also tested GASP on problems that define *functions*, which is a rather common situation when encoding CSPs using ASP (see, e.g., [7]). In this case, a typical encoding has the form:

```
domain(a1).      ...        domain(an).
range(b1).       ...        range(bm).
1 { assignment(X,Y):range(P) } 1 :- domain(X).
```

plus a set of ASP constraints on the `assignment` relation. The above cardinality constraint could be implemented by a naive series of rules of the form:

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

This, however, leads to poor performance of GASP w.r.t. LPARSE+SMODELS. For instance, if we encode in this way the Schur problem (see `test5`, dealt with below), for $(7, 3)$ GASP finds the first solution in 5 seconds, while LPARSE+SMODELS require 0.1 seconds.

The current GASP implementation admits an effective extension that we have exploited. We can precede the calling of the `fixpoint` procedure with a non-deterministic and constraint-based generation of the values of the functions that satisfy the ASP constraints. This approach has been completely realized in the current implementation only for certain classes of problems, but many of the notions used can be easily generalized (and this is work in progress).

In Figure 4, we report the main code relative to this part. The definition of the predicates `functions` and `funbuild` is problem-independent (whenever predicate names `domain`, `range`, and `assignments` are used). A list of pairs $D = [X, Y]$ is generated, where X takes values in `domain` and Y in `range`. All values of the domain must be chosen. The predicate `increasing` arranges these values in increasing order. We also add some problem-dependent constraints between these values, using the predicate `constraint_adhoc`, and launch a `labeling` stage that finds solutions.

The predicate `constraint_adhoc` is problem dependent, but there is a simple algorithm for translating a family of ASP constraints into recursive rules. We have used this idea in the remaining tests. For instance, `test3` implements a marriage problem where the constraint is of the form:

```
:- domain(X), range(Y), assignment(X,Y), X < Y.
```

Intuitively, this constraint indicates that, *for each* X and *for each* Y, it cannot be the case that $X < Y$ and `assignment`(X, Y). This induces a recursive predicate that, for all X and for all Y, states that $X \geq Y$. This is what we have done in lines 17–20 of the code in Figure 4.

This idea can be generalized as follows. Consider all ASP constraints dealing explicitly with the `assignment` predicate.

- For each ASP constraint C, split it into the following four parts:
  - the "domain" and "range" predicates
  - the occurrences of the predicate `assignment`
  - built-in arithmetic atoms
  - the other predicates.
- The number of occurrences of the predicate `assignment` determines the number of nested recursions needed—in other words, the number of `forall` to be implemented by the recursion.
- The conjunction of built-in and other predicates must be negated. This is done using built-in constraints (in the former case) or using constraints of non-membership to FDSETs.

Some other experiments are performed according to this design. In the encoding of the N-Queens problem (`test4`), two ASP constraints have to be translated, one for horizontal attack and one for diagonal attack. Each of them requires a double recursion. In the encoding of the Schur numbers (`test5`), a triple recursion is needed to state that if `assignment`$(X1, Y)$ and `assignment`$(X2, Y)$, then it cannot be that `assignment`$(X1 + X2, Y)$.

```
(1)    functions(WFModel, [atom(assignment,2,ASS,NDOM)|RModel]) :-
(2)       member(atom(domain,1,NUM,_),WFModel),
(3)       member(atom(range,1,RAN,_),WFModel),
(4)       select(atom(assignment,2,_,NDOM),WFModel,RModel),
(5)       fdset_size(NUM,L),
(6)       length(Xs,L), length(Ys,L),
(7)       funbuild(Xs,Ys,NUM,RAN,FUN),
(8)       increasing(Xs),
(9)       constraint_adhoc(RModel,FUN),
(10)      labeling([],FUN),
(11)      list_to_fdset(FUN,ASS).
(12)   funbuild([],[],_,_,[]).
(13)   funbuild([X|Xs],[Y|Ys],DOM,RAN,[D|Fun]) :-
(14)      X in_set DOM, Y in_set RAN,
(15)      tuple_num([X,Y],D,2),
(16)      funbuild(Xs,Ys,DOM,RAN,Fun).

(17)   constraint_adhoc(_,[]).
(18)   constraint_adhoc(_,[PAIR|FUN]) :-
(19)      pair_proj(PAIR,X,Y), %% Similar to num_tuple
(20)      X #>= Y,
(21)      constraint_adhoc(_,FUN).

(22)   constraint_adhoc(Model,FUN) :-
(23)      member(atom(hate,2,EDGES,_),Model),
(24)      constraint_marriage_rec(FUN,EDGES).
(25)   constraint_marriage_rec([],_).
(26)   constraint_marriage_rec([PAIR|FUN],EDGES) :-
(27)      nin_set(2,PAIR,EDGES),
(28)      constraint_marriage_rec(FUN,EDGES).
```

Figure 4: Constraint based handling of functions

In test6, we have encoded a marriage problem with an auxiliary predicate hate, just to describe the extension of the translation method when non built-in predicates are used together with `assignment` in a constraint. In this case, the ASP constraint

```
:- domain(X), range(Y), hate(X,Y), assignment(X,Y).
```

is translated into the predicate defined in lines 22–28 of Figure 4.

As reported in Table 2, the performance of GASP is promising in the case of execution of ASP programs encoding CSPs—in consideration of the proposed extensions of the basic implementation. ASPeRiX runs faster w.r.t. GASP, mostly due to its implementation in C (while GASP is completely written in Prolog). Moreover, the technique used for computing the well-founded model in GASP can be enhanced (see notes to test1). But for tests 3–6 ASPeRiX is not applicable since it does not deal with aggregates.

test7 points out the current limits of the lazy grounding approach. The problem tested is the same as for test5, but the naive definition of function has been employed. While the Lparse+Smodels system solves it in less than one second, GASP (and ASPeRiX) are thousands of times slower. The grounding of these programs are rather small (175 rules for 9 numbers and 3 partitions), and the search for solutions require exploring a very bushy search space. GASP and ASPeRiX repeat the local grounding a large number of times. The number of choice points used by GASP during the computation is 1/20 of those constructed by ASPeRiX.

The test test8 is called p2.nlp in [12] and used by authors to prove the effectiveness of ASPeRiX in a case with large grounding when one is interested in a single solution. Running time of GASP show the same trend of ASPeRiX, biased by a constant factor mainly due to different languages overhead. Moreover, if we look for all the solutions, ASPeRiX builds a number of choice points of double exponential size. GASP instead has no choice points for the problem and the time for closing the search tree is basically the same as for computing the first solution.

The test9 is an encoding of a variant of the csplib problem number 9 (perfect square placement problem). The problem requests to pack a set of squares with given integer sizes into a bigger square in such a way that no squares overlap each other and all square borders are parallel to the border of the big square. An instance n means that there are 6 squares to be placed into a square of size n. There is one square of size $n/3$ and 5 squares of size $2n/3$. The size of the grounded program is huge: in the first case it is 71MB, in the second 945MB, and in the third Lparse did not generate the file due to its size. Moreover, in the second case Smodels was not able to open the ground file and thus no computation was possible. It is interesting to note that solving CSPs using functions is able to reduce drastically the search space. In this example ASPeRiX is outperformed, despite of the Prolog computational time overhead. In the future work we plan to generalize this idea and to provide a program independent application of functions during GASP computation.

	N (n,p)	Lparse	Smodels	GASP	time GASP/ Lparse+Smodels	Asperix
test0	64	0.07	0.01	0.18	2.3x	0.01
(all sol)	128	0.29	0.03	0.58	1.8x	0.04
	256	1.28	0.16	2.57	1.8x	1.8
	512	5.64	0.59	14.25	2.3x	1.9
test1	64	0.11	0.02	1.31	10.1x	0.01
(all sol)	128	0.40	0.08	4.97	10.3x	0.05
	256	1.85	0.34	23.85	10.9x	0.33
	512	8.27	1.25	142.21	14.9x	2.14
test2	20	0.37	0.06	0.32	0.73x	0.02
(all sol)	40	2.72	0.48	1.26	0.39x	0.10
	80	20.67	4.20	7.19	0.28x	0.68
	160	164.2	38.45	70.51	0.35x	5.28
test3	100	0.26	0.62	0.30	0.34x	-
(1st sol)	200	1.05	8.75	0.63	0.06x	-
	400	4.11	137.2	1.25	1/114x	-
	800	16.33	2,195	1.91	1/1,157x	-
test4	10	0.04	0.01	0.01	0.20x	-
(1st sol)	15	0.14	0.15	0.01	0.03x	-
	20	0.34	10.95	45.1	3.99x	-
	25	0.69	2,339	18.6	1/125x	-
test5	(41,4)	0.09	110.0	2.11	1/52x	-
(1st sol)	(42,4)	0.09	165.8	2.12	1/77x	-
	(43,4)	0.11	226.8	2.37	1/95x	-
	(44,4)	0.11	2,986	3,201	1.07x	-
test6	100	0.47	0.15	0.73	1.17x	-
(1st sol)	200	2.06	0.54	2.27	0.87x	-
	400	9.89	2.13	8.32	0.69x	-
	800	64.4	8.39	24.97	0.34x	-
test7	(8,3)	0.01	0.01	141.5	7,050x	1.92
(all sol)	(9,3)	0.01	0.01	455.5	22,775x	7.26
	(10,3)	0.01	0.01	1,075	54,750x	23.3
test8	10	0.01	0.01	0.04	2x	0.01
(1st sol)	20	0.03	0.01	0.13	3.25x	0.01
	40	0.11	0.03	0.27	1.9x	0.02
	80	0.42	0.14	0.78	1.4x	0.07
	160	1.71	0.65	2.87	1.21x	0.37
test9	24	66.2	258.1	0.17	1/1,907x	0.27
(1st sol)	45	831	x	0.22	0x	1.29
	75	∞	x	0.29	0x	5.69

Table 2: Timings (expressed in seconds) '-' means that ASPeRiX is not applicable to function definitions.

6 Conclusions

In this paper we provided the foundation for a bottom-up construction of answer sets of a program P without preliminary program grounding. The notion of 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 Constraint Logic Programming and constraints on FD variables and FDSETs. We showed how to design T_P, well-founded and answer sets computation based on CSPs. This allowed us to encode the entire process in Prolog. Interestingly, the performance of the implementation of T_P and the well-founded computation are competitive with Smodels. Some ASP programs run slower, due to the inherent Prolog overheads and the limited efficiency of some (naive) data structures used.

As future work, we plan to investigate how to extend the design to enable the integration of other language features commonly encountered in ASP languages, including constraints as introduced in [18], 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 and the use of low level data structures that allow faster access to rules and models.

Acknowledgments

The work is partially supported by MUR FIRB RBNE03B8KK and PRIN projects, and NSF grants HRD0420407, IIS-0812267, and CNS0220590. We thank Andrea Formisano for the several useful discussions, and the reviewers of CILC08, LaSh08, and Fundamenta Informaticae for their useful comments.

References

[1] Babovich, Y., Maratea, M.: Cmodels-2: SAT-based Answer Sets Solver Enhanced to Non-tight Programs, *Logic Programming and Non-Monotonic Reasoning*, LNCS 2923, 2004, 346–350.

[2] Baral, C.: *Knowledge Representation, Reasoning, and Declarative Problem Solving*, Cambridge University Press, 2003.

[3] Bonatti, P., Pontelli, E., Son, T. Credulous Resolution for ASP, *AAAI*, 2008, 418-423.

[4] Brooks, D., Erdem, E., Erdogan, S., Minett, J., Ringe, D.: Inferring Phylogenetic Trees Using Answer Set Programming, *Journal of Automated Reasoning*, **39**(4), 2007, 471–511.

[5] Codognet, P., Diaz, D.: A Minimal Extension of the WAM for `clp(fd)`, *International Conference on Logic Programming*, pp. 774-790, MIT Press, 1993.

[6] Dal Palù, A., Dovier, A., Pontelli, E., Rossi, G.: GASP: Answer Set Programming with Lazy Grounding, *CILC08: 23-esimo Convegno Italiano di Logica Computazionale*, Perugia, Italy, July 10–12th, 2008, and *LaSh08: International Workshop on Logic and Search*, Nov. 6–7th 2008, Leuven, Belgium.

[7] Dovier, A., Formisano, A., Pontelli, E.: A Comparison of CLP(FD) and ASP Solutions to NP-Complete Problems, *International Conference on Logic Programming*, LNCS 3668, Springer Verlag, pp. 67-82, 2005.

[8] Dovier, A., Formisano, A., Pontelli, E.: Multivalued Action Languages with Constraints in CLP(FD), *International Conference on Logic Programming*, LNCS 4670, Springer Verlag, pp. 255-270, 2007.

[9] Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: CLASP: a Conflict-driven Answer Set Solver, *Logic Programming and Non-Monotonic Reasoning*, Springer Verlag, 2007, 260–265.

[10] Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programs, *International Symposium on Logic Programming*, MIT Press, 1988, 1070–1080.

[11] Hoffmann, J., Nebel, B.: The FF Planning System: Fast Plan Generation Through Heuristic Search, *Journal of Artificial Intelligence Research*, **14**, 2001, 253–302.

[12] Lefèvre, C., Nicolas, P.: Integrating Grounding in the Search Process for Answer Set Computing, *Workshop on Answer Set Programming and Other Computing Paradigms*, Udine, Italy, Dec. 13, 2008.

[13] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning, *ACM Transactions on Computational Logic*, **7**(3), 2006, 499–562.

[14] Lifschitz, V.: Answer Set Planning. *Logic Programming and Non-monotonic Reasoning*, LNCS 1730, Springer Verlag, 1999, 373–374.

[15] Lin, F., Zhao, Y.: ASSAT: Computing answer sets of a logic program by SAT solvers, *Artificial Intelligence*, **157**(1–2), 2004, 115–137.

[16] Liu, L., Pontelli, E., Tran, S., Truszczyński, M.: Logic Programs with Abstract Constraint Atoms: the Role of Computations, *International Conference on Logic Programming*, Springer Verlag, 2007, 286–301.

[17] Marek, V. W., Truszczyński, M.: Stable Models and an Alternative Logic Programming Paradigm, *The Logic Programming Paradigm*, Springer Verlag, 1999.

[18] Mellarkod, V. S., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. *Ann. Math. Artif. Intell. (AMAI)* 53(1–4):251–287, 2008.

[19] Niemela, I.: Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm, *Annals of Mathematics and AI*, **25**(3–4), 1999, 241–273.

[20] Niemela, I., Simons, P.: Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal LP, *Logic Programming and Non-monotonic Reasoning*, Springer Verlag, 1997, 421–430.

[21] P. Simons. Extending and Implementing the Stable Model Semantics. Ph.D. Thesis, Helsinki University of Technology, 2000.

[22] Son, T., Pontelli,E.: Set Constraints in Logic Programming, *Logic Programming and Non-Monotonic Reasoning*, LNCS 2923, Springer Verlag, 2004, 167–179.

[23] Son, T., Pontelli, E.: Planning for Biochemical Pathways: a Case Study of Answer Set Planning in Large Planning Problem Instances, *First International Workshop on Software Engineering for Answer Set Programming*, 2007, 116–130.

[24] Van Gelder, A., Ross, K. A., Schlipf, J. S.: The Well-Founded Semantics for General Logic Programs, *Journal of the ACM*, **38**(3), 1991, 620–650.

[25] Zukowski, U., Freitag, B., Brass, S.: Improving the Alternating Fixpoint: The Transformation Approach, *Logic Programming and Nonmonotonic Reasoning*, LNCS 1265, Springer-Verlag, 1997, 4–59.