An Efficient Solver for ASP(Q)*

WOLFGANG FABER

Alpen-Adria Universität Klagenfurt, Austria
(*e-mail:* Wolfgang.Faber@aau.at)

GIUSEPPE MAZZOTTA and FRANCESCO RICCA

University of Calabria, Rende, Italy
(*e-mails:* giuseppe.mazzotta@unical.it, francesco.ricca@unical.it)

Abstract

Answer Set Programming with Quantifiers ASP(Q) extends Answer Set Programming (ASP) to
allow for declarative and modular modeling of problems from the entire polynomial hierarchy.
The first implementation of ASP(Q), called QASP, was based on a translation to Quantified
Boolean Formulae (QBF) with the aim of exploiting the well-developed and mature QBF-solving
technology. However, the implementation of the QBF encoding employed in qasp is very general
and might produce formulas that are hard to evaluate for existing QBF solvers because of the
large number of symbols and subclauses. In this paper, we present a new implementation that
builds on the ideas of QASP and features both a more efficient encoding procedure and new optimized encodings of ASP(Q) programs in QBF. The new encodings produce smaller formulas (in
terms of the number of quantifiers, variables, and clauses) and result in a more efficient evaluation
process. An algorithm selection strategy automatically combines several QBF-solving back-ends
to further increase performance. An experimental analysis, conducted on known benchmarks,
shows that the new system outperforms QASP.

KEYWORDS: ASP with Quantifiers, Quantified Boolean Formulas, well-founded semantics

1 Introduction

Answer Set Programming (ASP) (Brewka *et al.* 2011; Gelfond and Lifschitz 1991) is a
popular logic programming paradigm based on the stable models semantics, offering the
capabilities to (*i*) modeling search and optimization problems in a declarative (and often
compact) way and (*ii*) solving them using efficient systems (Gebser *et al.* 2018) that can
handle real-world problems (Erdem *et al.* 2016; Gebser *et al.* 2017).

Despite being very effective in modeling and solving problems in NP (Gebser *et al.*
2017), the first level of the polynomial hierarchy (PH), ASP is less practical when one
has to approach problems beyond NP. The existing programming techniques, such as
saturation (Eiter and Gottlob 1995; Dantsin *et al.* 2001), which allow for encoding with

ASP problems that belong to the second level of the PH, are not very intuitive. Moreover, the expressive power of ASP does not span the entire PH.

Recently, these shortcomings of ASP have been overcome by the introduction of language extensions that expand the expressivity of ASP (Bogaerts *et al.* 2016; Fandinno *et al.* 2021; Amendola *et al.* 2019). Among these, Answer Set Programming with Quantifiers ASP(Q) extends ASP, allowing for declarative and modular modeling of problems of the entire PH (Amendola *et al.* 2019). The language of ASP(Q) expands ASP with quantifiers over answer sets of ASP programs and allows the programmer to use the standard and natural programming methodology, known as generate-define-test (Lifschitz 2002), to encode also problems beyond NP.

Motivation. As in the case of ASP, the adoption of ASP(Q) as a tool for modeling concrete problems (Faber *et al.* 2022; Faber and Morak 2022) has begun after the introduction of the first solver for ASP(Q), called QASP (Amendola *et al.* 2022). QASP was based on a translation to Quantified Boolean Formulae (QBF) with the aim of exploiting the well-developed and mature QBF-solving technology (Pulina and Seidl 2019). However, the implementation of the QBF encoding employed in QASP is very general and might produce formulas that are hard to evaluate for existing QBF solvers because of the large number of symbols and subclauses. Moreover, Amendola *et al.* (2022) observed that the implementation of the translation procedure could – in some specific cases – be so memory-hungry to prevent the production of the QBF formula even when a considerable amount of memory is available. Moreover, QASP's performance is dependent on the choice of the back-end QBF solver that performs differently over different domains. This means that *the quest for techniques resulting in faster solvers for ASP(Q) is still open and challenging* and directly impacts the deployment of ASP(Q) applications.

Contributions. In this paper, we present a new implementation of ASP(Q) that builds on the ideas of QASP but features both a more efficient encoding procedure and new optimized encodings of ASP(Q) programs in QBF. More specifically, we provide

1. An approach that exploits the well-founded semantics (Van Gelder *et al.* 1991) for simplifying ASP(Q) programs.
2. The identification of a natural syntactic fragment of ASP(Q) programs that can be directly translated to a QBF in Conjunctive Normal Form (CNF), thereby avoiding costly normalization steps.
3. A new system for ASP(Q) implemented in Python, called PYQASP, that is modular and features an automatic selection of a suitable back-end for the given input.

The new translations in QBF can produce smaller (or equally large) formulas, in terms of the number of quantifiers, variables, and clauses, with respect to the ones employed in QASP, and this results in a more efficient evaluation process. A porting to the ASP(Q) setting of the algorithm selection methodology employed by the competition-winning solver, ME-ASP (Maratea *et al.* 2014), allows PYQASP to deliver steady performance over different problem domains. An experimental analysis shows when the new optimizations provide benefits and demonstrates empirically that PYQASP outperforms QASP and compares favorably with the implementation of stable–unstable semantics by Janhunen (2022).

2 Preliminaries

In this section, we provide preliminary notions concerning logic programs, answer sets, well-founded semantics, and ASP(Q). For ease of presentation, we focus our attention on propositional logic programs, but our methods are applicable to the full language of ASP.

2.1 Programs

An atom is a propositional variable, and a literal is either an atom a or its negation $\sim a$ where $\sim$ represents negation as failure. A literal l is positive (resp. negative) if it is of the form a (resp. $\sim a$). The complement of a literal l, $\bar{l}$, is $\sim a$ if $l = a$ or a if $l =\sim a$. Given a set of literals L, $\neg L$ denotes the set of literals $\{\bar{l} \mid l \in L\}$. A choice atom is an expression of the form $\{a_1; \cdots ; a_m\}$ where $m \geq 0$ and $a_1, \ldots, a_m$ are atoms. A rule is an expression of the form $h \leftarrow l_1, \ldots, l_n$ where $n \geq 0$, h is either an atom or a choice atom, referred to as rule head and denoted by H_r, and $l_1, \ldots, l_n$ is a conjunction of literals, referred to as body and denoted by B_r. A rule is a fact if it has an empty body; it is normal if h is an atom; it is a choice rule if h is a choice atom. A constraint is a rule with an empty head $\leftarrow l_1, \ldots l_n$, which is a shorthand for $x \leftarrow l_1, \ldots, l_n, \sim x$ with x being a fresh atom not occurring anywhere else. A choice rule $\{a_1; \cdots ; a_m\} \leftarrow$ is (for simplicity) a shorthand for the rules $a_i \leftarrow \sim na_i$, $na_i \leftarrow \sim a_i$ for $1 \leq i \leq m$, where all na_i are fresh atoms not appearing elsewhere. A program is a finite set of normal rules.

Given a program P, the dependency graph, G_P, is a directed labeled graph where nodes are atoms in P and there is a positive (resp. negative) arc (a_1, a_2) if there exists a rule $r \in P$ such that a_1 (resp. $\sim a_1$) appears in the body of r and a_2 is the head of r. P is stratified if G_P contains no cycles involving negative arcs.

Stable Models Semantics. Given a program P, the Herbrand Base, $\mathcal{B}_P$, is the set of atoms occurring in P. A (partial) interpretation I is a subset of $\mathcal{B}_P \cup \neg \mathcal{B}_P$. A literal l is true (resp. false) w.r.t. I if $l \in I$ (resp. $\bar{l} \in I$), otherwise it is undefined. A conjunction of literals is true w.r.t. I if all literals are true. An interpretation I is consistent if for each $l \in I$, $\bar{l} \notin I$; it is total if for each $a \in \mathcal{B}_P$, a is either true or false w.r.t. I. A rule r is falsified w.r.t. I if B_r is true and H_r is false. A rule r is satisfied w.r.t. I if B_r is false or H_r is true. A consistent interpretation I is a model of P if it does not falsify any rule in P.

A total model M is a (subset-)minimal model if does not exist a total model M_1 such that $M_1^+ \subset M^+$. Given a model M, the (Gelfond–Lifschitz) reduct of P w.r.t. M, P^M, is obtained from P by removing rules with negative literals in the body that are false w.r.t. M and deleting from the body of the remaining rules all negative literals that are true w.r.t. M. A total model M is an answer set of P if M is a minimal model of P^M (Gelfond and Lifschitz 1991).

$AS(P)$ is the set of answer sets of P. P is *coherent* iff $AS(P) \neq \emptyset$.

Well-founded Semantics. Let P be a program and I be an interpretation, a set of atoms $U \subseteq \mathcal{B}_P$ is an unfounded set of P w.r.t. I if for each rule $r \in P$ such that $H_r \in U$, B_r is false w.r.t. I or $B_r \cap U \neq \emptyset$. The greatest unfounded set of P w.r.t. I, $U_P(I)$, is defined as the union of all unfounded sets of P w.r.t. I. Let $T_P(I)$ be the set of atoms $a \in \mathcal{B}_P$ such that there exists a rule $r \in P$ having a in the head and a true body w.r.t. I, the

well-founded operator, $\mathcal{W}_P(I)$, is defined as $T_P(I) \cup \neg U_P(I)$. The (partial) well-founded model is defined as the least fixed point of the operator $\mathcal{W}_P$ (Van Gelder *et al.* 1991). The well-founded model W of P is a subset of each answer set of P.

2.2 Answer set programming with quantifiers

An *ASP with Quantifiers* (ASP(Q)) program Π is of the form (Amendola *et al.* 2019):

$$\Box_1 P_1 \, \Box_2 P_2 \, \cdots \, \Box_n P_n : C, \tag{1}$$

where, for each $i = 1, \ldots, n$, $\Box_i \in \{\exists^{st}, \forall^{st}\}$, P_i is an ASP program, and C is a stratified ASP program possibly with constraints. An ASP(Q) program Π of the form (1) is *existential* if $\Box_1 = \exists^{st}$, otherwise it is *universal*.

Given a logic program P, a total interpretation I over the Herbrand base $\mathcal{B}_P$, and an ASP(Q) program Π of the form (1), we denote by $\textit{fix}_P(I)$ the set of facts and constraints $\{a \mid a \in I \cap \mathcal{B}_P\} \cup \{\leftarrow a \mid a \in \mathcal{B}_P \setminus I\}$, and by $\Pi_{P,I}$ the ASP(Q) program of the form (1), where P_1 is replaced by $P_1 \cup \textit{fix}_P(I)$, that is, $\Pi_{P,I} = \Box_1(P_1 \cup \textit{fix}_P(I)) \, \Box_2 P_2 \, \cdots \Box_n P_n : C$.

The *coherence* of ASP(Q) programs is defined by induction as follows:

- $\exists^{st} P : C$ is coherent, if there exists $M \in AS(P)$ such that $C \cup \textit{fix}_P(M)$ is coherent;
- $\forall^{st} P : C$ is coherent, if for every $M \in AS(P)$, $C \cup \textit{fix}_P(M)$ is coherent;
- $\exists^{st} P \, \Pi$ is coherent, if there exists $M \in AS(P)$ such that $\Pi_{P,M}$ is coherent;
- $\forall^{st} P \, \Pi$ is coherent, if for every $M \in AS(P)$, $\Pi_{P,M}$ is coherent.

"Unwinding" the definition for a quantified program $\Pi = \exists^{st} P_1 \forall^{st} P_2 \cdots \exists^{st} P_{n-1} \forall^{st} P_n : C$ yields that Π is coherent if there exists an answer set M_1 of P_1' such that for each answer set M_2 of P_2' there is an answer set M_3 of $P_3', \ldots$, there is an answer set M_{n-1} of P_{n-1}' such that for each answer set M_n of P_n', there is an answer set of $C \cup \textit{fix}_{P_n'}(M_n)$, where $P_1' = P_1$, and $P_i' = P_i \cup \textit{fix}_{P_{i-1}'}(M_{i-1})$, if $i \geq 2$. For an existential ASP(Q) program Π, $M \in AS(P_1)$ is a *quantified answer set* of Π, if $(\Box_2 P_2 \cdots \Box_n P_n : C)_{P_1,M}$ is coherent. We denote by $QAS(\Pi)$ the set of all quantified answer sets of Π.

Given a set of propositional atoms A, we denote by $ch(A)$ the program $\{\{a\} \mid a \in A\}$ made of choice rules over atoms in A. For two ASP programs P and P', let $Int(P, P')$ be the set $\mathcal{B}_P \cap \mathcal{B}_{P'}$ of common atoms. For two programs P and P', the choice interface program $CH(P, P')$ is defined as $ch(Int(P, P'))$. For a propositional formula Φ, $var(\Phi)$ denotes the variables occurring in Φ. For an ASP(Q) program Π, and an integer $1 \leq i \leq n$, we define the program $P_i^{\leq}$ as the union of program P_j with $1 \leq j \leq i$. Given an input program Π of the form (1), the intermediate versions G_i of its subprograms, and the QBF $\Phi(\Pi)$ encoding Π are as follows:

$$G_i = \begin{cases} P_1 & i = 1 \\ P_i \cup CH(P_{i-1}^{\leq}, P_i) & 1 < i \leq n, \\ C \cup CH(P_n^{\leq}, C) & i = n+1 \end{cases}$$

$$\Phi(\Pi) = \boxplus_1 \cdots \boxplus_{n+1} \left(\bigwedge_{i=1}^{n+1} (\phi_i \leftrightarrow CNF(G_i)) \right) \wedge \phi_c,$$

where $CNF(P)$ is a CNF formula encoding the program P (such that models of $CNF(P)$ correspond to $AS(P)$); $\phi_1, \ldots, \phi_{n+1}$ are fresh propositional variables; $\boxplus_i = \exists x_i$ if $\Box_i = \exists^{st}$

or $i = n+1$, and $\boxplus_i = \forall x_i$ otherwise, where $x_i = var(\phi_i \leftrightarrow CNF(G_i))$ for $i = 1, \cdots, n+1$, and ϕ_c is the formula:

$$\phi_c = \phi_1' \odot_1 (\phi_2' \odot_2 (\cdots \phi_n' \odot_n (\phi_{n+1}) \cdots)),$$

where $\odot_i = \vee$ if $\square_i = \forall^{st}$, and $\odot_i = \wedge$ otherwise, and $\phi_i' = \neg\phi_i$ if $\square_i = \forall^{st}$, and $\phi_i' = \phi_i$ otherwise. Intuitively, there is a direct correspondence between the quantifiers in Π and $\Phi(\Pi)$; moreover, in each subprogram of Π (i.e, $P_1, \cdots, P_n, C$) the atoms interfacing with preceding subprograms are left open; then the programs are converted into equivalent CNF formulas; finally, the formula ϕ_c is built to constrain the variable assignments corresponding to the stable models of each subprogram so that they behave as required by the semantics of ASP(Q).

Theorem 1 (*Amendola et al. 2022*)
Let Π be a quantified program. Then $\Phi(\Pi)$ is true iff Π is coherent.

3 Simplification based on well-founded semantics

We present an alternative approach that exploits the well-founded semantics in order to obtain a simplified but equivalent ASP(Q) program that allows a more compact translation into a QBF formula both in terms of number of clauses and average clause length.

Definition 1
Given a program P and its well-founded model $\mathcal{W}$, the *residual program*, $R(P)$, is obtained from P by removing all those rules with a false body w.r.t. $\mathcal{W}$ and true literals in $\mathcal{W}$ from the bodies of the remaining ones.

Proposition 1
Given a program P and its well-founded model $\mathcal{W}$, $AS(P) = AS(R(P))$

Note that $\mathcal{W}$ is a subset of any stable model M of P and so the missing rules in $R(P)$ have a false body w.r.t. M and so they are trivially satisfied by M. Each rule in P that has not been removed in $R(P)$ is satisfied if and only if the simplified rule in $R(P)$ is satisfied. So, it can be proved that the reduct of the two programs have the same minimal models for every model M. Let P and P' be two programs and $\mathcal{W}$ the well-founded model of P, $CH'(P, P') = \{\{a\} \mid a \in (Int(P, P') \setminus (\mathcal{W} \cup \neg\mathcal{W}))\} \cup \{a \leftarrow \mid a \in (Int(P, P') \cap \mathcal{W})\}$. Given an ASP(Q) program Π of the form (1), the QBF encoding $\Phi^{WF}(\Pi)$ is as follows:

$$G_i^{WF} = \begin{cases} R(P_1) & i = 1 \\ R(P_i \cup CH'(P_{i-1}^{\leq}, P_i)) & i \in [2..n], \\ R(C \cup CH'(P_n^{\leq}, C)) & i = n+1 \end{cases}$$

$$\Phi^{\mathcal{WF}}(\Pi) = \boxplus_1 \cdots \boxplus_{n+1} \left(\bigwedge_{i=1}^{n+1} (\phi_i^{\mathcal{WF}} \leftrightarrow CNF(G_i^{WF})) \right) \wedge \phi_c,$$

where $CNF(G_i^{WF})$ is a CNF formula encoding G_i^{WF}, $\phi_1^{\mathcal{WF}}, \ldots, \phi_{n+1}^{\mathcal{WF}}$ are fresh propositional variables; $\boxplus_i = \exists x_i$ if $\square_i = \exists^{st}$ or $i = n+1$, and $\boxplus_i = \forall x_i$ otherwise, where $x_i = var(\phi_i^{\mathcal{WF}} \leftrightarrow CNF(G_i^{WF}))$ for $i = 1, \cdots, n+1$, and ϕ_c is the formula:

$$\phi_c = \phi'_1 \odot_1 (\phi'_2 \odot_2 (\cdots \phi'_n \odot_n (\phi_{n+1}) \cdots)),$$

where $\odot_i = \vee$ if $\Box_i = \forall^{st}$, and $\odot_i = \wedge$ otherwise, and $\phi'_i = \neg \phi_i^{\mathcal{WF}}$ if $\Box_i = \forall^{st}$, and $\phi'_i = \phi_i^{\mathcal{WF}}$ otherwise. Intuitively, $\Phi^{\mathcal{WF}}(\Pi)$ is constructed by following the encoding proposed in Section 2.2, but each program P_i is replaced by its residual w.r.t. the well-founded model.

Theorem 2
Let Π be an ASP(Q) program, then $\Phi^{\mathcal{WF}}(\Pi)$ is true iff Π is coherent.

Proof
By means of CH' possible models of P_i are reduced to those that are coherent with models from previous levels by fixing the truth value of literals that have been determined by the well-founded operator and so $AS(G_i^{WF})) \subseteq AS(G_i)$. If there exists $M \in AS(G_i)$ such that $M \notin AS(G_i^{WF})$, then there exists some literal $l \in M$ such that $\sim l$ belongs to the well-founded model of P_{i-1} and so M is not coherent with models of previous levels. So, the program $P'_i = P_i \cup CH'(G_{i-1}^{WF}, P_i)$ preserves the coherence of Π. From Proposition 1, we know that $AS(P'_i) = AS(R(P'_i))$ and so from Theorem 1 we can conclude that $\Phi^{\mathcal{WF}}(\Pi)$ is true iff Π is coherent. $\square$

Proposition 2
Let Π be an ASP(Q) program of the form (1), if G_i^{WF} is incoherent then $\phi_i^{\mathcal{WF}}$ can be replaced by $\bot$.

It is easy to see that if G_i^{WF} is incoherent then $CNF(G_i^{WF})$ is unsatisfiable and so $\phi_i^{\mathcal{WF}}$ can be replaced by $\bot$.

Example 3.1
Let P_i be the program $\{a \leftarrow a; \quad p \leftarrow \sim a, \sim p\}$. Since predicates occurring in P_i are p and a and both are defined at level i then $CH'(P_{i-1}^{\leq}, P_i) = \emptyset$ and so, $G_i^{WF} = R(P_i)$. Since the well-founed model of P_i is $\mathcal{W} = \{\sim a\}$, then $R(P_i) = p \leftarrow \sim p$ that is incoherent, and so $CNF(G_i^{WF}) = p \wedge \neg p$ is unsatisfiable.

Proposition 3
Let Π be an ASP(Q) program of the form (1), if G_k^{WF} is incoherent then

$$\Phi^{\mathcal{WF}}(\Pi) \equiv \boxplus_1 \cdots \boxplus_{k-1} \left(\bigwedge_{i=1}^{k-1} (\phi_i^{\mathcal{WF}} \leftrightarrow CNF(G_i^{WF})) \right) \wedge \phi'_c,$$

where $\phi_1^{\mathcal{WF}}, \ldots, \phi_{k-1}^{\mathcal{WF}}$ are fresh propositional variables; $\boxplus_i = \exists x_i$ if $\Box_i = \exists^{st}$, and $\boxplus_i = \forall x_i$ otherwise, $x_i = var(\phi_i \leftrightarrow CNF(G_i^{WF}))$ for $i = 1, \cdots, k-1$, and ϕ'_c is the formula:

$$\phi'_c = \phi'_1 \odot_1 (\phi'_2 \odot_2 (\cdots \phi'_{k-1} \odot_{k-1} (\phi'_k) \cdots)),$$

where $\odot_i = \vee$ if $\Box_i = \forall^{st}$, and $\odot_i = \wedge$ otherwise, for $i \in [1, \cdots, k-1]$, $\phi'_i = \neg \phi_i^{\mathcal{WF}}$ if $\Box_i = \forall^{st}$, and $\phi'_i = \phi_i^{\mathcal{WF}}$ otherwise, and $\phi'_k = \top$ if $\Box_i = \forall^{st}$, and $\phi'_k = \bot$ otherwise.

From Proposition 3 it follows that if $k = 1$ then $\Phi^{\mathcal{WF}}(\Pi) = \phi'_k$ where $\phi'_k = \top$ if $\Box_i = \forall^{st}$, and $\phi'_k = \bot$ otherwise. So, in such cases, we can determine the coherence of the ASP(Q) directly in the encoding phase.

Proposition 4
Given an ASP(Q) program Π, it holds that $|clauses(\Phi^{\mathcal{WF}}(\Pi))| \leq |clauses(\Phi(\Pi))|$

We observe that, by definition 1, residual subprograms are obtained by removing some trivially satisfied rules in every stable model or deleting literals from the rules' body by means of the well-founded operator. This results in a smaller CNF both in terms of the number of clauses, since potentially fewer rules are encoded, and also in average clause length, since each rule is transformed into one or more clauses that have fewer literals. Moreover, by propagating information from the well-founded model of previous levels, stable models of the following levels are restricted to those that are coherent with previous models, if any. If no models exist, then the resulting QBF formula is pruned at the incoherent level. In the worst case scenario, that is $\mathcal{W} = \emptyset$ for every program, CH' produces the same interface program produced by CH, $G_i = G_i^{WF}$ and so $\Phi^{\mathcal{WF}}(\Pi) = \Phi(\Pi)$.

4 Direct CNF encodings for ASP(Q) programs

Formulas $\Phi(\Pi)$ and $\Phi^{\mathcal{WF}}(\Pi)$ are not in CNF because of the presence of equivalences for each i and the final formula ϕ_c (which is not in CNF either). While this might be seen as a minor issue, the translation of non-CNF formulas into CNF by means of a Tseytin transformation can be a time-consuming procedure that increases the length of the formulas and introduces extra symbols that could slow down QBF solvers.

A natural question, therefore, is whether it is possible to identify classes of ASP(Q) programs such that the resulting QBF formula is in CNF. In the following, we can answer this positively and provide some conditions under which this is possible.

Given a program P, $heads(P)$ denotes the set of atoms that appear in the head of some rules in P, $facts(P)$ denotes the set of facts in P. Given an ASP(Q) program Π, $Ext_i = heads(P_i) \cap \bigcup_{j>i} Int(P_i, P_j)$ denotes the set of atoms defined in P_i that belong to the interface of the following levels.

Definition 2
Let Π be an ASP(Q) program, a subprogram P_i is *trivial* if the following conditions hold: (i) $\forall \; 1 \leq j < i : Int(P_i, P_j) \subseteq facts(P_j)$ and (ii) $AS(P_i) \mid_{Ext_i} = 2^{Ext_i}$, where $AS(P_i) \mid_{Ext_i} = \{S \cap Ext_i \mid S \in AS(P_i)\}$ and 2^{Ext_i} denotes the power set of Ext_i.

Let Π be an ASP(Q) program, $K = \{k \mid P_k \text{ is a trivial subprogram} \wedge k \leq n\}$, the QBF encoding $\Phi^K(\Pi)$ is defined as follows:

$$\Phi^K(\Pi) = \boxplus_1 \cdots \boxplus_{n+1} \left(\bigwedge_{\substack{i=1 \\ i \notin K}}^{n+1} (\phi_i \leftrightarrow CNF(G_i)) \right) \wedge \phi_c^K,$$

$\boxplus_i = \exists x_i$ if $\square_i = \exists^{st}$ or $i = n + 1$, and $\boxplus_i = \forall x_i$ otherwise, $x_i = var(\phi_i \leftrightarrow CNF(G_i))$ if $i \notin K$, otherwise $x_i = Ext_i$, and ϕ_c^K is $\phi_c^K = \phi_{i_1}' \odot_{i_1} (\phi_{i_2}' \odot_{i_2} (\cdots (\phi_{i_m}' \odot_{i_m} (\phi_{n+1})) \cdots))$ where $E = \{1, \cdots, n\} \setminus K = \{i_1, i_2, \cdots, i_m\}$, $i_1 < i_2 < \cdots < i_m$, $\odot_i = \vee$ if $\square_i = \forall^{st}$, and $\odot_i = \wedge$ otherwise, and $\phi_i' = \neg\phi_i$ if $\square_i = \forall^{st}$, and $\phi_i' = \phi_i$ otherwise, with $i \in E$.

Theorem 3
Let Π be an ASP(Q) program, and $K = \{k \mid P_k$ *is a trivial subprogram* $\wedge\, k \leq n\}$, then $\Phi^K(\Pi)$ is satisfiable iff Π is coherent.

Proof
From Theorem 1 we know that Π is coherent iff $\Phi(\Pi)$ is satisfiable. Now we observe that, by hypothesis, for all $k \in K$ it holds that P_k is trivial, thus $AS(P_k)\mid_{Ext_k} = 2^{Ext_k}$. This implies that for all $k \in K$, $CNF(G_k)$ is satisfiable, and thus $\phi_k \leftrightarrow \top$ holds in $\Phi(\Pi)$. Since $\phi_k \leftrightarrow CNF(G_k) \leftrightarrow \top$ holds, then the k-th conjunct is satisfiable and can be omitted obtaining an equivalent formula:

$$\Phi(\Pi) = \boxplus_1 \cdots \boxplus_n \left(\bigwedge_{\substack{i=1 \\ i \notin K}}^{n+1} (\phi_i \leftrightarrow CNF(G_i)) \right) \wedge \phi_c,$$

$\boxplus_i = \exists x_i$ if $\square_i = \exists^{st}$, and $\boxplus_i = \forall x_i$ otherwise, $x_i = var(\phi_i \leftrightarrow CNF(G_i))$ if $i \notin K$, otherwise $x_i = Ext_i$. Moreover, we observe that for each $k \in K$, $\phi_k \wedge (\phi_{k+1} \odot_{k+1} \cdots)$, can be replaced by $\top \wedge (\phi_{k+1} \odot_{k+1} \cdots)$ which is equivalent to $\phi_{k+1} \odot_{k+1} (\cdots)$ and $\neg\phi_k \vee (\phi_{k+1} \odot_{k+1} \cdots)$, can be replaced by $\bot \vee (\phi_{k+1} \odot_{k+1} \cdots)$ which is equivalent to $(\phi_{k+1} \odot_{k+1} \cdots)$. So, ϕ_c can be simplified by removing $\phi'_k \odot_k$ for each $k \in K$, obtaining an equivalent formula that is ϕ_c^K. Thus, by construction, $\Phi(\Pi)$ is equivalent to $\Phi^K(\Pi)$ and so $\Phi^K(\Pi)$ is satisfiable iff Π is coherent. $\square$

Basically, for any trivial P_k the formula $CNF(G_k)$ is a tautology, allowing for the simplifications that result in $\Phi^K(\Pi)$.

Proposition 5
Let Π be an ASP(Q) program, $K = \{k \mid P_k$ *is a trivial subprogram* $\wedge\, k \leq n\}$. If for each subprogram P_i such that $\square_i = \forall^{st}$, it holds that $i \in K$, then Φ^K is equivalent to the CNF formula with the same quantifiers: $\Phi^K_{CNF} = \boxplus_1 \cdots \boxplus_{n+1} \bigwedge_{j \in J} CNF(G_j)$, where $J = \{1, \ldots, n+1\} \setminus K$.

Programs satisfying Proposition 5 have a direct CNF encoding. However, verifying that a program is trivial is hard since Definition 2 requires a co-NP check. There is, however, a very common syntactic class of programs for which this property is trivially satisfied: the ASP(Q) programs of the form (1) where each P_i contains only choice rules. An example is the encoding of QBF in ASP(Q) proposed by Amendola *et al.* (2019). In the following, we identify a larger class of ASP(Q) programs featuring a direct encoding in CNF, the ones that follow the well-known Guess and Check methodology (Eiter and Gottlob 1995).

Definition 3
An ASP program P is *Guess&Check* if it can be partitioned into two subprograms G_P, Guess, C_P, Check, where G_P contains only choice rules and C_P is the maximal stratified subprogram possibly with constraints of P, such that $\{H_r \mid r \in C_P\} \cap \mathcal{B}_{G_P} = \emptyset$.

Example 4.1
Let P be the program $\{r_1 : \{a; b; c\} \leftarrow,\ r_2 : d \leftarrow a,\ r_3 : d \leftarrow b,\ r_4 :\leftarrow c, d.\}$. P can be partitioned in $G_P = \{r_1\}$ and $C_P = \{r_2, r_3, r_4\}$.

Guess&Check programs feature a modularity property.

Proposition 6
Let P be a Guess&Check program then $M \in AS(P)$ iff there exists $M' \in AS(G_P)$ such that $M = M' \cup W$ and $W \in AS(C_P \cup fix_{G_P}(M'))$.

Definition 4
An ASP(Q) program Π of the form (1) is *Guess&Check* if (i) universal and existential quantifiers are alternated, and (ii) all P_i with $\square_i = \forall^{st}$ are Guess&Check subprograms.

The following definition provides a rewriting for a universal Guess&Check subprogram.

Definition 5
Given a Guess&Check program P_1, a program P_2, and a propositional atom u such that $u \notin (\mathcal{B}_{P_1} \cup \mathcal{B}_{P_2})$, we define

$$\tau(u, P_1) = \begin{cases} H_r \leftarrow B_r & r \in C_{P_1} \wedge H_r \neq \emptyset \\ u \leftarrow B_r & r \in C_{P_1} \wedge H_r = \emptyset \end{cases},$$

$$\sigma(u, P_1, P_2) = \tau(u, P_1) \cup \rho(u, P_2),$$

$$\rho(u, P_2) = \{H_r \leftarrow B_r, \sim u \mid r \in P_2\}.$$

Given a *Guess&Check* ASP(Q) program Π, let $i \in [1, \dots, n]$ be such that $\square_i = \forall^{st}$:

$$\Pi^{GC_i} = \begin{cases} \square_1 P_1 \cdots \forall^{st} G_{P_i} : \sigma(u, P_i, C) & i = n \\ \square_1 P_1 \cdots \forall^{st} G_{P_i} \exists^{st} \sigma(u, P_i, P_{i+1}) : \rho(u, C) & i = n-1 \\ \square_1 P_1 \cdots \forall^{st} G_{P_i} \exists^{st} \sigma(u, P_i, P_{i+1}) \forall^{st} P_{i+2} \cup \{\leftarrow u\} \cdots \square_n P_n : C & otherwise \end{cases}.$$

Theorem 4
Let Π be a Guess&Check ASP(Q) program, for each $i \in [1, \dots, n]$ such that $\square_i = \forall^{st}$, Π is coherent iff Π^{GC_i} is coherent.

Intuitively, this theorem holds because in Π^{GC_i} the answer sets of the replaced subprograms are preserved with respect to those in Π. Interpretations that violate constraints become additional answer sets, that are either invalidated in the next universal subprogram or do not affect the coherence of Π. A formal proof follows.

Proof
(I) First assume that Π is incoherent. Obviously, if Π is incoherent due to a P_j with $j < i$, then Π^{GC_i} is incoherent for the same reason.

If Π is incoherent due to P_i, in the following we will consider P_i only (as it is the case when $i = 1$), rather than $P_i \cup fix_{P_{i-1}}(N)$ for some answer set N of the previous level (for $i > 1$). All arguments transfer directly to the latter case.

For any $M \in AS(P_i)$ we know from Proposition 6 that $M = M_G \cup M_C$ where $M_G \in AS(G_{P_i})$ and $M_C \in AS(C_{P_i} \cup fix_{G_{P_i}}(M_G))$. If M is the reason for incoherence in Π, we will show that then M_G is a reason for incoherence in Π^{GC_i}. We distinguish three cases.

(1) If $i = n$, there is $M \in AS(P_i)$ with $C \cup fix_{P_i}(M)$ incoherent. Here, M_C is also the unique answer set of $\tau(u, P_i) \cup fix_{G_{P_i}}(M_G)$, but since $C \cup fix_{P_i}(M)$ is incoherent, M_C does not satisfy $\rho(u, C)$ either (as u is false in it). It follows that there is $M_G \in AS(G_{P_i})$ such that $\sigma(u, P_i, C) \cup fix_{G_{P_i}}(M_G)$ is incoherent, and hence Π^{GC_i} is incoherent.

(2) If $i = n - 1$, there is $M \in AS(P_i)$ such that there is no $M' \in AS(P_n \cup fix_{P_i}(M))$ such that $C \cup fix_{P_n}(M')$ is incoherent. Also here, M_C is also the unique answer set of $\tau(u, P_i) \cup fix_{G_{P_i}}(M_G)$, and each $M' \in AS(P_n \cup fix_{P_i}(M))$ satisfies $\rho(u, P_n)$ as well, because u is false in M', which means that $AS(P_n \cup fix_{P_i}(M)) = AS(\sigma(u, P_i, P_n) \cup fix_{G_{P_i}}(M_G))$. Finally, since u is false in each M', from $C \cup fix_{P_n}(M')$ being incoherent we also get that $\rho(u, C) \cup fix_{P_n}(M')$ is incoherent. Then we have $M_G \in AS(G_{P_i})$ such that there is no $M' \in AS(\sigma(u, P_i, P_n) \cup fix_{G_{P_i}}(M_G))$ such that $\rho(u, C) \cup fix_{P_n}(M')$ is incoherent, hence Π^{GC_i} is incoherent.

(3) If $i < n - 1$, there is $M \in AS(P_i)$ such that there is no $M' \in AS(P_{i+1} \cup fix_{P_i}(M))$ such that $\Pi'_{P_{i+1},M'}$ is incoherent, where Π' is the suffix of Π starting at P_{i+2}. Also here, M_C is also the unique answer set of $\tau(u, P_i) \cup fix_{G_{P_i}}(M_G)$, and each $M' \in AS(P_{i+1} \cup fix_{P_i}(M))$ satisfies $\rho(u, P_{i+1})$ as well, because u is false in M', which means that $AS(P_{i+1} \cup fix_{P_i}(M)) = AS(\sigma(u, P_i, P_{i+1}) \cup fix_{G_{P_i}}(M_G))$. Also, observe that since u is false in each of these M', $AS(P_{i+2} \cup fix_{P_{i+1}}(M')) = AS(P_{i+2} \cup \{\leftarrow u\} \cup fix_{P_{i+1}}(M'))$. So there is $M_G \in AS(G_{P_i})$ such that there is no $M' \in AS(\sigma(u, P_i, P_{i+1}) \cup fix_{G_{P_i}}(M_G))$ such that $\Pi''_{P_{i+1},M'}$ (the suffix of Π^{GC_i} starting at $P_{i+2} \cup \{\leftarrow u\}$) is incoherent, so Π^{GC_i} is incoherent.

(II) Now assume that Π is coherent. As above, in the following we will consider P_i only (as it is in the case when $i = 1$), rather than $P_i \cup fix_{P_{i-1}}(N)$ for some answer set N of the previous level (for $i > 1$).

Here we have to show that from coherence for each $M \in AS(P_i)$ in Π coherence for each $M_G \in AS(G_{P_i})$ follows. If (case a) there is an $M_C \in AS(C_{P_i} \cup fix_{G_{P_i}}(M_G))$ (the unique answer set), this follows quite easily because $M = M_G \cup M_C$ due to Proposition 6 and M_C is the unique answer set of $\tau(u, P_i) \cup fix_{G_{P_i}}(M_G)$, in which u is false. If (case b) $AS(C_{P_i} \cup fix_{G_{P_i}}(M_G))$ is incoherent, then there is a single answer set M_u of $\tau(u, P_i) \cup fix_{G_{P_i}}(M_G)$, in which u is true. We distinguish three cases.

(1) If $i = n$, for any $M \in AS(P_i)$ the program $C \cup fix_{P_i}(M)$ is coherent; let X be one of its answer sets. In case a, $M_C \cup X$ is an answer set of $\sigma(u, P_i, C) \cup fix_{G_{P_i}}(M_G)$. In case b, M_u is an answer set of $\sigma(u, P_i, C) \cup fix_{G_{P_i}}(M_G)$ (as all rules in $\rho(u, C)$ are satisfied by M_u due to u being true). In both cases, Π^{GC_i} is coherent.

(2) If $i = n - 1$, for any $M \in AS(P_i)$ there is an $M' \in AS(P_n \cup fix_{P_i}(M))$ such that $C \cup fix_{P_n}(M')$ is coherent with an answer set X. In case a, $M_C \cup M'$ is an answer set of $\sigma(u, P_i, P_n) \cup fix_{G_{P_i}}(M_G)$, and $C \cup fix_{P_n}(M_C \cup M')$ is coherent with the answer set X. In case b, M_u is an answer set of $\sigma(u, P_i, P_n) \cup fix_{G_{P_i}}(M_G)$ (as all rules in $\rho(u, C)$ are satisfied by M_u due to u being true). But then M_u is also an answer set of $C \cup fix_{P_n}(M_u)$, which is therefore coherent. In both cases, Π^{GC_i} is coherent.

(3) If $i < n - 1$, for any $M \in AS(P_i)$ there is an $M' \in AS(P_{i+1} \cup fix_{P_i}(M))$ such that $\Pi'_{P_{i+1},M'}$ is coherent, where Π' is the suffix of Π starting at P_{i+2}. In case a, $M_C \cup M'$ is an answer set of $\sigma(u, P_i, P_{i+1}) \cup fix_{G_{P_i}}(M_G)$, and $\Pi''_{P_{i+1},M_C \cup M'}$ (the suffix of Π^{GC_i} starting at $P_{i+2} \cup \{\leftarrow u\}$) is coherent since $M_C \cup M' = M'$ and u is false in M'. In case b, M_u is an answer set of $\sigma(u, P_i, P_{i+1}) \cup fix_{G_{P_i}}(M_G)$ (as all rules in $\rho(u, P_{i+1})$ are satisfied by M_u due to u being true). But then $P_{i+2} \cup \{\leftarrow u\} \cup fix_{P_{i+1}}(M_u)$ has no answer sets (because of u), so $\Pi''_{P_{i+1},M_C \cup M'}$ is trivially coherent. In both cases, Π^{GC_i} is coherent. $\square$

We now define a recursive transformation that, given a Guess&Check ASP(Q) program Π, builds a sequence of ASP(Q) programs $(\Pi_1, \ldots, \Pi_n)$ such that the last program of that sequence is both equivalent to Π and features an encoding in CNF.

Definition 6
Let Π be a Guess&Check ASP(Q) program, then

$$
\Pi_i = \begin{cases}
\Pi & i = 1 \wedge \square_i = \exists^{st} \\
\Pi^{GC_1} & i = 1 \wedge \square_i = \forall^{st} \\
\Pi_{i-1} & i \in [2..n] \wedge \square_i = \exists^{st} \\
(\Pi_{i-1})^{GC_i} & i \in [2..n] \wedge \square_i = \forall^{st}
\end{cases} .
$$

Theorem 5
Let Π be a Guess&Check ASP(Q) program, and K be the set of indexes $K = \{k | k \in [1,\ldots,n] \wedge \square_k = \forall^{st}\}$ (i.e., s.t. P_k a universally quantified subprogram), then Π is coherent iff $\Phi^K_{CNF}(\Pi_n)$ is satisfied.

Proof
Observe that, by definition, Π_n is such that all of its universally quantified subprograms are trivial (contain only choice rules). Let $K = \{k | k \in [1,\ldots,n] \wedge \square_i = \forall^{st} \text{ in } \Pi_n\}$, from Theorem 3 it follows that Π_n is coherent iff $\Phi^K(\Pi_n)$ is satisfiable. Moreover, from Proposition 5, we have that $\Phi^K(\Pi_n)$ is equivalent to $\Phi^K_{CNF}(\Pi_n)$. From Theorem 4, we have that, Π is coherent iff Π_n is coherent; since Π_n is coherent iff $\Phi^K(\Pi_n)$ is satisfiable, and $\Phi^K(\Pi_n)$ is equivalent to $\Phi^K_{CNF}(\Pi_n)$, the thesis follows. $\square$

Intuitively, here all universal subprograms are replaced by programs that contain only choice rules and are therefore trivial. The result then follows from Theorem 3, Proposition 5, and Theorem 4.

5 Implementation and experiments

In this section, we describe our implementation and discuss an experimental analysis conducted to (*i*) demonstrate empirically the efficacy of the techniques described above, (*ii*) compare PYQASP with QASP, and (*iii*) compare PYQASP with a recent implementation of the stable unstable semantics (Janhunen 2022).

5.1 Implementation, benchmarks, and experiment setup

The PYQASP system is an implementation in Python of the transformation techniques described in Sections 2.2, 3, and 4. The input ASP(Q) program is transformed into a QBF formula to be processed later by a QBF solver that supports the QCIR format. PYQASP can handle non-propositional inputs, indeed the user can select either gringo (Gebser *et al.* 2011) or iDLV (Calimeri *et al.* 2020) as grounders. The SAT encoding of ASP subprograms ($CNF(\cdot)$) is produced using ASPTOOLS (Janhunen 2004; 2018). The computation of the well-founded-based rewriting (see Section 3) uses the computation of the well-founded model in iDLV. PYQASP is modular, in the sense that the user can choose a QBF solver to use as the back-end. PYQASP supports the same back-ends as QASP, which are based on *DepQBF*, *Quabs*, and *RareQS* QBF solvers. Moreover, PYQASP implements an automatic algorithm selection strategy, devised according to the methodology employed in the ME-ASP multi-engine ASP solver proposed by Maratea *et al.* (2014), that

selects automatically a suitable back-end for the given input. A more detailed description of the evaluation process is available in the online appendix. The source code is available at `https://github.com/MazzottaG/PyQASP.git`.

ASP(Q) Benchmarks. We run a suite of benchmarks that has already been used to assess the performance of ASP(Q) implementations (Amendola *et al.* 2022). The suite contains encodings in ASP(Q) and instances of four problems: Quantified Boolean Formulas (QBF); Argumentation Coherence (AC); Minmax Clique (MMC); and Paracoherent ASP (PAR). The suite comprises a selection of instances from QBF Lib (`https://www.qbflib.org/`), ICCMA 2019 (`http://argumentationcompetition.org/2019`), ASP Competitions (Gebser *et al.* 2017), and PAR instances by Amendola *et al.* (2021). A detailed description of these benchmarks was provided by Amendola *et al.* (2022).

Experiment Setup. All the experiments of this paper were run on a system with 2.30GHz Intel(R) Xeon(R) Gold 5118 CPU and 512GB of RAM with Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-137-generic x86_64). Execution time and memory were limited to 800 seconds (of CPU time, i.e., user+system) and 12 GB, respectively. Each system was limited to run in a single core.

5.2 Impact of the new techniques

Compared methods. We run three variants of PYQASP, namely:

- PYQASP: basic encoding with gringo as grounder;
- PYQASP$_{WF}$: basic encoding with well-founded simplification (iDLV as grounder);
- PYQASP$_{WF+GC}$: well-founded simplification and direct encoding in CNF (i.e., production of a CNF encoding for *guess&check* programs).

These variants were combined with the following three QBF back-ends:

- *RQS*: *qcir-conv.py* (by Klieber - `https://www.wklieber.com/ghostq/qcir-converter.html`) transforms QCIR to the GQ format of *RareQS* solver (by Janota `http://sat.inesc-id.pt/~mikolas/sw/areqs`), that is called.
- *DEPS*: *qcir-conv.py* and *fmla* convert the formula from QCIR to QDIMACS, *bloqqer* (by Biere *et al.* - `http://fmv.jku.at/bloqqer`) simplifies it, then the QBF solver *DepQBF* (by Lonsin - `https://lonsing.github.io/depqbf`) is called.
- *QBS*: The QBF solver *Quabs* (by Tentrup - `https://github.com/ltentrup/quabs`) is called, with no pre-processor.

All this amounts to running nine variants of PYQASP. In our naming conventions, the selected back-end is identified by a superscript, and a subscript identifies the optimizations enabled. For example, PYQASP^{DEPS} indicates PYQASP with back-end *DEPS*, and PYQASP$^{DEPS}_{WF+GC}$ indicates PYQASP with *DEPS* back-end and all optimizations enabled.

Results. Obtained results are summarized in Figure 1, which aggregates the performance of each compared method in four cactus plots, one per considered problem. Recall that, a line in a cactus plot contains a point (x, y) whenever the corresponding system solves at most x instances in y seconds.
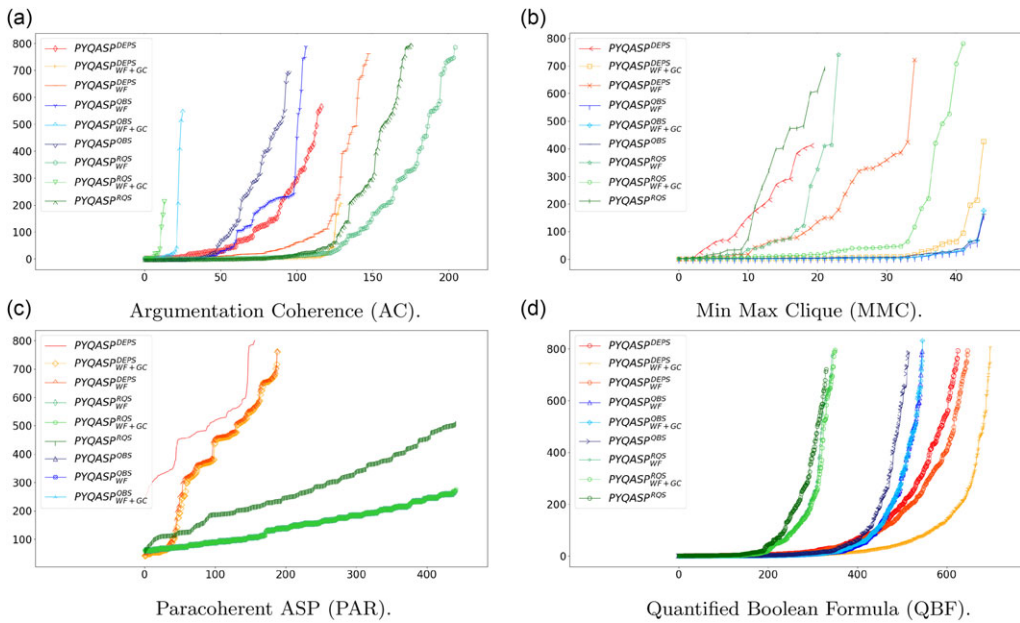
Fig. 1. Analysis of proposed optimizations.

We first observe that the different back-ends are preferable depending on the benchmark domain. In particular, RQS is the fastest option in AC and PAR (see Figures 1a-1c), QBS is the fastest in MMC (see Figure 1b), and DEPS in QBF (see Figure 1d). This behavior confirms the findings of Amendola *et al.* (2022).

The well-founded optimization allows to solve more instances and in less time in AC, MMC, and QBF benchmarks, independently of the back-end, whereas, the identification of *guess&check* programs pays off in terms of solved instances in QBF and MMC, again independently of the back-end solver. The two techniques combine their positive effects in MMC, PAR, and QBF. In particular, PYQASP$_{WF+GC}^{DEPS}$ solves 25 more instances than PYQASP^{DEPS} in MC, and 73 in QBF; moreover, PYQASP$_{WF+GC}^{RQS}$ solves 20 more instances than PYQASP^{RQS} in MC and 20 in QBF. However, the application of the *guess&check* optimization has a negative effect on AC, since the well-founded operator, applied to the rewritten program, is no longer able to derive some simplifications that instead can be derived from the original program. For this reason, PYQASP$_{WF}^{RQS}$ is the best option in AC, solving 29 instances more than PYQASP^{RQS}.

All in all, the results summarized in Figure 1 confirm the efficacy of both well-founded optimization and identification of *guess&check* programs.

5.3 Comparison with qasp

Compared methods. We compare the best variants of PYQASP identified in the previous subsection with QASP running the same back-end QBF solvers. As before, the selected back-end is identified by a superscript. In addition, we run a version of PYQASP capable of selecting automatically a suitable back-end solver for each instance, denoted by
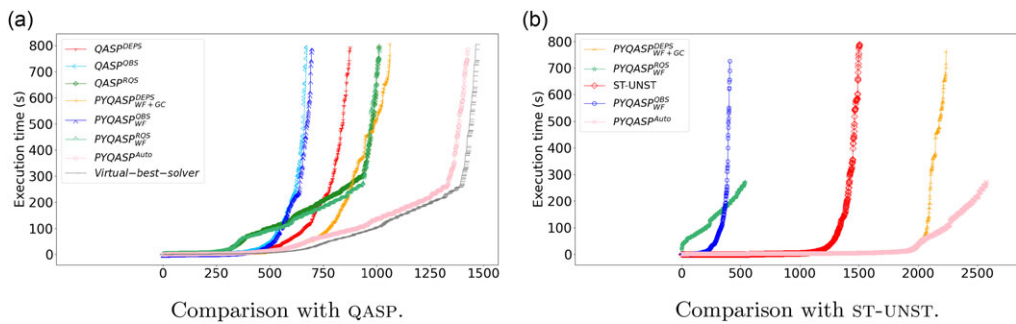
Fig. 2. Comparison with QASP and ST-UNST.

PYQASP^{AUTO}. This latter was obtained by applying to PYQASP the methodology used in the ME-ASP solver (Maratea *et al.* 2014) for ASP. In particular, we measured some syntactic program features, the ones of ME-ASP augmented with the number of quantifiers, existential (resp. universal) atoms count, and existential (resp. universal) quantifiers to characterize QASP instances. Then, we used the *random forest* classification algorithm. We sampled about 25% of the instances (i.e., 1094 instances uniquely solved) from all benchmark domains and split in 30% test set (329 instances) and 70% training set (765 instances), obtaining: 98% accuracy, 95% recall, and 97% of f-measure, which is acceptable. As it is customary in the literature, to assess on the field the efficacy of the algorithm selection strategy, we also computed the Virtual Best Solver (VBS). VBS is the *ideal* system one can obtain by always selecting the best solver for each instance.

Results. The results we obtained are reported in the cactus plot of Figure 2a. First of all, we note that PYQASP is faster and solves more instances than QASP no matter the back-end solver. In particular, PYQASP$^{DEPS}_{WF+GC}$ solves 186 instances more than QASP^{DEPS}, PYQASP$^{RQS}_{WF}$ solves 4 instances more than QASP^{RQS}, and PYQASP$^{QBS}_{WF}$ solves 21 instances more than QASP^{QBS}.

Diving into the details, we observed that PYQASP also uses less memory on average than QASP. Indeed, QASP used more than 12 GB in some instances of PAR and AC, whereas PYQASP never exceeded the memory limit in these domains. This is due to a combination of factors. On the one hand, PYQASP never caches the entire program in main memory; on the other hand, the formulas built by PYQASP are smaller than the ones of QASP, and this causes the back-end QBF solver to use less memory and be faster during the search. (More detailed data on time and memory usage are available in the online appendix)

Finally, as one might expect, the best solving method is PYQASP^{AUTO}. Comparing PYQASP^{AUTO} with the VBS there is only a small gap (38 instances overall). In particular, we observe that, in the majority of cases, the selector is able to pick the best method; it sometimes misses a suitable back-end (especially in MMC which is the smallest and less represented domain in the training set). As a result, PYQASP^{AUTO} is generally effective in combining the strengths of all the back-end solvers. Indeed, PYQASP^{AUTO} solves 363 instances more than PYQASP$^{DEPS}_{WF+GC}$ (i.e., the best variant of PYQASP with fixed back-end) and 414 instances more than QASP^{RQS} (i.e., the best variant of QASP).

5.4 Comparison with stable–unstable

In this section, we compare PYQASP with an efficient implementation of the stable-unstable semantics by Janhunen (2022) on common benchmarks.

Compared methods. In this comparison, we considered the best fixed back-end variants of PYQASP, PYQASP^{AUTO}, with Janhunen's solver (2022), which is labeled ST-UNST.

Benchmarks. ST-UNST can solve only problems on the second level of the PH (more on this in Section 6). Thus, to perform a fair comparison, we considered in addition to PAR (the only problem in our suite having suitable complexity), a set of hard 2-QBF instances generated according to the method by Amendola *et al.* (2020), and the *point of no return* (PONR) benchmark introduced by Janhunen (2022) to assess ST-UNST.

Results. The results are summarized in the cactus plot of Figure 2b. (More details in the online appendix). Analyzing the results in each domain, we report that ST-UNST solves 60 instances of PAR, where the best fixed back-end version of PYQASP (namely, PYQASP$^{RQS}_{WF}$) solves 442. In PONR, ST-UNST solves 30 instances, where PYQASP$^{RQS}_{WF}$ solves 94. In QBF, ST-UNST solves 1416 instances, where PYQASP$^{DEPS}_{WF+GC}$ solves 2048. Finally, PYQASP^{AUTO} is the best method overall, solving a total of 2578 instances, that is 1072 more instances than ST-UNST, which solves 1506 overall.

6 Related work

The most closely related work is the one proposed by Amendola *et al.* (2022) where QASP, the first implementation of a solver for ASP(Q), was introduced. First, we observe that both PYQASP and QASP are based on the translation from ASP(Q) to QBF introduced by Amendola *et al.* (2022)), both resort to the lp2* tools for converting ASP programs to CNF formulas (Janhunen 2004; 2018), and both can be configured with several back-end QBF solvers. QASP is implemented in Java, whereas PYQASP is implemented in Python, which proved to be a very flexible and handy language to implement the composition of tools that is needed to develop a QBF-based system for ASP(Q). QASP processes the entire ASP(Q) program rewriting in main memory, whereas PYQASP implements a more memory-aware algorithm that keeps at most one subprogram in main memory. This implementation choice was empirically demonstrated to overcome the high memory usage limiting the performance of QASP described by Amendola *et al.* (2022). QASP uses gringo (Gebser *et al.* 2011) as grounder, whereas PYQASP can be configured to use both gringo and iDLV (Calimeri *et al.* 2020), and offers an interface that makes it easier to integrate external QBF solvers. It is worth pointing out that PYQASP supports novel rewriting techniques that result in more efficient encoding in QBF (see Section 4) that are absent in QASP.

Concerning closely related formalisms that feature an implementation, we mention the stable–unstable semantics (Bogaerts *et al.* 2016) and quantified answer set semantics (Fandinno *et al.* 2021).

The stable–unstable semantics was first supported by a proof of concept prototype (Bogaerts *et al.* 2016); later, Janhunen (2022) proposed an implementation based on a rewriting to plain ASP. The implementation proposed by Janhunen (2022) employed ASPTOOLS for some preprocessing, but the transformations and the solving techniques

are different w.r.t. PYQASP. Stable–unstable semantics can be used to model problems in the second level of the PH; thus, our implementation can handle problems of higher complexity. From the usage point of view, we observe that in the system of Janhunen (2022) the user is required to define the interface of modules (by means of ASP programs) and to manually combine the tool chain, whereas in PYQASP this is done in a more accessible way. An empirical comparison of PYQASP with Janhunen's system (2022) is provided in Section 5.4.

The quantified answer set semantics (Fandinno *et al.* 2021) was also implemented by resorting to a translation to QBF (Fandinno *et al.* 2021). However, the difference in the semantics of quantifiers results in a quite different translation to QBF with respect to ASP(Q). Translations from quantified answer set semantics to ASP(Q) and back were proposed by Fandinno *et al.* (2021) but never implemented.

Finally, we refer the reader to the works proposed by Amendola *et al.* (2019) and Fandinno *et al.* (2021) for an exhaustive comparison of the ASP(Q) language with alternative formalisms and semantics.

7 Conclusion

An important aspect that can boost the adoption of ASP(Q) as a practical tool for developing applications is the availability of more efficient implementations. In this paper, we present PYQASP, a new system for ASP(Q) that features both a memory-aware implementation in Python and a new optimized translation of ASP(Q) programs in QBF. In particular, PYQASP exploits the well-founded operator to simplify ASP(Q) programs and can recognize a (popular) class of ASP(Q) programs that can be encoded directly in CNF and thus do not require to perform any additional normalization to be handled by QBF solvers. Moreover, PYQASP is able to select automatically a suitable back-end for the given input and can deliver steady performance over varying problem instances. PYQASP outperforms QASP, the first implementation of ASP(Q), and pushes forward the state of the art in ASP(Q) solving.

As future work, we plan to further optimize PYQASP by providing more efficient encodings in QBFs and improve the algorithm selection model with extended training and a deeper tuning of parameters.

Supplementary material

To view supplementary material for this article, please visit http://doi.org/10.1017/S1471068423000121.

References

AMENDOLA, G., CUTERI, B., RICCA, F. AND TRUSZCZYNSKI, M. 2022. Solving problems in the PH with ASP(Q). In *Proceedings of LPNMR*. LNCS, vol. 13416. Springer, 373–386.

AMENDOLA, G., DODARO, C., FABER, W. AND RICCA, F. 2021. Paracoherent answer set computation. *Artif. Intell. 299*, 103519.

AMENDOLA, G., RICCA, F. AND TRUSZCZYNSKI, M. 2019. Beyond NP: quantifying over answer sets. *TPLP 19,* 5–6, 705–721.

AMENDOLA, G., RICCA, F. AND TRUSZCZYNSKI, M. 2020. New models for generating hard random boolean formulas and disjunctive logic programs. *Artif. Intell. 279.*

BOGAERTS, B., JANHUNEN, T. AND TASHARROFI, S. 2016. Stable-unstable semantics: Beyond NP with normal logic programs. *TPLP 16,* 5–6, 570–586.

BREWKA, G., EITER, T. AND TRUSZCZYNSKI, M. 2011. Answer set programming at a glance. *Commun. ACM 54,* 12, 92–103.

CALIMERI, F., DODARO, C., FUSCÀ, D., PERRI, S. AND ZANGARI, J. 2020. Efficiently coupling the I-DLV grounder with ASP solvers. *TPLP 20,* 2, 205–224.

DANTSIN, E., EITER, T., GOTTLOB, G. AND VORONKOV, A. 2001. Complexity and expressive power of logic programming. *ACM Comput. Surv. 33,* 3, 374–425.

EITER, T. AND GOTTLOB, G. 1995. On the computational cost of disjunctive logic programming: Propositional case. *Ann. Math. Artif. Intell. 15,* 3–4, 289–323.

ERDEM, E., GELFOND, M. AND LEONE, N. 2016. Applications of answer set programming. *AI Magazine 37,* 3, 53–68.

FABER, W. AND MORAK, M. 2022. Evaluating epistemic logic programs via answer set programming with quantifiers. In *HYDRA/RCRA@LPNMR.* CEUR WS, vol. 3281, 78–89.

FABER, W., MORAK, M. AND CHRPA, L. 2022. Determining action reversibility in STRIPS using asp with quantifiers. In *PADL.* LNCS, vol. 13165. Springer, 42–56.

FANDINNO, J., LAFERRIÈRE, F., ROMERO, J., SCHAUB, T. AND SON, T. C. 2021. Planning with incomplete information in quantified answer set programming. *TPLP 21,* 5, 663–679.

GEBSER, M., KAMINSKI, R., KÖNIG, A. AND SCHAUB, T. 2011. Advances in *gringo* series 3. In *LPNMR 2011. Proceedings.* LNCS, vol. 6645. Springer, 345–351.

GEBSER, M., LEONE, N., MARATEA, M., PERRI, S., RICCA, F. AND SCHAUB, T. 2018. Evaluation techniques and systems for answer set programming: a survey. In *Proceedings of IJCAI 2018.* ijcai.org, 5450–5456.

GEBSER, M., MARATEA, M. AND RICCA, F. 2017. The sixth answer set programming competition. *J. Artif. Intell. Res. 60,* 41–95.

GELFOND, M. AND LIFSCHITZ, V. 1991. Classical negation in logic programs and disjunctive databases. *New Gener. Comput. 9,* 3/4, 365–386.

JANHUNEN, T. 2004. Representing normal programs with clauses. In *Proceedings of ECAI'2004.*, R. L. de Mántaras and L. Saitta, Eds. IOS Press, 358–362.

JANHUNEN, T. 2018. Cross-translating answer set programs using the ASPTOOLS collection. *Künstliche Intell. 32,* 2–3, 183–184.

JANHUNEN, T. 2022. Implementing stable-unstable semantics with ASPTOOLS and clingo. In *PADL 2022, Proceedings.* LNCS, vol. 13165. Springer, 135–153.

LIFSCHITZ, V. 2002. Answer set programming and plan generation. *Artif. Intell. 138,* 1–2, 39–54.

MARATEA, M., PULINA, L. AND RICCA, F. 2014. A multi-engine approach to answer-set programming. *TPLP 14,* 6, 841–868.

PULINA, L. AND SEIDL, M. 2019. The 2016 and 2017 QBF solvers evaluations (qbfeval'16 and qbfeval'17). *Artif. Intell. 274,* 224–248.

VAN GELDER, A., ROSS, K. A. AND SCHLIPF, J. S. 1991. The well-founded semantics for general logic programs. *J. ACM 38,* 3, 620–650.