

Towards Exploiting Partial Knowledge in Declarative Domain-Specific Heuristics for ASP

Richard Taupe^{1,2}, Konstantin Schekotihin², Peter Schüller³,
Antonius Weinzierl^{3,4}, and Gerhard Friedrich²

¹ Siemens AG Österreich

richard.taupe@siemens.com

<https://www.siemens.com/innovation>

² Alpen-Adria-Universität Klagenfurt

{konstantin.schekotihin,gerhard.friedrich}@aau.at

³ Technische Universität Wien, Institut für Logic and Computation, KBS Group

{ps,weinzierl}@kr.tuwien.ac.at

⁴ Aalto University, Department of Computer Science

Abstract. Domain-specific heuristics are an important technique for solving combinatorial problems efficiently. We propose a novel semantics for declarative specifications of domain-specific heuristics in Answer Set Programming (ASP). Our approach is more intuitive than existing ones because heuristics can use negation as failure and aggregates, both of which are evaluated on a partial solver state. Such conditions are a frequent ingredient of existing domain-specific heuristics, e.g., for placing an item that has not been placed yet in bin packing. State-of-the-art solvers do not allow a declarative specification of such preconditions. We implement support for heuristic directives under this semantics in the lazy-grounding ASP system Alpha and experimentally validate that the combination of ASP solving with lazy grounding and our novel heuristics can be a vital ingredient for solving industrial-size problems.

Keywords: answer set programming · domain-specific heuristics · lazy grounding

1 Introduction

Answer Set Programming (ASP) [4, 19, 26] is a declarative knowledge representation formalism that has been applied successfully in a variety of industrial and scientific applications such as configuration [2, 29], team building [37], molecular biology [36], planning [12], and others [13, 15]. In the vast majority of these applications well-known ASP solvers, such as CLINGO [18] or DLV [31], applied the ground-and-solve approach. That is, such solvers first instantiate the given non-ground program and then apply various strategies to find answer sets of the obtained ground program.

Modern applications showed however that there are two issues with the ground-and-solve approach. First, problem instances in industrial applications can be quite large and cannot be grounded by modern grounders like GRINGO [21] or I-DLV [5] in acceptable time and/or space [11]. Second, even if the problem can be grounded, computation of

answer sets might take considerable time, as indicated by the results of the last ASP Competitions [6, 24].

There are two directions in the research on ASP systems aiming to resolve these issues. In the first case, lazy grounding ASP systems, such as GASP [8], ASPERIX [30], OMIGA [9], or ALPHA [40], interleave grounding and solving in order to instantiate and store only relevant parts of the ground program in memory.

To overcome the second issue modern solvers employ various techniques. Among them the ability to use domain-specific heuristics is fundamental to solve complex problems [25]. In the first approach [22] heuristics are specified using a dedicated declarative language as a part of the encoding. The solver then evaluates all heuristics rules as a part of the program. The approach presented in [10] allows for specification of procedural heuristics that interact directly with the internal decision-making procedures and therefore can dynamically evaluate heuristics wrt. a partial solution. For example, a heuristic for bin packing may need to compute the amount of space left in a bin after an item is placed into it. As the authors show in [10], static heuristics [22] should generate all possible amounts of space left in bins after all possible placements of items in those bins. Dynamic heuristics can compute all required sums on-the-fly given a partial assignment of items to bins.

However, procedural heuristics of Dodaro et al. [10] counteract the declarative nature of ASP. Also, declarative heuristics for the lazy-grounding case have not yet been addressed. They have to be adapted for such systems because of the different solving mechanisms in effect.

In this work we present a novel approach that combines lazy-grounding ASP systems with dynamic declarative heuristics for solution of large and complex problems. In summary, our work makes the following contributions:

- we present a variant of the declarative language presented in [22] and equip it with a novel semantics that makes declarative specifications of domain-specific heuristics more intuitive;
- we show how the language can be integrated into a well-known lazy-grounding ASP system ALPHA and provide a reference implementation;
- finally, we demonstrate the benefits of our approach by preliminary experimental results.

The remainder of this paper is organized as follows: After briefly describing ASP’s syntax and semantics in Section 2, we discuss the state of the art of domain-specific heuristics in ASP in Section 3. Then we present a novel semantics for such heuristics in Section 4 and show how to integrate it into a lazy-grounding ASP solver in Section 5. Finally, experimental results are presented and discussed in Section 6.

2 Preliminaries

Answer Set Programming (ASP) [4, 19, 26] is an approach to declarative programming. Instead of stating how to solve a problem at hand, the programmer should formulate the problem in the form of a logic program. An ASP solver then finds models (so-called *answer sets*) for this logic program, which correspond to solutions for the original problem.

2.1 Syntax

An answer-set program P is a finite set of rules of the form

$$h_1; \dots; h_d \leftarrow b_1, \dots, b_m, \text{ not } b_{m+1}, \dots, \text{ not } b_n. \quad (1)$$

where h_1, \dots, h_d and b_1, \dots, b_m are positive literals (i.e. atoms) and $\text{not } b_{m+1}, \dots, \text{not } b_n$ are negative literals.

An atom is either a classical atom, a cardinality atom, or an aggregate atom. A classical atom is an expression $p(t_1, \dots, t_n)$ where p is an n -ary predicate and t_1, \dots, t_n are terms. If p is preceded by a minus sign indicating the strong negation, then the classical atom is negative, otherwise it is positive. ASP knows a second kind of negation: A literal is either an atom a or its default negation $\text{not } a$. Default negation refers to the absence of information, i.e. an atom is assumed to be false as long as it is not proven to be true. Note that, while strong negation is usually compiled away, in this work it will be vital to combine strong and default negation to express domain-specific heuristics over partial solver assignments.

A *cardinality atom* is of the form $l \{a_1 : l_{i_1}, \dots, l_{i_m}; \dots; a_n : l_{n_1}, \dots, l_{n_o}\} u$, where

- $a_i : l_{i_1}, \dots, l_{i_m}$ represent *conditional literals* in which a_i (the head of the conditional literal) and all l_{i_j} are literals, and
- l and u are terms representing non-negative integers indicating lower and upper bound. If one or both of the bounds are not given, their defaults are used, which are 0 for l and ∞ for u .

As an extension of cardinality atoms, ASP also supports aggregate atoms that apply aggregate functions like *max*, *min* or *sum* to such sets [3].

Given a rule r , $H(r) = \{h_1, \dots, h_d\}$ is called the *head* of r , and $B(r) = \{b_1, \dots, b_m, \text{not } b_{m+1}, \dots, \text{not } b_n\}$ is called the *body* of r . A rule r with $H(r)$ consisting of a single cardinality atom is called *choice rule*. A rule r with a head consisting of more than one classical atom (i.e. $|H(r)| > 1$) is called *disjunctive rule*. A rule r with $H(r)$ consisting of at most one classical atom is called a *normal rule*. A normal rule r where $H(r) = \{\}$, e.g. $\leftarrow b.$, is called *integrity constraint*, or simply *constraint*. A normal rule r where $B(r) = \{\}$, e.g. $h \leftarrow .$, is called *fact*.

2.2 Semantics

There are several ways to define the semantics of an answer-set program, i.e. to define the set of answer sets $AS(P)$ of an answer-set program P . An overview is provided by [33]. Probably the most popular semantics is based on the *Gelfond-Lifschitz reduct* [27]. The *FLP semantics* also covers aggregates [14]. A variant that applies to choice rules also is presented in [6].

Informally, an answer set A of a program P is a subset-minimal model of P (i.e. a set of atoms interpreted as *true*) which satisfies the following conditions: All rules in P are satisfied by A ; and all atoms in A are “derivable” by rules in P . A rule is satisfied if its head is satisfied or its body is not. The disjunctive head of a rule is satisfied if at least one of its atoms is. Rules containing cardinality atoms or aggregates are usually rewritten

by the solver, a process in which auxiliary atoms are introduced. A cardinality atom is satisfied if $l \leq |C| \leq u$ holds, where C is the set of head atoms in the cardinality atom whose conditions (e.g. l_{i_1}, \dots, l_{i_m} for a_i) are satisfied and which are satisfied themselves. Similarly, an aggregate atom is satisfied if the value computed by the aggregate function respects the given bounds. For example, $1 = \#sum\{1 : a; 2 : b\}$ is satisfied if a but not b is true.

3 Domain-Specific Heuristics in ASP: State of the Art

State-of-the-art ASP solvers are well suited to solve a wide range of problems, as shown in the ASP competitions and other experiments reported in the literature (see, e.g., [6, 7, 13]). However, applying general ASP solvers to large instances of industrial problems often does not perform well enough. In some cases, sophisticated encodings or solver tuning methods, e.g. portfolio solvers like CLASPFOLIO [28] or ME-ASP [34], can significantly improve performance.

However, the major breakthrough in solving industrial configuration problems was achieved by applying domain-specific heuristics to ASP. For example, the HWASP solver could find solutions for all available instances of the Partner Units Problem (PUP) using a number of externally embedded heuristics [10].

There is a number of approaches to embed heuristic knowledge into the ASP solving process. HWASP [10] is an extension of the solver WASP that facilitates the integration of external heuristics implemented in a procedural language which are consulted at specific points during the solving process via an API.

One of the first approaches integrating domain-specific heuristics in ASP was suggested in [22]. It consists of an extension of the ASP language that allows for declarative specification of atom weights and signs for the internal heuristics of the CLASP solver. The initial suggestion of using atoms of a dedicated heuristic predicate was later revised. Currently, the CLINGO system supports `#heuristic` directives, which are described in detail in [17, Section 10]. The weight of an atom influences the order in which atoms are considered by the solver when making a decision and a sign modifier instructs whether the selected atom must be assigned true or false.

In CLINGO, the following (non-ground) meta statement defines domain-specific heuristics, where A is an atom, B is a rule body, and w , p , and m are terms [17].

$$\#heuristic A : B. \qquad [w@p, m] \qquad (2)$$

The optional term p gives a preference between heuristic values for the same atom (preferring those with higher p). The term m specifies the type of heuristic information: `m=true` specifies that A should be guessed to true with weight w if B is true, `m=false` is the analog heuristic for false. The weight defines a partial order over atoms and atoms with a higher weight are assigned a value before atoms with a lower weight. Further values for m are `init` and `factor`, which allow to replace initial scores and dynamically modify scores assigned to atoms by the VSIDS heuristic underlying the domain-specific heuristics [17, 22].

To the best of our knowledge, the first account of domain-specific heuristics in the special case of lazy-grounding ASP solving is presented in [39], where heuristic decisions are also made procedurally.

Although purely declarative approaches are preferable to those resorting to procedural means, so far declarative approaches suffer from a modelling issue that we illustrate in the following.

Consider the following program containing two heuristic directives:

```
{ a(2) ; a(4) ; a(6) ; a(8) ; a(5) } ← .
← #sum { X : a(X) } = S, S \ 2 ≠ 0.           % S mod 2 ≠ 0
#heuristic a(5).                               [1,true]
#heuristic a(4) : not a(5).                     [2,true]
```

The program guesses a subset of $\{a(2), a(4), a(6), a(8), a(5)\}$, and the constraint forbids that the sum of the extension of $a/1$ is odd, i.e., $a(5)$ must not be chosen. The heuristic statements specify that $a(5)$ shall be set to true with weight 1, and that $a(4)$ shall be set to true if *not* $a(5)$ is true with weight 2.

In solving this program, CLINGO (v. 5.2.2) used in our experiments first assigns $a(5)$ to true, although $a(4)$ has a higher level and $a(5)$ is not known to be true in the beginning. Next, $a(8)$ is chosen to be false, the solver backtracks and only $a(5)$ stays assigned. Finally, $a(8)$ is chosen to be true and a conflict is learned such that after backtracking $a(5)$ is also made false. Now that *not* $a(5)$ is satisfied, the second heuristic chooses $a(4)$ to be true, and we obtain the answer set $\{a(4), a(8)\}$ after a few more guesses on the yet unassigned atoms.

The second heuristic becomes active only late because *not* $a(5)$ is evaluated with respect to the answer set where it is true if $a(5)$ is false. This has small implications in our toy example but might be crucial for industrial problems where heuristics must be defined depending on the variables that were not assigned so far by the solver. Therefore, we propose to evaluate negation as failure (i.e., *not*) in heuristic statements with respect to the solver process. Then, *not* X is true if X is false or still unassigned.

Similarly, heuristics comprising aggregate atoms are not evaluated by the current semantics as expected. Consider the *best-fit* heuristic for the *bin-packing problem* (BPP) that suggests to place items in a bin such that after placement the amount of free space in the bin is minimal. For a simple BPP, where sizes of items correspond to their numbers, this heuristic can be encoded using aggregate atoms as follows:

```
1{ in(I,B) : bin(B) }1 ← item(I).
← #sum { I : in(I,B) } > C, bcap(C), bin(B).
#heuristic in(I,B) : bin(B), item(I), bcap(C),
    C ≥ S + I, S = #sum { I' : in(I',B) }.    [S + I, true]
```

The program guesses among possible assignments of items to bins ($in/2$) and forbids those guesses in which the sum of item sizes assigned to one bin is greater than the capacity ($bcap/1$). The heuristic directive assigns level and sign values to atoms over $in/2$ and thus aims to influence choices made by the solver. According to the best-fit heuristic, for any item I and bin B the aggregate atom sums up sizes of all items in B including I . If this sum is greater than the bin capacity C , then no level and sign are assigned to an atom in the head and, therefore, such atoms will have the smallest

priority among all choice atoms. Otherwise, the larger the sum, the more preferred is the assignment.

For example, if a BPP instance with three bins, five items, and bin capacity set to 5:

$$bcap(5). \quad bin(1..3). \quad item(1..5).$$

then according to the heuristic the solver should place item 5 first, since the sum of items in a bin after placing this item is 5 and, consequently, the remaining space 0. Next, the heuristic would suggest to place item 4 into some bin,⁵ followed by the item 1 in the same bin, and so forth.

Nevertheless, evaluation of aggregate atoms using the current semantics of heuristic directives does not allow for expected evaluation of the best-fit heuristic. Let partial assignment A include all facts and nothing else, i.e. $A = \{bcap(5), bin(1), \dots, item(5)\}$. Then, one would expect that each body of the heuristic directives is evaluated to true with all aggregate functions returning 0, since none of the atoms over $in/2$ predicate are true. Given the results of the heuristic evaluation the solver selects one of the atoms $in(5, 1)$, $in(5, 2)$, or $in(5, 3)$ and assigns it to true. However, after the item 5 is assigned, e.g. $A = \{bcap(5), bin(1), \dots, item(5)\} \cup \{in(5, 3)\}$, evaluation of the all aggregate atoms would be different for all ground heuristic directives, e.g. which body comprises the atom $bin(3)$ the sum would be 5 instead of 0. This behavior is however impossible given the standard semantics of aggregates [14, 16] implemented in CLINGO.

4 A Novel Semantics for Declarative Domain-Specific Heuristics in ASP

Supporting the declarative specification of domain-specific heuristics in ASP plays an important role in enabling ASP to solve large-scale industrial problems. Although the language and semantics of heuristic directives in CLINGO have shown to be beneficial in many cases, dynamic aspects of negation as failure and aggregates in heuristic conditions have not been addressed satisfactorily. An alternative approach is necessary.

In this paper, we present a novel semantics for heuristic directives in ASP that improves this situation.

Definition 1. *Slightly different from (2), we define a heuristic directive to have the following form, where h is a classical atom, $c_1, \dots, c_k, not\ c_{k+1}, \dots, not\ c_n$ is a rule body (the so-called heuristic condition), and w and l are terms:*

$$\#heuristic\ h : c_1, \dots, c_k, not\ c_{k+1}, \dots, not\ c_n. \quad [w@l]$$

A heuristic directive must be safe, i.e. all variables occurring in it must also occur in $\{c_1, \dots, c_k\}$.

⁵ Note that this depends on whether the solver can recognize at this point that an item can be placed into only one bin. If this is not the case, an additional atom $not\ in(I, _)$ can be used in the condition of the heuristic directive to prevent the heuristic from preferring to assign item 5 to a second bin after placing it in the first one.

This proposal differs from CLINGO’s in the following ways: We do not use the modifier m . Instead, the sign can be implicitly encoded in h : If h is a positive atom, the heuristic makes the solver guess h to be true; if it is of the strongly negated form $-a$, a will be made false. Instead of weight w and tie-breaking priority p , we use terms w and l denoting weight and level as familiar from optimize statements in ASP-Core-2 [3] or weak constraints in DLV [31]. Level is more important than weight, both default to 1, and together they are called priority.

We now describe our semantics more formally. In this description, we assume that the underlying solver can assign to any atom one of the three values: *true* (denoted with \mathbf{T}), *false* (\mathbf{F}), and *must-be-true* (\mathbf{M}) (cf. [40]). For solvers that do not use the third truth value \mathbf{M} , the following definitions can be used without modification, the set of atoms assigned must-be-true will just be empty in this case. Also, we will use the following notations in the definitions below:

The *pos* modifier removes strong negation from an atom, i.e. $pos(-a) = pos(a) = a$. The *head* of a heuristic directive d of the form given in Definition 1 is denoted by $H(d) = h$, its *weight* by $weight(d) = w$ if given, else 1, and its *level* by $level(d) = l$ if given, else 1. A (partial) *assignment* A is a set of signed literals over \mathbf{T} , \mathbf{F} , and \mathbf{M} . Its *atom projection* $A^\pm := \{a \mid \mathbf{T}a \in A \text{ or } \mathbf{M}a \in A\} \cup \{-a \mid \mathbf{F}a \in A\}$ maps atoms currently assigned \mathbf{T} or \mathbf{M} to positive atoms and those assigned \mathbf{F} to negative atoms. The (*heuristic*) *condition* of a heuristic directive d is denoted by $c(d) := \{c_1, \dots, c_k, not\ c_{k+1}, \dots, not\ c_n\}$. The *positive condition* is $c^+(d) := \{c_1, \dots, c_k\}$ and the *negative condition* is $c^-(d) := \{c_{k+1}, \dots, c_n\}$.

Definition 2. *Given a ground heuristic directive d and a partial assignment A , $c(d)$ is satisfied w.r.t. A iff $c^+(d) \subseteq A^\pm$ and $c^-(d) \cap A^\pm = \emptyset$.*

Intuitively, a heuristic condition is satisfied if and only if its positive part is already true and all its default-negated literals are either false or unassigned.

Definition 3. *A ground heuristic directive d is applicable w.r.t. a partial assignment A iff: $c(h)$ is satisfied, $\mathbf{T}pos(H(d)) \notin A$, and $\mathbf{F}pos(H(d)) \notin A$.*

Intuitively, a heuristic directive is applicable if and only if its condition is satisfied and its head is assigned neither \mathbf{T} nor \mathbf{F} . If the head is \mathbf{M} , the heuristic directive may still be applicable, however: The intuition of this is that \mathbf{M} is a ‘weak’ truth value which may be overridden by a ‘strong’ one by a guess driven by the heuristic.

Definitions 2 and 3 reveal the main difference between the semantics proposed here and the one implemented by CLINGO: In our approach, strong negation can be used in heuristic conditions to reason about atoms that are already assigned \mathbf{F} in a partial assignment, while default negation can be used to reason about atoms that are assigned \mathbf{F} or still unassigned. Our semantics truly means default negation in the current partial assignment, while the one implemented by CLINGO basically amounts to strong negation in the current search state. This difference is crucial, since reasoning about incomplete information is important in many cases. An example is a heuristic for bin packing that only applies to items not yet placed.

What remains to be defined is the semantics of weight and level. Given a set of applicable heuristic directives, from the ones on the highest level one with the highest

weight will be chosen. If there are several with the same maximum priority, the solver can use a domain-independent heuristic like VSIDS [35] as a fallback to break the tie.

Definition 4. Given a set D of applicable ground heuristic directives, the subset eligible for immediate choice is defined as $maxpriority(D)$ in two steps:

$$maxlevel(D) := \{d \mid d \in D \text{ and } level(d) = \max_{d \in D} level(d)\}$$

$$maxpriority(D) := \{d \mid d \in maxlevel(D) \text{ and } weight(d) = \max_{d \in maxlevel(D)} weight(d)\}$$

After choosing a heuristic using $maxpriority$, a CDNL-based ASP solver makes a decision on the head atom of the heuristic directive. The other parts of CDNL, e.g. deterministic propagation, are unaffected by this.

Example 1. Consider the program given in section 3. Its heuristic directives, when converted to the syntax proposed in Definition 1, look like directives (3) and (4) in the following program. Consider also the newly introduced directives (5) and (6) in this program.

$$\begin{array}{lll} \#heuristic \ a(5). & [1] & (3) \\ \#heuristic \ a(4) : \ not \ a(5). & [2] & (4) \\ \#heuristic \ -a(5) : \ a(4). & [2] & (5) \\ \#heuristic \ a(6) : \ -a(5). & [2] & (6) \end{array}$$

Intuitively, directive (3) unconditionally prefers to make $a(5)$ true with weight 1. Note that the weight 1 is just given for clarity and could be omitted because 1 is the default weight. The other directives all have the higher weight 2, but they become applicable at different points in time. Directive (4) prefers to make $a(4)$ true if $a(5)$ is not true, directive (5) prefers to make $a(5)$ false if $a(4)$ is true and directive (6) prefers to make $a(6)$ true if $a(5)$ is false.

Let $A_0^\pm = \emptyset$ be the atom projection of the empty partial assignment before any decision has been made. W.r.t. this assignment, (3) is applicable because its condition is empty and its head is still unassigned. Directive (4) is also applicable, because $a(5)$ is still unassigned. Directives (5) and (6) are not applicable w.r.t. the empty assignment. From (3) and (4), (4) is chosen because it has the higher priority. Thus, $a(4)$ is assigned **T**, which updates our assignment to $A_1^\pm = \{a(4)\}$. This makes (5) applicable, $a(5)$ is assigned **F** and our assignment is $A_2^\pm = \{a(4), -a(5)\}$. Note that the condition of (4) was still satisfied at this point, but it was not applicable because its head was already assigned. Now, also (3) is not applicable anymore and the only directive that remains is (6). Since it is applicable, $a(6)$ is made true and added to the assignment. Next, the atoms that remained unassigned are guessed by the default heuristic until an answer set is found. Note that all choices driven by the heuristics are only possible to be executed by the solver because there are corresponding rules in the input program that can fire.

5 Integration into a Lazy-Grounding ASP Solver

Most ASP systems split the evaluation into grounding and solving. The first step produces the grounding of a program, i.e. its variable-free equivalent. Thereby, the vari-

ables in each rule of the program are substituted by constants. The second step then solves this propositional encoding. The associated blow-up in space leads to the so-called *grounding bottleneck* which is tackled by lazy grounding [32,40].

The approach presented in Section 4 is not tailored towards a specific solving paradigm in ASP. We describe in this section how to integrate it into a lazy-grounding ASP solver. Integration into a ground-and-solve system belongs to future work. The lazy-grounding system we are working with is ALPHA, which is briefly described in the following paragraphs. More details can be found in [40].

5.1 ALPHA: Answer Set Solving with Lazy Grounding

Alpha combines lazy grounding with CDNL search (cf. [23]) to avoid the grounding bottleneck of ASP and obtain very good search performance. CDNL-based ASP solvers require a fully grounded input, usually in the form of nogoods. Alpha provides this by having two dedicated components: a lazy grounder and a modified CDNL solver. This separation is common for pre-grounding ASP solvers, but for ALPHA these components interact cyclically: whenever the solving component derives new truth assignments to atoms, the grounding component is queried for new ground nogoods obtainable by the new assignments. In contrast to traditional CDNL-based solving, the result of this interplay is a computation sequence.

Most importantly, the solver does not guess on each atom whether it is true or false, but it guesses on ground instances of rules whether they fire or not. This is realised by creating a unique atom for each ground body and then guessing on these body-representing atoms (henceforth called *choice points*). The solver can guess on a choice point if it is *active*, which is the case when the corresponding ground rule is *applicable*. A rule is applicable w.r.t a partial assignment A^\pm if its positive body has already been derived and its negative body is not contradicted, i.e. $B^+(r) \subseteq A^\pm$ and $B^-(r) \cap A^\pm = \emptyset$.

Intuitively, the ALPHA algorithm incrementally grounds those rules whose positive body is already satisfied. Supportedness of answer sets is not achieved by completion nogoods, but by optionally designating one literal in a nogood as the nogood’s head and deriving the justified truth value ‘true’ (**Ta**) when propagating to heads, while the not-yet-justified truth value ‘must-be-true’ (**Ma**) is propagated in other directions.

Note that ALPHA does not (yet) support the full language of ASP as described in section 2.1. For example, it supports neither disjunctive rules nor upper bounds of cardinality and aggregate atoms.

5.2 Respecting heuristic directives in lazy grounding and solving

To be able to reuse standard grounding procedures, a preprocessor component in ALPHA transforms heuristic directives occurring in input programs to normal rules with head atoms of a built-in predicate, henceforth called *heuristic rules*. The body of the heuristic rule equals the heuristic condition, while information on weight, level, heuristic head and sign are stored in the head of the rule. Due to this transformation, a heuristic directive is grounded under the same precondition as a normal rule: which is, when its positive body is satisfied.

The way the heuristic directive is grounded is similar to the way a normal rule is grounded, but still has to be different. Bodies of heuristic rules are not represented as choice points, but as a distinct type of atoms, such that the solver can then treat heuristic rules differently.

When a heuristic rule is applicable (i.e. its positive body is already derived and its negative body is not yet contradicted), it is not eligible for choice but the heuristic itself becomes applicable. When a heuristic rule ceases to be applicable, the corresponding heuristic information is also not used by the solver any longer. Thus, a heuristic condition is satisfied if and only if its corresponding heuristic rule is applicable.

The task of finding the applicable heuristic with the highest priority is aided by the use of efficient data structures like a heap. When this heuristic is found, it cannot be chosen immediately in a lazy-grounding system like ALPHA. The reason for this is that the head of a heuristic directive is an ordinary atom, but ALPHA can only choose on choice points – on atoms representing rule bodies. Therefore, an additional step is necessary: The set of known ground rules that can derive the chosen atom is determined. From this set, a fallback heuristic chooses one. Then, the choice point corresponding to this rule is assigned true or false, depending on the sign given by the heuristic directive. Propagation following this choice will immediately assign the desired truth value to the atom originally chosen.

Example 2. Consider the following program P :

$$\begin{aligned} &x(1..2). \\ &\{a(X) : x(X)\}. \\ &b(X) \leftarrow x(X), \text{not } c(X). \\ &c(X) \leftarrow x(X), \text{not } b(X). \\ &\#heuristic\ b(X) : x(X), \text{not } a(X). \quad [X@2] \end{aligned}$$

Let $h/3$ be the built-in predicate used to define heads of heuristic rules. In a pre-processing step, the heuristic statement in P is first translated to the heuristic rule $h(b(X), X, 2) \leftarrow x(X), \text{not } a(X)$. Since the positive body of every rule in P is satisfied, the full grounding of P is immediately produced. Under the initial partial assignment consisting just of facts $A_0^\pm = \{x(1), x(2)\}$, both ground heuristic rules are applicable, since both their positive bodies are satisfied and neither $a(1)$ nor $a(2)$ is assigned yet. The directive in which X has been substituted by 2 has the higher weight, however. For this reason, it is chosen, and the solver finds the (in this case) only rule that can make the heuristic's head $b(2)$ true: $b(2) \leftarrow x(2), \text{not } c(2)$. The choice point representing the body of this rule is made true and, after some propagation, the new partial assignment will contain $b(2)$ (amongst other consequences of propagation).

6 Experimental Results

We ran a set of experiments on encodings of the House Reconfiguration Problem (HRP). The HRP is an abstracted version of an industrial reconfiguration problem [38]. Given a set of things that belong to one person each, the task is to assign these things to cabinets

and the cabinets to rooms such that no room contains things of more than one person. Things and cabinets have various sizes. There are various constraints that constrain the search space.

The full encodings used during our experiments are available online.⁶ The version of ALPHA used for the experiments is available in the history of our open-source repository.⁷ ALPHA was used in its default configuration. The Java Virtual Machine was called with command-line parameters `-Xmx16g -jar Alpha-bundled.jar`. For comparison, CLINGO [20] was used in version 5.2.2.

HRP is originally an optimisation problem. Since ALPHA does not support optimisation statements yet, we had to discard them from the encoding. However, heuristic directives can be written in a way that optimal or near-optimal solutions are preferably found. For experiments with CLINGO, optimisation was therefore also switched off.⁸ The encoding for CLINGO does not contain domain-specific heuristics. To measure time and memory consumption, PYRUNLIM⁹ was used. A machine with an Intel® Xeon® CPU E5-2630 v2 @ 2.60GHz with two cores, 31.4G of memory and Ubuntu 16.04.5 LTS was employed to conduct the experiments. 20 instances of the HRP were generated, ranging from 5 persons owning 25 things to 100 persons owning 500 things. All instances used for these experiments use an empty legacy configuration, i.e. we consider a configuration problem and not a reconfiguration problem. Future experiments will also consider other classes of problem instances. The heuristic implemented in our ALPHA encoding manages to solve all these instances with zero backtracks.

Experimental results are shown in fig. 1. A striking feature of these plots is that CLINGO cannot solve the three largest instances from our benchmark set within available memory, while ALPHA can solve all instances by successful application of lazy grounding. Time consumption of ALPHA grows a bit faster than that of CLINGO, but generally compares well with it. It has to be kept in mind, however, that CLINGO manages to produce these results with domain-independent heuristics alone, while ALPHA (currently) needs domain-specific heuristics to compete in these terms. Adding domain-specific heuristics to the CLINGO encoding could possibly make it even more efficient, but that would also aggravate its memory consumption problem.

Turning our attention to the memory consumption shown in the plot on the right-hand side, we see the overwhelming benefits of the lazy-grounding approach. While ALPHA needs more memory than CLINGO until instance sizes of about 200 (which probably results from the fact that CLINGO has been optimized for over a decade now), its memory consumption grows much slower asymptotically. Within available memory much larger instances could be solved by ALPHA, while instances larger than 425 things are out of reach for CLINGO under the given encoding.

⁶ Encodings [house_alpha_2018-09-10b.asp](https://github.com/alpha-asp/Alpha/blob/domspec_heuristics/src/test/resources/DomainHeuristics/House/) and [house_clingo_2018-06-15c.asp](https://github.com/alpha-asp/Alpha/blob/domspec_heuristics/src/test/resources/DomainHeuristics/House/) as well as instances at https://github.com/alpha-asp/Alpha/blob/domspec_heuristics/src/test/resources/DomainHeuristics/House/

⁷ <https://github.com/alpha-asp/Alpha/commit/7229f1e32e4977ea0cf590a9487b32258f41fab5>

⁸ Optimisation was switched off in CLINGO using command-line argument `--opt-mode=ignore`.

⁹ <https://alviano.com/software/pyrunlim/>

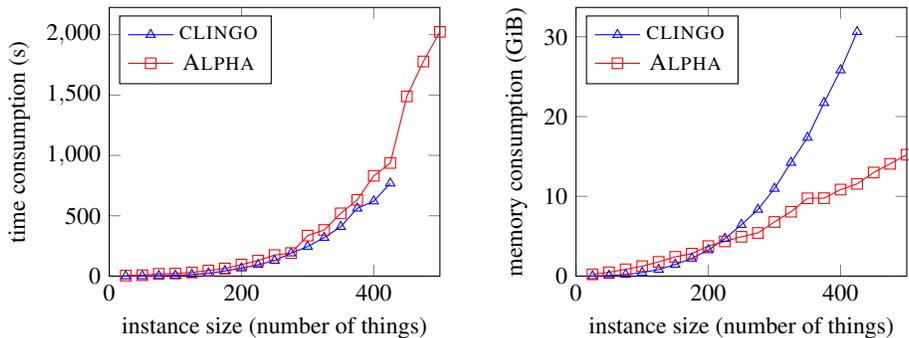


Fig. 1. Time and memory consumption for finding the first answer set for various HRP instances

7 Conclusions and Future Work

We have proposed a novel semantics for declarative domain-specific heuristics in ASP, demonstrated how to integrate them in a lazy-grounding ASP system and presented experimental results obtained with the lazy-grounding solver ALPHA. Our semantics differs from the previous state of the art by evaluating default negation with respect to incomplete information more naturally during solving. Benefits of this semantics for many problem domains have been made evident by use of examples. Our experimental results show that our approach is feasible, exhibiting encouraging time consumption and outstanding memory consumption behaviour.

The experiments reported in this paper are rather limited. In future experiments we plan to compare the solvers ALPHA, CLASP, and WASP with each other, each with and without domain-specific heuristics.

Further work is planned to extend syntax and semantics proposed here. For example, aggregates in heuristic conditions remain to be studied. Also, it could be worthwhile to adopt more ideas like `init` and `factor` modifiers from CLINGO for our semantics. It should also be examined how to support randomness and restarts, since such features are used by several real-world domain-specific heuristics. Since our approach has only been implemented in a lazy-grounding system so far, an adaption to ground-and-solve systems like CLINGO [20] or WASP [1] should be investigated in the future.

Acknowledgements

This work has been conducted in the scope of the research project *DynaCon* (FFG-PNr.: 861263), which is funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under the program “ICT of the Future” between 2017 and 2020 (see <https://iktderzukunft.at/en/> for more information).

This research was also supported by the Academy of Finland, project 251170, and by EU ECSEL Joint Undertaking under grant agreement no. 737459 (project Productive4.0).

We also thank Andreas Falkner for his comments on an earlier version of this paper.

References

1. Alviano, M., Dodaro, C., Faber, W., Leone, N., Ricca, F.: WASP: A Native ASP Solver Based on Constraint Learning. In: Cabalar, P., Son, T.C. (eds.) *Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 8148, pp. 54–66. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40564-8_6
2. Aschinger, M., Drescher, C., Friedrich, G., Gottlob, G., Jeavons, P., Ryabokon, A., Thorstensen, E.: Optimization Methods for the Partner Units Problem. In: *Proceedings of the 8th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. pp. 4–19. CPAIOR’11, Springer-Verlag, Berlin, Heidelberg (2011)
3. ASP Standardization Working Group: ASP-Core-2 Input Language Format (2012-12-13), <https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03b.pdf>
4. Baral, C.: *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press (2003)
5. Calimeri, F., Fuscà, D., Perri, S., Zangari, J.: I-DLV: the new intelligent grounder of DLV. *Intelligenza Artificiale* **11**(1), 5–20 (2017)
6. Calimeri, F., Gebser, M., Maratea, M., Ricca, F.: Design and results of the Fifth Answer Set Programming Competition. *Artificial Intelligence* **231**, 151–181 (2016). <https://doi.org/10.1016/j.artint.2015.09.008>
7. Calimeri, F., Ianni, G., Ricca, F.: The third open answer set programming competition. *Theory and Practice of Logic Programming* **14**(01), 117–135 (2014). <https://doi.org/10.1017/S1471068412000105>
8. Dal Palù, A., Dovier, A., Pontelli, E., Rossi, G.: GASP: Answer Set Programming with Lazy Grounding. *Fundamenta Informaticae* **96**(3), 297–322 (2009)
9. Dao-Tran, M., Eiter, T., Fink, M., Weidinger, G., Weinzierl, A.: OMiGA: An Open Minded Grounding On-The-Fly Answer Set Solver. In: Fariñas del Cerro, L., Herzig, A., Mengin, J. (eds.) *Logics in Artificial Intelligence*. Lecture Notes in Artificial Intelligence, vol. 7519, pp. 480–483. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33353-8_38
10. Dodaro, C., Gasteiger, P., Leone, N., Musitsch, B., Ricca, F., Schekotihin, K.: Combining Answer Set Programming and domain heuristics for solving hard industrial problems (Application Paper). *Theory and Practice of Logic Programming* **16**(5-6), 653–669 (2016). <https://doi.org/10.1017/S1471068416000284>
11. Eiter, T., Faber, W., Fink, M., Woltran, S.: Complexity results for answer set programming with bounded predicate arities and implications. *Annals of Mathematics and Artificial Intelligence* **51**(2), 123 (2008). <https://doi.org/10.1007/s10472-008-9086-5>
12. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: A logic programming approach to knowledge-state planning, II: the dlv^k system. *Artif. Intell.* **144**(1-2), 157–211 (2003)
13. Erdem, E., Gelfond, M., Leone, N.: Applications of answer set programming. *AI Magazine* **37**(3), 53–68 (2016), <http://www.aaai.org/ojs/index.php/aimagazine/article/view/2678>
14. Faber, W., Pfeifer, G., Leone, N.: Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence* **175**(1), 278–298 (2011). <https://doi.org/10.1016/j.artint.2010.04.002>
15. Falkner, A.A., Friedrich, G., Schekotihin, K., Taupe, R., Teppan, E.C.: Industrial applications of answer set programming. *KI* **32**(2-3), 165–176 (2018)
16. Ferraris, P.: Logic programs with propositional connectives and aggregates. *ACM Trans. Comput. Log.* **12**(4), 25:1–25:40 (2011)

17. Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T., Thiele, S.: Potassco User Guide version 2.1.0 (2017), <https://github.com/potassco/guide/releases/tag/v2.1.0>
18. Gebser, M., Kaminski, R., Kaufmann, B., Romero, J., Schaub, T.: Progress in clasp series 3. In: LPNMR. Lecture Notes in Computer Science, vol. 9345, pp. 368–383. Springer (2015)
19. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers (2012)
20. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Clingo = ASP + Control: Preliminary Report. In: Leuschel, M., Schrijvers, T. (eds.) Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP' 14). vol. arXiv:1405.3694v1 (2014)
21. Gebser, M., Kaminski, R., König, A., Schaub, T.: Advances in *gringo* series 3. In: LPNMR. Lecture Notes in Computer Science, vol. 6645, pp. 345–351. Springer (2011)
22. Gebser, M., Kaufmann, B., Otero, R., Romero, J., Schaub, T., Wanko, P.: Domain-specific Heuristics in Answer Set Programming. In: Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence. pp. 350–356. AAAI Press (2013)
23. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven Answer Set Solving: From Theory to Practice. Artificial Intelligence **187-188**, 52–89 (2012). <https://doi.org/10.1016/j.artint.2012.04.001>
24. Gebser, M., Maratea, M., Ricca, F.: The sixth answer set programming competition. J. Artif. Intell. Res. **60**, 41–95 (2017). <https://doi.org/10.1613/jair.5373>, <https://doi.org/10.1613/jair.5373>
25. Gebser, M., Ryabokon, A., Schenner, G.: Combining Heuristics for Configuration Problems Using Answer Set Programming. In: Calimeri, F., Ianni, G., Truszczyński, M. (eds.) Logic Programming and Nonmonotonic Reasoning. Lecture Notes in Artificial Intelligence, vol. 9345, pp. 384–397. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-23264-5_32
26. Gelfond, M., Kahl, Y.: Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach. Cambridge University Press, New York, NY, USA (2014)
27. Gelfond, M., Lifschitz, V.: The Stable Model Semantics For Logic Programming. In: Kowalski, R., Bowen, K. (eds.) Proceedings of the Fifth International Conference and Symposium of Logic Programming. pp. 1070–1080. MIT Press (1988)
28. Hoos, H., Lindauer, M.T., Schaub, T.: claspfolio 2: Advances in algorithm selection for answer set programming. TPLP **14**(4-5), 569–585 (2014)
29. Hotz, L., Felfernig, A., Stumptner, M., Ryabokon, A., Bagley, C., Wolter, K.: Chapter 6 - Configuration Knowledge Representation and Reasoning. In: Felfernig, A., Hotz, L., Bagley, C., Tiihonen, J. (eds.) Knowledge-Based Configuration, pp. 41–72. Morgan Kaufmann, Boston (2014). <https://doi.org/10.1016/B978-0-12-415817-7.00006-2>, <https://www.sciencedirect.com/science/article/pii/B9780124158177000062>
30. Lefèvre, C., Béatrix, C., Stéphan, I., Garcia, L.: ASPeRiX, a first-order forward chaining approach for answer set computing. Theory and Practice of Logic Programming **17**(3), 266–310 (2017). <https://doi.org/10.1017/S1471068416000569>
31. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transaction on Computational Logic **7**(3), 499–562 (2006). <https://doi.org/10.1145/1149114.1149117>
32. Leutgeb, L., Weinzierl, A.: Techniques for Efficient Lazy-Grounding ASP Solving. In: Seipel, D., Hanus, M., Abreu, S. (eds.) Declare 2017 - Conference on Declarative Programming. pp. 123–138. Technical Report, University of Würzburg (2017)

33. Lifschitz, V.: Thirteen Definitions of a Stable Model. In: Blass, A., Dershowitz, N., Reisig, W. (eds.) *Fields of Logic and Computation, Lecture Notes in Computer Science*, vol. 6300, pp. 488–503. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15025-8_24
34. Maratea, M., Pulina, L., Ricca, F.: The multi-engine ASP solver me-asp. In: *JELIA*. pp. 484–487 (2012)
35. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: *Proceedings of the 38th Design Automation Conference*. pp. 530–535. IEEE (2001)
36. Ostrowski, M., Schaub, T., Durzinsky, M., Marwan, W., Wagler, A.: Automatic network reconstruction using ASP. *CoRR* **abs/1107.5671** (2011), <http://arxiv.org/abs/1107.5671>
37. Ricca, F., Grasso, G., Alviano, M., Manna, M., Lio, V., Iiritano, S., Leone, N.: Team-building with answer set programming in the Gioia-Tauro seaport. *TPLP* **12**(3), 361–381 (2012)
38. Ryabokon, A.: Knowledge-based (Re)configuration of Complex Products and Services. Ph.D. thesis, Alpen-Adria-Universität Klagenfurt (2015), <http://netlibrary.aau.at/urn:nbn:at:at-ubk:1-26431>
39. Taupe, R., Weinzierl, A., Schenner, G.: Introducing Heuristics for Lazy-Grounding ASP Solving. In: *1st International Workshop on Practical Aspects of Answer Set Programming* (2017), <https://sites.google.com/site/paoasp2017/Taupe-et-al.pdf>
40. Weinzierl, A.: Blending Lazy-Grounding and CDNL Search for Answer-Set Solving. In: *Logic Programming and Nonmonotonic Reasoning*. pp. 191–204 (2017). https://doi.org/10.1007/978-3-319-61660-5_17