Dynamic Aggregates in Expressive ASP Heuristics for Configuration Problems

Richard Comploi-Taupe[1,*], Gerhard Friedrich[2,*] and Tilman Niestroj[2,*]

[1]*Siemens AG Österreich, Vienna, Austria*
[2]*Universität Klagenfurt, Klagenfurt, Austria*

Abstract

First-order logic has been applied successfully to real-world configuration problems through Answer Set Programming (ASP). To extend the application scope of ASP, lazy grounding and domain-specific heuristics were introduced. Domain-specific heuristics support the problem solver in selecting choices aiming at minimizing the search effort. Dynamic heuristics exploit the current state of the problem-solving process and assign priorities to choices. Depending on the domain, heuristics must be formulated which reason about the properties of sets of atoms. E.g., how many components are connected to a particular type of component, or what is the current sum/maximum/minimum of a physical quantity (power, voltage, current, etc.) of a particular subconfiguration? For expressing such queries, ASP offers aggregates. The semantics of these aggregates are defined w.r.t. a complete solution. However, in dynamic heuristics, the problem solver has to reason about partial solving states. In this paper, we extend heuristics in ASP with dynamic aggregates and show their implementation as well as effectiveness.

Keywords

knowledge-based configuration, answer set programming, heuristics

1. Introduction

Answer Set Programming (ASP) [1] is a declarative knowledge representation and reasoning framework based on first-order logic that has been applied successfully to a variety of industrial problems [2] such as configuration [3]. Current ASP solvers transform first-order descriptions of problem instances into propositional logic (called grounding) and apply a propositional problem solver (e.g., backtracking search) to generate solutions. However, applications manifested two issues with the ground-and-solve approach. The first issue is the so-called *grounding bottleneck*: Large problem instances cannot be grounded by modern grounders like grıngo [4] in acceptable time and space. The second issue is that, even if the problem can be grounded, computation of answer sets might take considerable time, as indicated by ASP competition reports [5].

Both issues were recently addressed. First, to overcome the grounding bottleneck, lazy grounding ASP systems interleave grounding and solving to instantiate and store only relevant parts of the ground program in memory. The second performance-related issue is tackled by modern solvers using various techniques, among which domain-specific heuristics play a central role.

The work in [6] extends existing approaches by (1) in-

troducing dynamic heuristics and (2) their exploitation in a lazy-grounding ASP system. The ASP system extended by this approach is Alpha [7], the most actively developed lazy-grounding system available. Dynamic heuristics allow reasoning about the current problem-solving state represented by a partial assignment of truth values to some (but not all) atoms of a logical specification.

This reasoning may require the application of aggregation. E.g., during the configuration process of electronic equipment, an effective heuristic for problem-solving can say: Given the current state of problem-solving, select the most power-hungry, currently unconnected electronic board, and connect this board to the rack with minimal power consumption (i.e., the rack for which the total power consumption of all boards currently connected is minimal).

However, state-of-the-art ASP aggregates are evaluated only w.r.t. a complete assignment of truth values, i.e., only if every atom (proposition) is true or false, so that their value cannot change during solving. For our rack configuration example, this semantics implies that the power consumption of a rack can only be determined if the assignment of boards to a rack is final. Consequently, ASP aggregates like sum cannot be employed in dynamic heuristics to reason about the current search state where board assignments to a rack are not completed.

In this paper, we introduce *dynamic aggregates*, which are computed w.r.t. the current state of problem-solving. Consequently, such aggregates can be exploited in dynamic heuristics to steer the reasoning process depending on the current state of problem-solving.

The paper is organized as follows. In Section 2, we give a brief introduction to ASP and lazy grounding. Section 3

provides a driving example and introduces dynamic aggregates informally. In Section 4, we present the syntax and semantics of dynamic aggregates. Section 5 shows the implementation and integration of dynamic aggregates using a query-driven approach. Finally, in Section 6, we present the results of our evaluation.

2. Answer Set Programming

Answer Set Programming (ASP) [1] is an approach to declarative programming. Instead of stating how to solve a problem, the programmer formulates the problem as a logic program specifying the search space and the properties of valid solutions. 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

ASP offers a rich input language, of which we introduce only the core concepts needed in this paper. For a comprehensive definition of ASP's syntax and semantics, we refer to [8].

Let $\langle \mathcal{V}, \mathcal{C}, \mathcal{F}, \mathcal{P} \rangle$ define a first-order language, where $\mathcal{V}$ is a set of variable symbols, $\mathcal{C}$ is a set of constant symbols, $\mathcal{F}$ is a set of function symbols, and $\mathcal{P}$ is a set of predicate symbols.

A classical *atom* is of the form $p(t_1, \dots, t_n)$, where $p \in \mathcal{P}$ is a predicate symbol and $t_1, \dots, t_n$ are *terms*. Each variable $v \in \mathcal{V}$ and each constant $c \in \mathcal{C}$ is a term. Furthermore, for $f \in \mathcal{F}$, $f(t_1, \dots, t_n)$ is a function term. ASP also allows built-in atoms, such as equality or comparison predicates, which take arithmetic terms as arguments, e.g., $X**2 > 1$ where $**$ is the power operator.

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

$$h \leftarrow b_1, \ \dots \ , b_m, \text{ not } b_{m+1}, \ \dots \ , \text{ not } b_n. \qquad \langle 1 \rangle$$

where h and $b_1, \dots, b_n$ are atoms and *not* is negation as failure (a.k.a. default negation), which refers to the absence of information, i.e., an atom is assumed to be false as long as it is not derived by some rule. A *literal* is either an atom a or its negation not a. Given a rule r of the form $\langle 1 \rangle$, head$(r) = \{h\}$ is called the *head* of r, and body$(r) = \{b_1, \dots, b_m, \text{ not } b_{m+1}, \dots, \text{ not } b_n\}$ is called the *body* of r. By body$^+(r) = \{b_1, \dots, b_m\}$ and body$^-(r) = \{b_{m+1}, \dots, b_n\}$ we denote the positive and negative atoms in the body of r, respectively. A rule r where head$(r) = \emptyset$, e.g., $\leftarrow$ b., is called *constraint*. A rule r where body$(r) = \emptyset$, e.g., h $\leftarrow$., is called *fact*. In facts the arrow can be omitted. A rule is ground if all its atoms are variable-free. A ground program comprises only ground rules.

2.2. Semantics

Given a program P, the *Herbrand universe* of P, denoted by U_P, consists of all integers and of all ground terms constructible from constant symbols and function symbols appearing in P. The *Herbrand base* of P, denoted by B_P, is the set of all ground classical atoms that can be built by combining predicates appearing in P with terms from U_P as arguments [8].

A substitution σ is a mapping from variables $\mathcal{V}$ to elements of the Herbrand universe U_P of a program P. Let O be a rule, an atom, or a literal, then by $O\sigma$ we denote a rule, atom, or literal obtained by replacing each variable $v \in$ vars(O) by $\sigma(v)$. The function vars maps any rule, atom, literal, or any other object containing variables to the set of variables it contains. For instance, vars$(a(X)) = \{X\}$ and for a rule $r_1 : a(X) \leftarrow b(X, Y).$, vars$(r_1) = \{X, Y\}$.

As usual, we assume rules to be *safe*, which is the case for a rule r if vars$(r) \subseteq \bigcup_{a \in \text{body}^+(r)}$ vars(a), e.g., all variables must occur in the positive atoms of the rule, which allows the grounding process to substitute them with constants.

The (ground) instantiation of a rule r equals $r\sigma$ for some substitution σ, which maps all variables in r to ground terms. The (ground) instantiation grd(P) of a program P is the set of all possible instantiations of the rules in P [8]. Function symbols may cause the Herbrand base and the full grounding of a program to be infinite. By restricted usage of function symbols, answer-set programs can be designed in a way that reasoning is decidable.

An *Herbrand interpretation* for a program P is a set of ground classical atoms $I \subseteq B_P$. A ground classical atom a is true w.r.t. an interpretation I, denoted $I \vDash a$, iff $a \in I$. A ground literal not a is true w.r.t. an interpretation I, denoted $I \vDash$ not a, iff $I \nvDash a$. A ground rule r is *satisfied* w.r.t. I, denoted $I \vDash r$, if its head atom is true w.r.t. I ($h \in$ head$(r) : I \vDash h$) whenever all body literals are true w.r.t. I ($\forall b \in$ body$(r) : I \vDash b$). An interpretation I is a model of P, denoted $I \vDash P$, if $I \vDash r$ for all rules $r \in$ grd(P).

Given a ground program P and an interpretation I, let P^I denote the transformed program obtained from P by deleting rules in which a body literal is false w.r.t. I: $P^I = \{r \mid r \in P, \forall b \in$ body$(r) : I \vDash b\}$.

An interpretation I of a program P is an *answer set* of P if it is a subset-minimal model of grd$(P)^I$, i.e., I is a model of grd$(P)^I$ and there exists no $I' \subsetneq I$ that is a model of grd$(P)^I$.

2.3. Notation

In this section, we introduce some notation that will be used later in the article.

An *assignment* A over B_P is a set of signed literals $\mathbf{T}\, a$, $\mathbf{F}\, a$, or $\mathbf{M}\, a$, where $\mathbf{T}\, a$ and $\mathbf{F}\, a$ express that an atom a

is true and false, respectively, and $\mathbf{M}$ a indicates that a "must-be-true". $\mathbf{M}$ means that an atom must eventually become true by derivation in a correct solution extending the current partial assignment, but no derivation has yet been found that would make the atom true. E.g., given constraint ← not b. we know that atom b must be true and has to eventually become true by derivation. Intuitively, $\mathbf{T}$ $b \in A$ means that b is true and justified, i.e., derived by a rule that fires under A, while $\mathbf{M}$ $b \in A$ only indicates that b is true but potentially not derived. Let $A_s = \{a \mid s\, a \in A\}$ for $s \in \{\mathbf{F}, \mathbf{M}, \mathbf{T}\}$ denote the set of atoms occurring with a specific sign in assignment A. We assume assignments to be *consistent*, i.e., no negative literal may also occur positively ($A_\mathbf{F} \cap (A_\mathbf{M} \cup A_\mathbf{T}) = \varnothing$), and every positive literal must also occur with must-be-true ($A_\mathbf{T} \subseteq A_\mathbf{M}$). The latter condition ensures that assignments are monotonically growing (w.r.t. set inclusion) in case an atom that was must-be-true becomes justified by a rule deriving it and hence changes to true.

An assignment A is *complete* if every atom in the Herbrand base is assigned true or false ($\forall a \in B_P : a \in A_\mathbf{F} \cup A_\mathbf{T}$). An assignment that is not complete is *partial*.

Many useful language constructs have been introduced to extend the basic language of ASP defined in Sections 2.1 and 2.2. We discuss such extensions only briefly and refer to [8] and [1] for full details.

A *cardinality atom* is of the form

$lb\ \{a_1 : l_{1_1}, \dots, l_{m_1}; \dots; a_n : l_{1_n}, \dots, l_{m_n}\}\ ub,$

where, for $1 \leq i \leq n$, $a_i : l_{1_i}, \dots, l_{m_i}$ represents a *conditional literal* in which a_i (the head of the conditional literal) is a classical atom and all l_{j_i} are literals, and lb and ub are integer terms indicating a lower and an upper bound, respectively. If one or both of the bounds are not given, their defaults are used, i.e., 0 for lb and ∞ for ub. A cardinality atom is satisfied if $lb \leq |C| \leq ub$ holds, where C is the set of head atoms in the cardinality atom that are satisfied together with their conditions (e.g., $l_{1_i}, \dots, l_{m_i}$ for a_i).

As an extension of cardinality atoms, ASP also supports aggregate atoms that apply aggregate functions like *count* or *sum* to sets of literals. An aggregate atom is satisfied if the value computed by the aggregate function respects the given bounds, e.g., $1 = \#sum\{1 : a; 2 : b\}$ is satisfied if a but not b is true.

2.4. Lazy Grounding

Lazy grounding is an approach that interleaves the solving and grounding phases, such that computations are guaranteed to yield all answer sets. The foundation for lazy grounding is known as the computation sequence [9]. A computation sequence $\mathbf{S} = \langle S_0, S_1, \dots, S_n \rangle$ is a sequence of partial assignments that is monotonically growing (w.r.t. set inclusion). Every element S_i of the sequence

represents the state of the computation at step i. The first element of the sequence is empty ($S_0 = \varnothing$), and every other element S_i contains the signed literals that can be derived from the preceding partial assignment S_{i-1} in the program P.

Since each element of a computation sequence is a partial assignment containing signed literals, and the sequence is monotonically growing, each S_i contains atoms assigned $\mathbf{T}$ that will remain true in all extensions of S_i, and atoms assigned $\mathbf{F}$ that will definitely remain false in all extensions of S_i.

Computation sequences require a normal logic program as input (i.e., rules of the form $\langle 1 \rangle$ without cardinality atoms and aggregate atoms, cf. [9, 10, 7]). Hence lazy grounding systems usually only accept normal logic programs or, in the case of Alpha, rewrite enhanced ASP constructs like aggregates into normal rules.

A rule r is said to be *applicable* in S_i if $\{\mathbf{T}\ a \mid a \in body^+(r)\} \subseteq S_i$ and $\{\mathbf{M}\ a \mid a \in body^-(r)\} \cap S_i = \varnothing$, i.e., if the positive body is satisfied and S_i does not contradict the negative body. For every applicable rule r in S_i without a negative body, the partial assignment S_i is extended to S_{i+1} by $\mathbf{T}$ head(r).

Based on the fact that the computation sequence only needs to know those ground rules that are applicable only those rules are grounded, whose positive body holds in the current partial assignment.

Each applicable rule r in S_i with a non-empty negative body constitutes an *active choice point*. Given a set of choice points for S_i the problem solver has to decide which rule to apply. Applying an applicable rule r has the consequence that S_i is extended to S_{i+1} by adding $\mathbf{T}$ head(r) and $\mathbf{F}$ a for all $a \in body^-(r)$, i.e., all atoms of the negative body are assumed to be false.

In the following example in S_0, Rule 1 is the only applicable rule. Consequently, $S_1 = \{\mathbf{M}\ x(1), \mathbf{T}\ x(1)\}$. In S_1 Rules 2 and 3 are applicable. If the solver decides to apply Rule 2 then $\mathbf{M}\ b(1)$, $\mathbf{T}\ b(1)$ and $\mathbf{F}\ c(1)$ are added to assignment S_2 and therefore Rule 3 is not applicable in S_2.

x(1) ← .	*%*	*Rule 1*
b(1) ← x(1), not c(1).	*% guessing b*	*Rule 2*
c(1) ← x(1), not b(1).	*% guessing c*	*Rule 3*

Deciding which rule to apply is based on heuristics which may be general, i.e., designed for every ASP program [11], or they may be domain-specific, e.g., designed for a specific problem [6].

3. Example

As an introductory example, consider the following ASP program. The idea is that for every number i $\in \{1, \dots, n\}$ the solver can decide either to assert b(i) or c(i). As an

example we set $n = 400$. Let Bs and Cs be all the $b/1$ and $c/1$ atoms in an answer set of the example program. We require that any answer set must fulfill the constraint $((\sum_{b(i)\in Bs} i) - (\sum_{c(i)\in Cs} i))^2 \leq 1$, e.g., the difference between these two sums must be at most 1. We call this problem the *Balanced Sum Problem (BSP)*. The example program comprises a guessing part and a part where solutions are checked. Moreover, we may specify initial facts like b(200) and b(201). In the worst case, 2^{398} guesses are possible. To avoid a high number of possible guesses, we can formulate heuristics that aid the solver in performing correct guesses such that backtracking is minimized.

x(1..400).   % *initializing values from 1 to 400.*
% **guessing**
b(X) ← x(X), not c(X).   % *guessing b*
c(X) ← x(X), not b(X).   % *guessing c*
% **initial imbalance**
b(200). b(201).
% **check solution**
sumB(Sum) ← Sum = #sum{Y : b(Y)}.
sumC(Sum) ← Sum = #sum{Y : c(Y)}.
% **constrain difference between sums**
← sumB(SB), sumC(SC), (SB − SC)∗∗2 > 1.
% **heuristics**
#heuristic b(X) :   % *b-heuristic*
    x(X), not c(X), S = #sum{Y : c(Y)},
    Weight = X, Level = S. [Weight@Level]
#heuristic c(X) :   % *c-heuristic*
    x(X), not b(X), S = #sum{Y : b(Y)},
    Weight = X, Level = S. [Weight@Level]

As an example, let us consider an instantiated version of a heuristic for the partial assignment $S_1 = \{\mathbf{M}\,b(200), \mathbf{T}\,b(200), \mathbf{M}\,b(201), \mathbf{T}\,b(201), \mathbf{M}\,x(1), \mathbf{T}\,x(1), \dots, \mathbf{M}\,x(400), \mathbf{T}\,x(400)\}$, i.e., the partial assignment comprising all initial facts which are true. For x(400) an instance of the c-heuristic (including the evaluation of the aggregate) is #heuristic c(400) : x(400), not b(400), $401 = \#sum\{200, 201\}$, [400@401].

Heuristic directives assign a weight and a level to a rule which derives an atom. In this instantiated heuristic directive, the weight is 400, and the level is 401. All other instantiated c-heuristics and b-heuristics have either a lower level or lower weights in case of the same level. For performing choices, guesses are preferred with higher levels, and higher weights are prioritized among guesses with the same level. Consequently, the solver will apply a rule which asserts c(400).

The novel concept of this paper is that aggregates in heuristic directives like #sum are evaluated w.r.t. the current assignment. For the partial assignment S_1, the aggregate #sum{Y : b(Y)} in the c-heuristic is evaluated as #sum{200, 201} since S_1 contains the atoms $\mathbf{T}\,b(200)$ and $\mathbf{T}\,b(201)$. Applying the aggregate function #sum derives 401. Note, in the partial assignment S_1, the c-heuristic is

not applicable if the #sum aggregate is evaluated under the standard declarative semantics of ASP. This semantics assumes that the truth assignments for the b/1 atoms are fixed.

By adding the shown heuristic directives to the example program, wrong choices, which lead to backtracks, can be avoided for the depicted problem instance. The following section will present the syntax and semantics of heuristics that employ dynamic aggregates.

4. Syntax and semantics

In [6] domain-specific heuristics for answer set programming were proposed which allow to reason about the current state of the problem-solving process. This state is reflected by the latest partial assignment. Consequently, heuristic directives are evaluated w.r.t. this assignment. However, in the declarative semantics of ASP the truth value of aggregates as presented in the example (e.g., S = #sum{Y : b(Y)}) can only be determined w.r.t. a fixed set of truth assignments for atoms. In the declarative semantics of ASP assigning a truth value to S = #sum{Y : b(Y)} implies that the set of b/1 atoms which are assigned to true is fixed, i.e., rules must not be applied which assert additional b/1 atoms to true.

However, in the spirit of [6] we propose to evaluate aggregates w.r.t. the latest partial assignment S_i to evaluate heuristic directives for determining the choice, i.e., which rule to apply to compute the next partial assignment.

Definition 1 (Heuristic Directive). *A* heuristic directive *is of the form* $\langle 2 \rangle$, *where* a_i $(0 \leq i \leq n)$ *are atoms and* w *and* l *are integer terms.*

$$\#heuristic\ a_0 : a_1, \dots, a_k,$$
$$not\ a_{k+1}, \dots, not\ a_n.[w@l] \quad \langle 2 \rangle$$

The heuristics' head is given by a_0 *and its condition by* $\{a_1, \dots, a_k, not\ a_{k+1}, \dots, not\ a_n\}$.

We call an atom in a heuristic directive a *heuristic atom*. We now describe our semantics, beginning with the condition under which a heuristic atom is satisfied.

Definition 2 (Satisfying a Heuristic Atom). *Given a ground heuristic atom* a *and a partial assignment* A, a *is satisfied w.r.t.* A *iff* $a \in A_{\mathbf{T}}$, *i.e., atom* a *is assigned to true.*

The *head* of a heuristic directive d of the form $\langle 2 \rangle$ is denoted by head(d) = a_0, its *weight* by weight(d) = w if given, else 0, and its *level* by level(d) = l if given, else 0. The *(heuristic) condition* of a heuristic directive d is denoted by cond(d) := $\{a_1, \dots, a_k, not\ a_{k+1}, \dots, not\ a_n\}$, the *positive condition* is cond$^+(d)$:= $\{a_1, \dots, a_k\}$ and the *negative condition* is cond$^-(d)$:= $\{a_{k+1}, \dots, a_n\}$.

Whether a heuristic directive is satisfied depends on whether the atoms occurring in the directive are satisfied.

Definition 3 (Satisfying a Heuristic Directive).
Given a ground heuristic directive d and a partial assignment A, $\mathrm{cond}(d)$ *is* satisfied *w.r.t. A iff: every $a \in \mathrm{cond}^+(d)$ is satisfied and no $a \in \mathrm{cond}^-(d)$ is satisfied.*

Intuitively, a heuristic condition is satisfied iff its positive part is fully satisfied and none of its default-negated literals is contradicted.

Definition 4 (Applicability of a Heuristic Directive).
A ground heuristic directive d is applicable *w.r.t. a partial assignment A and a ground program P iff:* $\mathrm{cond}(d)$ *is satisfied, $\exists r \in P$ s.t. $\mathrm{head}(r) = \mathrm{head}(d)$ and $\{\mathbf{T}\, a \mid a \in \mathrm{body}^+(r)\} \subseteq A$ and $\{\mathbf{M}\, a \mid a \in \mathrm{body}^-(r)\} \cap A = \emptyset$, and $\mathrm{head}(d) \notin (A_\mathbf{T} \cup A_\mathbf{F})$.*

Intuitively, a heuristic directive is applicable iff its condition is satisfied, there exists a currently applicable rule that can derive the atom in the heuristic directive's head, and the atom in its head is assigned neither $\mathbf{T}$ nor $\mathbf{F}$. If the atom in the head is assigned $\mathbf{M}$, the heuristic directive is still applicable, because any atom with the non-final truth value $\mathbf{M}$ must be either $\mathbf{T}$ or $\mathbf{F}$ in any answer set.

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

Definition 5 (Heuristics Eligible for Choice).
Given a set D of applicable ground heuristic directives, the subset eligible for immediate choice *is defined as* $\mathrm{maxpriority}(D)$ *in two steps:*

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

After choosing a heuristic using maxpriority, a solver makes a decision on the directive's head atom. Other solving procedures, e.g., deterministic propagation, are unaffected by processing heuristics. In case no heuristic directive is applicable or multiple directives have the same maxpriority the solver's default heuristic (e.g., VSIDS) makes a choice as usual.

Aggregate atoms may be employed in the condition of a heuristics directive. An *aggregate atom* is of the form

$$s_1 \prec_1 \alpha \{\mathbf{t} : l_{1_1}, \dots, l_{m_1}; \dots;$$
$$\mathbf{t} : l_{1_n}, \dots, l_{m_n}\} \prec_2 s_2$$

where $\mathbf{t}$ corresponds to a variable, an integer, or a ground atom. We call $\mathbf{t}$ an aggregate term. α refers to some aggregate function that is applied to the multiset of aggregate terms $\mathbf{t}$ that remain after evaluating the condition

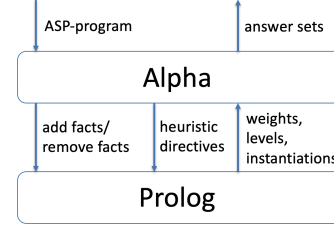**Figure 1:** Integration of Alpha and Prolog

$l_{1_i}, \dots, l_{m_i}$. The aggregate terms are treated as members of a mulitset. Duplicates are allowed.[1]

The result of applying α is exploited to evaluate the comparison condition expressed by $s_1 \prec_1$ and $\prec_2 s_2$. These conditions may be omitted. s_1, s_2 are terms, e.g., numbers or variables. $s_1 \prec_1$ and $\prec_2 s_2$ are called guards. For the guard operator $\prec$ comparison operators such as $=, \neq, \leq, \geq, <, >$ may be employed.

If in $\mathbf{t} : l_{1_i}, \dots, l_{m_i}$ of an aggregate atom the term $\mathbf{t}$ is a variable, then this variable must be safe. This variable is safe if it is contained in the condition or it is a global variable. A variable in a heuristic directive d is global if it appears in a classical atom in $\mathrm{cond}^+(d)$ or in a guard of an aggregate atom of d where $\prec$ corresponds to $=$.

We allow aggregate functions α like #count (the number of aggregate terms) or #sum (sum of aggregate terms).

An aggregate atom is satisfied if the value computed by the aggregate function respects the given bounds, e.g., $1 = \#\mathrm{sum}\{1 : \mathrm{a}; 1 : \mathrm{b}\}$ is satisfied if either a or b is true. Let us assume that the facts a(1). a(2). b(5). are given. Evaluating the aggregate atom X = $\#\mathrm{sum}\{Y : \mathrm{a}(Y); Y : \mathrm{b}(Y)\}$ will bind 8 to variable X.

5. Integration into a lazy-grounding ASP solver

In search for answer sets, Alpha applies heuristics to select an active choice point. In contrast to [6], the heuristic directives are transformed to Prolog queries and evaluated by a Prolog interpreter. We have chosen this approach to implement the efficient evaluation of dynamic aggregates in heuristic directives.

Figure 1 shows the integration of Alpha with Prolog.

Query-driven heuristics are employed by Alpha if the -uqh flag is set. The heuristic directives are removed from the input program and translated into internal data structures. These data structures comprise all the necessary

---

[1]Note that this semantics differs from the ASP semantics of aggregates employed in rules. First, for our prototypical system $\mathbf{t}$ is a single term and not a tuple of terms for simplicity reasons. Second, we allow a multiset of aggregate terms instead of a set, i.e., we do not remove duplicates. Sets and multisets can be easily implemented. However, the removal of duplicates may introduce additional computational costs.

information for evaluating the heuristic directives, such as their head atom, variables, atoms occurring inside the heuristic, and crucially, their respective Prolog query. Thus, the heuristic directives are separately stored from the program and are all evaluated whenever a choice is made.

As an example, the following heuristic directive

```
#heuristic c(X) :
    x(X), not b(X), S = #sum{Y : b(Y)},
    W = S * 10 + X. [W@1]
```
is translated to the following Prolog query:
```
x(X), \+ b(X),
    aggregate_all(sum(Y), b(Y), _0), S is _0,
    WEIGHT is S * 10 + X, LEVEL is 1, \+ c(X).
```

The Prolog predicate `aggregate_all(+Template, :Goal, −Result)` aggregates bindings in `Goal` according to `Template`. Possible template values comprise the aggregate functions, `count`, `sum(X)`, `max(X)`, and `min(X)`. The variable in `sum(X)`, `max(X)`, and `min(X)` corresponds to the variable serving as aggregate term and is instantiated by querying `Goal` which contains this variable. The result is bound to an anonymous variable (`_0` in our example) and exploited in the aggregates' guards. Any negated atom is preceded by the operator `\+`, equivalent to not for our purposes. Finally, the negated head atom of the heuristic directive is added to the query to exclude already assigned head atoms.

During problem-solving, Alpha synchronizes the assignments with the database of the Prolog system. Every atom assigned as true by Alpha is inserted in the Prolog database. If such atoms are removed from the assignment, the corresponding facts are retracted from the Prolog database. Atoms assigned to false or must-be-true by Alpha are not considered since the heuristic directives are evaluated on atoms assigned to true.

The current implementation of Alpha sources and binaries which implement query-driven heuristics can be found on https://github.com/tilmanni/Alpha/tree/domspec_heuristics_extended_prolog.

6. Evaluation

We tested our approach to declarative domain-specific query-driven heuristics by creating heuristics for two example domains and applying the extended Alpha system. The two concrete domains under investigation were the Partner Units Problem (PUP) and the Balanced Sum Problem (BSP) introduced in Section 3.

These two problems are abstracted variants of typical configuration (sub)problems experienced in more than 25 years of applying AI technology in the automated configuration of electronic systems [3]. To put ASP systems under stress, we used problem encodings and instances of

varying sizes, where the larger instances were challenging to ground and/or to solve. More precisely, traditional grounders excessively consumed space or time when grounding these instances, and/or solving was infeasible without domain-specific heuristics.

6.1. Experimental setup

Encodings (including heuristics) and instances used for our experiments are available online.[2]

Alpha was used without justification analysis [12] (command-line argument `-dj`) and without support for negative integers in aggregates (`-dni`). Apart from that, Alpha was used in its default configuration. The JVM running Alpha was called with command-line parameters `-Xms1G -Xmx24G`, thus initially allocating 1 GiB for Java's heap and setting the maximum heap size to 24 GiB. The Prolog interpreter swı-prolog[3] [13] version 9.0.4 was integrated with Alpha via jpl.[4] For comparison, clıngo[5] [14] was used in version 5.6.2.

Each of the machines used to run the experiments ran Ubuntu 22.04.2 LTS Linux and was equipped with two Intel® Xeon® E5-2650 v4 @ 2.20GHz CPUs with 12 cores. Hyperthreading was disabled and the maximum CPU frequency was set to 2.90GHz. Scheduling of benchmarks was done with slurm[6] version 21.08.5. runsolver[7] v3.4.1 was used to limit time consumption to 10 minutes per instance and memory to 32 GiB. Care was taken to avoid side effects between CPUs, e.g., by requesting exclusive access to an entire machine for each benchmark.

All solvers were configured to search for the first answer set of each problem instance. Finding one or only a few solutions is often sufficient in industrial use cases since solving large instances can be challenging [3]. Therefore, the domain-specific heuristics used in the experiments are designed to help the solver find one answer set that is "good enough", even though it may not be optimal.

6.2. Case Study 1: The Partner Units Problem (PUP)

The Partner Units Problem (PUP) [15] is an abstracted version of industrial configuration problems. In particular, PUP deals with configuring parts of railway safety systems. This problem is a benchmark problem for ASP systems since its challenges for grounding and solving.

---

[2]https://github.com/tilmanni/Alpha/tree/domspec_heuristics_extended_prolog/Evaluation
[3]https://www.swi-prolog.org/
[4]https://jpl7.org/
[5]https://potassco.org/clingo/
[6]https://slurm.schedmd.com/
[7]https://github.com/utpalbora/runsolver

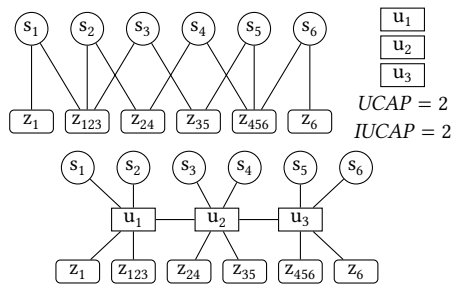**Figure 2:** Sample PUP instance and one of its solutions [6]

Definition 6 (PUP). *The input to the* (PUP) *is given by a set of units U and a bipartite graph $G = (S, Z, E)$ (also called the input graph), where S is a set of sensors, Z is a set of zones, and E is a relation between S and Z.*

The task is to find a partition of vertices $v \in S \cup Z$ into bags $u_i \in U$ such that for each bag the following requirements hold: (1) the bag contains at most UCAP vertices from S and at most UCAP vertices from Z; and (2) the bag has at most IUCAP adjacent bags, where the bags u_1 and u_2 are adjacent whenever $v_i \in u_1$ and $v_j \in u_2$ for some $(v_i, v_j) \in E$.

We say a unit u_i is connected to a sensor/zone iff the sensor/zone is in u_i. Two units are connected iff they are adjacent. Connections correspond to physical connections in an assembled configuration.

Figure 2 shows an example of a PUP instance. The bipartite graph comprises six sensors and six zones. Each of the three units can be adjacent to at most two other units, and each unit can contain at most two sensors and two zones. Connections of sensors, zones, and units that satisfy all PUP requirements are presented in Figure 2.

Encodings and instances. To show the application and effectiveness of query-driven heuristics, we focus on the PUP instances employed in the ASP competition [5]. Domain-specific heuristics allow exploiting knowledge about properties of classes of problem instances. We concentrate on the double and double-variant classes of PUP instances. For these instances, domain-specific heuristics were formulated.

Figure 3 shows the basic structure of the double instances. There are two rows of rooms connected by doors. Each room corresponds to a zone, and each door represents a sensor. For each room and the doors of this room, there is an edge in the bipartite graph G, i.e., the zone and its doors are connected through an edge. The double instances' sizes vary depending on the number of columns of rooms. The structure depicted in Figure 3 shows three columns of rooms. The bipartite graphs G of the double-variant instances comprise the nodes and edges of the double instances. However, each dotted rectangle represents an additional zone (i.e., the dotted rectangle clusters rooms). Each door (i.e., a sensor) next
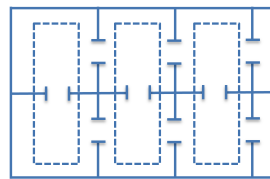
to a dotted rectangle (i.e., a zone) is connected by an edge in G. Note that there is no edge between a door surrounded by a rectangle and the zone corresponding to this rectangle.

Figure 3: Double and double-variant instances

Heuristics. The double PUP instances can be solved efficiently without backtracking by formulating the following heuristic directives, which employ dynamic aggregates.

```
#heuristic assigned_sensor_unit(S, U) :
  assignable_sensor_unit(S, U),
  not sensor_blocked_on_unit(S, U),
  Deg_sensor_dyn =
    #count{Z : zone2sensor(Z, S),
           assigned_zone_unit(Z, _)},
  Forbidden_placement_total =
    #max{N : num_forbidden_places_of_sensors(S, N)},
  Assigned_sensors_unit =
    #count{SN : assigned_sensor_unit(SN, U)},
  Direct_con_zones =
    #count{Z : assigned_zone_unit(Z, U),
           zone2sensor(Z, S)},
  W =
  Deg_sensor_dyn * 10000+
  Forbidden_placement_total * 1000+
  Assigned_sensors_unit * 100+
  Direct_con_zones * 10.[W@0]
```

The atom assignable_sensor_unit(S, U) is true if a sensor is ready to be assigned, i.e., if a sensor is connected to a zone in the input graph and this zone is connected to a unit. The atom sensor_blocked_on_unit(S, U) is true if sensor S cannot be connected to unit U. The variable Deg_sensor_dyn is assigned to the number of zones connected to sensor S in the input graph and which are already assigned to a unit. The atom zone2sensor(Z, S) encodes the edges of the input graph (i.e., connections between zones and sensors). Values of the variable Deg_sensor_dyn express the number of constraints put on placing sensor S. We prefer connecting sensors to units with a higher number of constraints. The atom num_forbidden_places_of_sens(S, N) represents the number of connections to units that are not possible for S for a given set of connections between sensors, zones, and units (e.g., the configuration in a specific

(a) Solver configurations, with numbers of solved instances

(b) Number of guesses

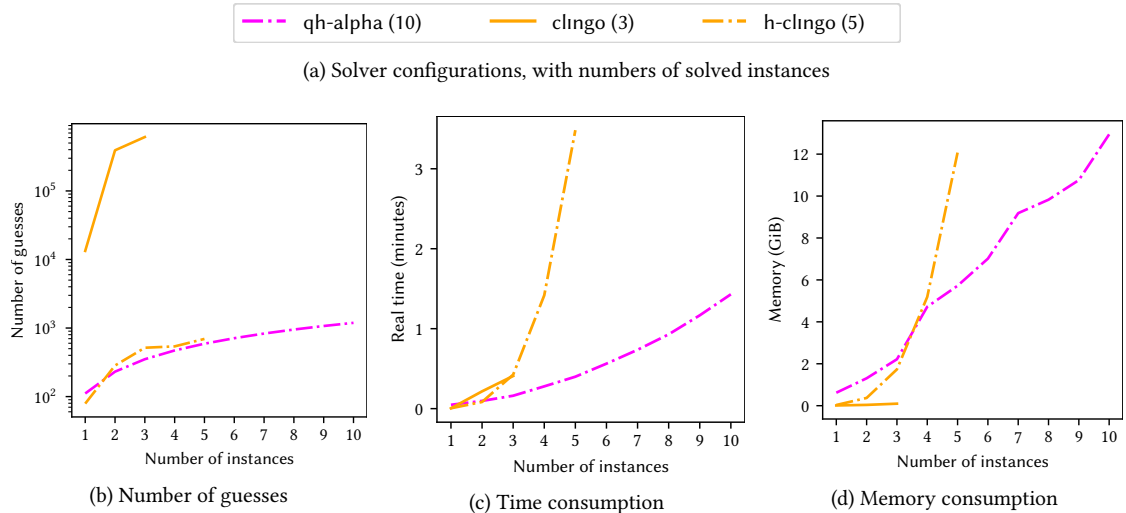(c) Time consumption

(d) Memory consumption

Figure 4: Resource consumption for solving each PUP Double instance

partial assignment). Rules compute different numbers depending on the connections. We prefer connecting sensors to units with a higher number of forbidden places (i.e., connections). The variable Assigned_sensors_unit counts the number of sensors connected to unit U. The variable Direct_con_zones counts the number of zones that are connected to unit U and which are connected to sensor S in the input graph. All these numbers are added with different weights resulting in the final weight of the heuristic directive expressing the priority to connect S and U. The design of the heuristic directive follows the principle of preferring assignments of connections that are most constrained in the spirit of heuristics of heuristics for constraint satisfaction problems (CSPs) such as "fail-first" or "degree" [16].

The second heuristic for assigning zones to units in double PUP instances can be formulated shorter than the presented one. We prefer assignments of zones to units U, where the number of connected zones to U is high, and the number of possible connections for sensors to U and its adjacent units is large.

Results. Figure 4 shows performance data for experiments with the double PUP instances. Cactus plots were created in the usual way. In Figure 4c, the x-axis gives the number of instances solved within real (i.e., wall-clock) time, given on the y-axis. Similarly, Figure 4b shows the number of guesses needed and Figure 4d shows the memory consumed to solve the instances. In all plots, data points are sorted by y-values. Figure 4a contains a legend with all solver configurations. The number of instances solved by each system is shown next to its name (in parentheses).

One curve was drawn for each solver configuration: Alpha with query-driven evaluation of domain-specific heuristics (qh-alpha), and clingo with (h-clingo) and without domain-specific heuristics.

Figure 5 shows the results for the double-variant instances in exactly the same way.

Alpha was used with encodings and heuristics designed to achieve a good performance as described above. clingo was used with the "new" encoding from the Fifth ASP competition[8] [17]. h-clingo used the domain-specific heuristics devised in previous work [6]. Both systems used the same sets of problem instances, which consisted of 10 instances of the "double" class (with a number of units ranging between 20 and 200), and 6 instances of the "double-variants" class (with 30–180 units).

Substantial differences can be observed. The curves for qh-alpha reach farthest to the right, meaning that Alpha with query-driven heuristics solved the highest number of instances (all 10 double, 5 of 6 double-variants). clingo needed more time and thus solved fewer instances. Apparently, the domain-specific heuristics used with h-clingo were not useful for solving the double-variants instances.

6.3. Case Study 2: The Balanced Sum Problem (BSP)

The second evaluation case deals with the BSP. In configuring, sub-problems arise where quantities such as power consumption should be equally distributed.

(b) Number of guesses

(c) Time consumption

(d) Memory consumption
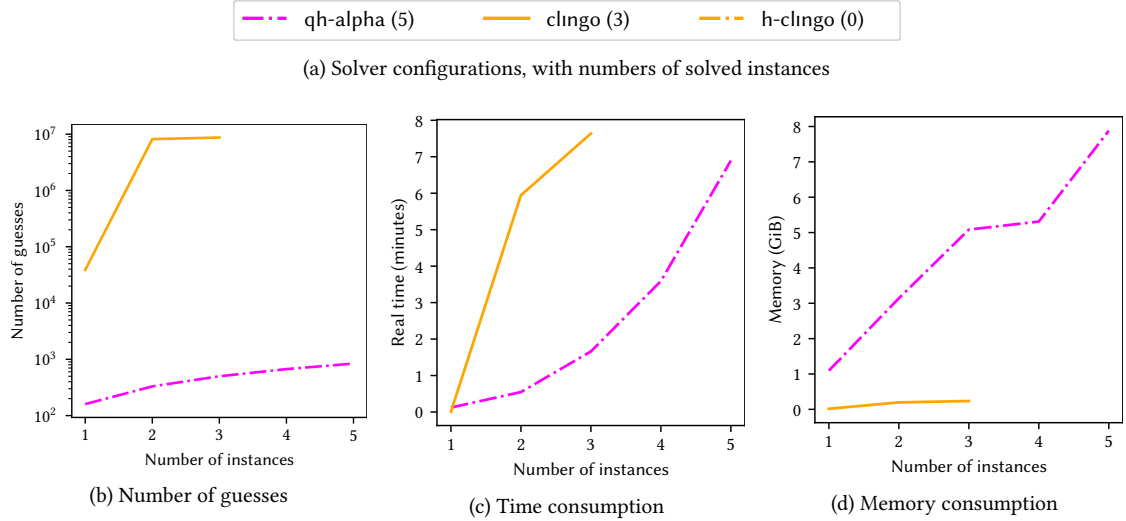
Figure 5: Resource consumption for solving each PUP DoubleV instance

(a) Solver configurations, with numbers of solved instances

(b) Number of guesses

(c) Time consumption
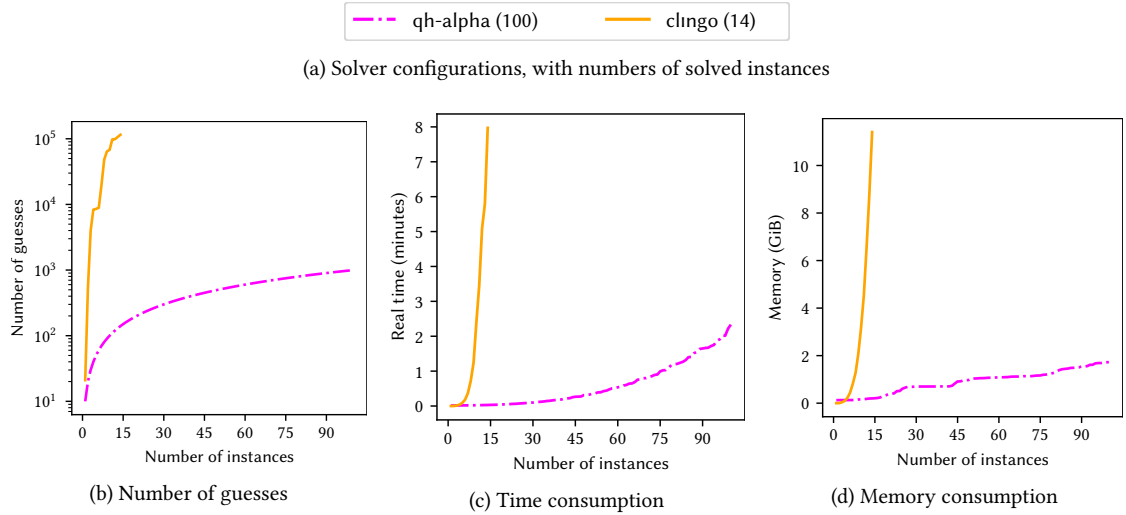
(d) Memory consumption

Figure 6: Resource consumption for solving each BSP instance

Encodings and instances. As the encoding of the problem, we use the ASP code introduced in Section 3. To strain the problem-solving, we increment the number of x/1 atoms and adapt the constant in the rules for sumB/1 and sumC/1.

Results. Results obtained for BSP are shown in Figure 6, which was generated in the same way as for PUP (cf. Section 6.2). clıngo was used with the encoding presented in Section 3, while Alpha used an alternative representation of the sum constraints that the lazy-grounding system could evaluate more efficiently.

Since heuristics in the non-dynamic semantics are not known for this problem, clıngo was only used without domain-specific heuristics. A hundred instances with $n \in \{10, 20, \ldots, 990, 1000\}$ were used for the experiments.

In the BSP experiments, qh-alpha greatly outperformed clıngo, showing the benefits of domain-specific heuristics evaluated in a query-driven way within a lazy-grounding ASP solver. While qh-alpha solved all 100 instances within at most 2.33 minutes per instance, clıngo reaches the grounding bottleneck very quickly and is not able to solve instances larger than $n = 140$.

7. Conclusions and future work

Dynamic heuristics are an effective means for formulating domain-specific heuristics to speed up problemsolving. We have introduced dynamic aggregates, allowing us to reason about the current problem-solving state. E.g., we can reason about second-order properties of this state, such as the number of atoms with specific properties or summing quantities over sets of atoms or computing the maximum/minimum of such quantities. We have provided the prototypical implementation qh-alpha by integrating Prolog with the lazy-grounding system Alpha. This system was evaluated on two problems related to configuration, i.e., the double and double variant cases of the well-known Partner Units Problem, and the Balanced Sum Problem. However, dynamic aggregates are a general concept that knowledge engineers can apply to other problem domains. The evaluation shows that dynamic aggregates employed in domain-specific heuristics can considerably improve solving performance.

In future work, we plan to develop further the prototypical implementation to incorporate standard semantics of aggregate elements and terms. Additionally, exploring incorporating sign set semantics in query-driven heuristics would further enhance expressiveness. Efforts should also be made to utilize query-driven and regular domainspecific heuristics concurrently. Finally, we recommend evaluating the feasibility of applying the query-driven approach to specific aggregates in normal rules as an alternative to resource-intensive normalization.

Acknowledgments

References

[1] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, S. Thiele, P. Wanko, Potassco guide version 2.2.0, 2019. URL: https://github.com/potassco/guide/releases/tag/v2.2.0.

[2] A. A. Falkner, G. Friedrich, K. Schekotihin, R. Taupe, E. C. Teppan, Industrial applications of answer set programming, Künstliche Intell. 32 (2018) 165–176.

[3] A. A. Falkner, G. Friedrich, A. Haselböck, G. Schenner, H. Schreiner, Twenty-five years of successful application of constraint technologies at Siemens, AI Magazine 37 (2016) 67–80.

[4] M. Gebser, R. Kaminski, A. König, T. Schaub, Advances in *gringo* series 3, in: LPNMR, volume 6645 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 345–351.

[5] M. Gebser, M. Maratea, F. Ricca, The seventh answer set programming competition: Design and results, Theory Pract. Log. Program. 20 (2020) 176–204.

[6] R. Comploi-Taupe, G. Friedrich, K. Schekotihin, A. Weinzierl, Domain-specific heuristics in answer set programming: A declarative non-monotonic approach, J. Artif. Intell. Res. 76 (2023) 59–114.

[7] A. Weinzierl, Blending lazy-grounding and CDNL search for answer-set solving, in: LPNMR, volume 10377 of *Lecture Notes in Computer Science*, Springer, 2017, pp. 191–204.

[8] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, T. Schaub, ASP-Core-2 input language format, Theory Pract. Log. Program. 20 (2020) 294–309.

[9] L. Liu, E. Pontelli, T. C. Son, M. Truszczynski, Logic programs with abstract constraint atoms: The role of computations, in: ICLP, volume 4670 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 286–301.

[10] C. Lefèvre, C. Béatrix, I. Stéphan, L. Garcia, ASPeRiX, a first-order forward chaining approach for answer set computing, Theory Pract. Log. Program. 17 (2017) 266–310.

[11] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an efficient SAT solver, in: DAC, ACM, 2001, pp. 530–535.

[12] B. Bogaerts, A. Weinzierl, Exploiting justifications for lazy grounding of answer set programs, in: IJCAI, ijcai.org, 2018, pp. 1737–1745.

[13] J. Wielemaker, T. Schrijvers, M. Triska, T. Lager, Swi-prolog, Theory Pract. Log. Program. 12 (2012) 67–96. doi:10.1017/S1471068411000494.

[14] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot ASP solving with clingo, Theory Pract. Log. Program. 19 (2019) 27–82.

[15] E. C. Teppan, Solving the partner units configuration problem with heuristic constraint answer set programming, in: Configuration Workshop, 2016, pp. 61–68.

[16] S. J. Russell, P. Norvig, Artificial Intelligence – A Modern Approach, Fourth Edition, Pearson Education, 2022.

[17] F. Calimeri, M. Gebser, M. Maratea, F. Ricca, Design and results of the fifth answer set programming competition, Artif. Intell. 231 (2016) 151–181. doi:10.1016/j.artint.2015.09.008.