TLP **14** (4–5): 587–601, 2014.
doi:10.1017/S1471068414000222

Vicious Circle Principle and Logic Programs with Aggregates

MICHAEL GELFOND and YUANLIN ZHANG
Texas Tech University, Lubbock, Texas 79414, USA
(*e-mail:* {michael.gelfond, y.zhang}@ttu.edu)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

The paper presents a knowledge representation language $\mathscr{A}log$ which extends ASP with aggregates. The goal is to have a language based on simple syntax and clear intuitive and mathematical semantics. We give some properties of $\mathscr{A}log$, an algorithm for computing its answer sets, and comparison with other approaches.

KEYWORDS: Aggregates, Answer Set Programming

1 Introduction

The development of answer set semantics for logic programs (Gelfond and Lifschitz 1988; Gelfond and Lifschitz 1991) led to the creation of powerful knowledge representation language, Answer Set Prolog (ASP), capable of representing recursive definitions, defaults, effects of actions and other important phenomena of natural language. The design of algorithms for computing answer sets and their efficient implementations in systems called *ASP solvers* (Niemela *et al.* 2002; Leone *et al.* 2006; Gebser *et al.* 2007) allowed the language to become a powerful tool for building non-trivial knowledge intensive applications (Brewka *et al.* 2011; Erdem *et al.* 2012). There are a number of extensions of the ASP which also contributed to this success. This paper is about one such extension – *logic programs with aggregates.* By *aggregates we mean functions defined on sets of objects of the domain.* (For simplicity of exposition we limit our attention to aggregates defined on finite sets.) Here is a typical example.

Example 1 (*Classes That Need Teaching Assistants*)
Suppose that we have a complete list of students enrolled in a class c that is represented by the following collection of atoms:

```
enrolled(c,mike).
enrolled(c,john).
...
```

Suppose also that we would like to define a new relation *need_ta*(C) that holds iff the class C needs a teaching assistant. In this particular school *need_ta*(C) is true iff the number of students enrolled in the class is greater than 20. The definition can be given by a simple rule in the language of logic programs with aggregates:

$$need\_ta(C) \leftarrow card\{X : enrolled(C, X)\} > 20$$

where *card* stands for the cardinality function. Let us call the resulting program P_0.

The program is simple, has a clear intuitive meaning, and can be run on some of the existing ASP solvers. However, the situation is more complex than that. Unfortunately, currently there is no *the* language of logic programs with aggregates. Instead there is a comparatively large collection of such languages with different syntax and, even more importantly, different semantics (Pelov *et al.* 2007; Niemela *et al.* 2002; Son and Pontelli 2007; Faber *et al.* 2011; Gelfond 2002; Kemp and Stuckey 1991). As an illustration consider the following example:

Example 2
Let P_1 consist of the following rule:

$$p(a) \leftarrow card\{X : p(X)\} = 1.$$

Even for this seemingly simple program, there are different opinions about its meaning. According to (Faber *et al.* 2011) the program has one answer set $A = \{\}$; according to (Gelfond 2002; Kemp and Stuckey 1991) it has two answer sets: $A_1 = \{\}$ and $A_2 = \{p(a)\}$.

In our judgment this and other similar "clashes of intuition" cause a serious impediment to the use of aggregates for knowledge representation and reasoning. In this paper we aim at addressing this problem by suggesting yet another logic programming language with aggregates, called $\mathscr{A}log$, which is based on the following design principles:

- the language should have a simple syntax and intuitive semantics based on understandable informal principles, and
- the informal semantics should have clear and elegant mathematics associated with it.

In our opinion existing extensions of ASP by aggregates often do not have clear intuitive principles underlying the semantics of the new constructs. Moreover, some of these languages violate such original foundational principles of ASP as the rationality principle. The problem is compounded by the fact that some of the semantics of aggregates use rather non-trivial mathematical constructions which makes it difficult to understand and explain their intuitive meaning.

The semantics of $\mathscr{A}log$ is based on *Vicious Circle Principle* (VCP): *no object or property can be introduced by the definition referring to the totality of objects satisfying this property*. According to Feferman (Feferman 2002) the principle was first formulated by Poincare (Poincare 1906) in his analysis of paradoxes of set theory. Similar ideas were already successfully used in a collection of logic programming definitions of stratification including that of stratified aggregates (see, for instance,

https://doi.org/10.1017/S1471068414000222 Published online by Cambridge University Press

(Faber *et al.* 2011). Unfortunately, limiting the language to stratified aggregates eliminates some of the useful forms of circles (see Example 9 below). In this paper we give a new form of VCP which goes beyond stratification: *p*(*a*) *cannot be introduced by the definition referring to a set of objects satisfying p if this set can contain a.* Technically, the principle is incorporated in our new definition of answer set (which coincides with the original definition for programs without aggregates). The definition is short and simple. We hope that, combined with a number of informal examples, it will be sufficient for developing an intuition necessary for the use of the language. The paper is organized as follows. In Section 2, we define the syntax and semantics of $\mathscr{A}log$. We give some properties of $\mathscr{A}log$ programs in Section 3 and present an algorithm for computing an answer set of an $\mathscr{A}log$ program in Section 4. A comparison with the existing work is done in Section 5, and we conclude the paper in Section 6.

2 Syntax and Semantics of $\mathscr{A}log$

We start with defining the syntax and intuitive semantics of the language.

2.1 Syntax

Let Σ be a (possibly sorted) signature with a finite collection of predicate, function, and object constants and $\mathscr{A}$ be a finite collection of symbols used to denote functions from finite sets of terms of Σ into integers. Terms and literals over signature Σ are defined as usual and referred to as *regular*. Regular terms are called *ground* if they contain no variables and no occurrences of symbols for arithmetic functions. Similarly for literals. An *aggregate term* is an expression of the form

$$f\{\bar{X} : cond\} \tag{1}$$

where $f \in \mathscr{A}$, *cond* is a collection of regular literals, and $\bar{X}$ is a list of variables occurring in *cond*. We refer to an expression

$$\{\bar{X} : cond\} \tag{2}$$

as a *set name*. An occurrence of a variable from $\bar{X}$ in (2) is called *bound* within (2). If the condition from (2) contains no variables except those in $\bar{X}$ then it is read as *the set of all objects of the program satisfying cond*. If *cond* contains other variables, say $\bar{Y} = \langle Y_1, \ldots, Y_n \rangle$, then $\{\bar{X} : cond\}$ defines the function mapping possible values $\bar{c} = \langle c_1, \ldots, c_n \rangle$ of these variables into sets $\{\bar{X} : cond|^{\bar{Y}}_{\bar{c}}\}$ where $cond|^{\bar{Y}}_{\bar{c}}$ is the result of replacing $Y_1, \ldots, Y_n$ by $c_1, \ldots, c_n$.

By an *aggregate atom* we mean an expression of the form

$$\langle aggregate\_term \rangle \langle arithmetic\_relation \rangle \langle arithmetic\_term \rangle \tag{3}$$

where *arithmetic_relation* is $>, \geqslant, <, \leqslant, =$ or $!=$, and *arithmetic_term* is constructed from variables and integers using arithmetic operations, $+$, $-$, $\times$, etc.

By *e-literals* we mean regular literals possibly preceded by default negation *not*. The latter (former) are called *negative* (*positive*) e-literals.

https://doi.org/10.1017/S1471068414000222 Published online by Cambridge University Press

A *rule* of $\mathscr{A}log$ is an expression of the form

$$head \leftarrow pos, neg, agg \tag{4}$$

where *head* is a disjunction of regular literals, *pos* and *neg* are collections of regular literals and regular literals preceded by *not* respectively, and *agg* is a collection of aggregate atoms. All parts of the rule, including *head*, can be empty. An occurrence of a variable in (4) not bound within any set name in this rule is called *free* in (4). A rule of $\mathscr{A}log$ is called *ground* if it contains no occurrences of free variables and no occurrences of arithmetic functions.

A *program* of $\mathscr{A}log$ is a finite collection of $\mathscr{A}log$'s rules. A program is *ground* if its rules are ground.

As usual for ASP based languages, rules of $\mathscr{A}log$ program with variables are viewed as collections of their ground instantiations. A *ground instantiation* of a rule r is the program obtained from r by replacing free occurrences of variables in r by ground terms of Σ and evaluating all arithmetic functions. If the signature Σ is sorted (as, for instance, in (Balai *et al.* 2013)) the substitutions should respect sort requirements for predicates and functions.
Clearly the grounding of an $\mathscr{A}log$ program is a ground program. The following examples illustrate the definition:

Example 3 (*Grounding: all occurrences of the set variable are bound*)
Consider a program P_2 with variables:

```
q(Y) :- card{X:p(X,Y)} = 1, r(Y).
r(a).  r(b).  p(a,b).
```

Here all occurrences of a set variable X are bound; all occurrences of a variable Y are free. The program's grounding, $ground(P_2)$, is

```
q(a) :- card{X:p(X,a)} = 1, r(a).
q(b) :- card{X:p(X,b)} = 1, r(b).
r(a).  r(b).  p(a,b).
```

The next example deal with the case when some occurrences of the set variable in a rule are free and some are bound.

Example 4 (*Grounding: some occurrences of a set variable are free*)
Consider an $\mathscr{A}log$ program P_3

```
r :- card{X:p(X)} >= 2, q(X).
p(a).  p(b).  q(a).
```

Here the occurrence of X in $q(X)$ is free. Hence the ground program $ground(P_3)$ is:

```
r :- card{X:p(X)} >= 2, q(a).
r :- card{X:p(X)} >= 2, q(b).
p(a).  p(b).  q(a).
```

https://doi.org/10.1017/S1471068414000222 Published online by Cambridge University Press

Note that despite its apparent simplicity the syntax of $\mathscr{A}log$ differs substantially from syntax of most other logic programming languages allowing aggregates (with the exception of that in (Gelfond 2002)). We illustrate the differences using the language presented in (Faber *et al.* 2011). (In what follows we refer to this language as $\mathscr{F}log$.) While syntactically programs of $\mathscr{A}log$ can also be viewed as programs of $\mathscr{F}log$ the opposite is not true. Among other things $\mathscr{F}log$ allows parameters of aggregates to be substantially more complex than those of $\mathscr{A}log$. For instance, an expression $f\{a : p(a,a), b : p(b,a)\} = 1$ where f is an aggregate atom of $\mathscr{F}log$ but not of $\mathscr{A}log$. This construction which is different from a usual set-theoretic notation used in $\mathscr{A}log$ is important for the $\mathscr{F}log$ definition of grounding. For instance the grounding of the first rule of program P_2 from Example 3 understood as a program of $\mathscr{F}log$ consists of $\mathscr{F}log$ rules

```
q(a) :- card{a:p(a,a),b:p(b,a)} = 1, r(a).
q(b) :- card{a:p(a,b),b:p(b,b)} = 1, r(b).
```

which is not even a program of $\mathscr{A}log$. Another important difference between the grounding methods of these languages can be illustrated by the $\mathscr{F}log$ grounding $ground_f(P_3)$ of program P_3 from Example 4 that looks as follows:

```
r :- card{a:p(a)} >= 2, q(a).
r :- card{b:p(b)} >= 2, q(b).
p(a).  p(b).  q(a).
```

Clearly this is substantially different from the $\mathscr{A}log$ grounding of P_3 from Example 4. In Section 5 we show that this difference in grounding reflects substantial semantic differences between the two languages.

2.2 Semantics

To define the semantics of $\mathscr{A}log$ programs we expand the standard definition of answer set from (Gelfond and Lifschitz 1988). The resulting definition captures the rationality principle - *believe nothing you are not forced to believe* (Gelfond and Kahl 2014) - and avoids vicious circles. As usual the definition of answer set is given for ground programs. Some terminology: a ground aggregate atom $f\{X : p(X)\} \odot n$ (where $\odot$ is one of the arithmetic relations allowed in the language) is *true* in a set of ground regular literals S if $f\{X : p(X) \in S\} \odot n$; otherwise the atom is *false* in I.

Definition 1 (*Aggregate Reduct*)
The *aggregate reduct* of a ground program Π of $\mathscr{A}log$ with respect to a set of ground regular literals S is obtained from Π by

1. removing from Π all rules containing aggregate atoms false in S.
2. replacing every remaining aggregate atom $f\{X : p(X)\} \odot n$ by the set $\{p(t) : p(t) \in S\}$ (which is called the *reduct of the aggregate* with respect to S).

(Here $p(t)$ is the result of replacing variable X by ground term t). The second clause of the definition reflects the principle of avoiding vicious circles – a rule with aggregate atom $f\{X : p(X)\} \odot n$ in the body can only be used if "the totality" of all

https://doi.org/10.1017/S1471068414000222 Published online by Cambridge University Press

objects satisfying p has already being constructed. Attempting to apply this rule to define $p(t)$ will either lead to contradiction or to turning the rule into tautology (see Examples 7 and 9).

Definition 2 (*Answer Set*)
A set S of ground regular literals over the signature of a ground program Π of $\mathcal{A}log$ is an *answer set* of Π if it is an answer set of an aggregate reduct of Π with respect to S.

We will illustrate this definition by a number of examples.

Example 5 (*Example 3 Revisited*)
Consider a program P_2 and its grounding from Example 3. It is easy to see that the aggregate reduct of the program with respect to any set S not containing $p(a,b)$ consists of the program facts, and hence S is not an answer set of P_2. However the program's aggregate reduct with respect to $A = \{q(b), r(a), r(b), p(a,b)\}$ consists of the program's facts and the rule

```
q(b) :- p(a,b),r(b).
```

Hence A is an answer set of P_2.

Example 6 (*Example 4 Revisited*)
Consider now the grounding

```
r :- card{X:p(X)} >= 2, q(a).
r :- card{X:p(X)} >= 2, q(b).
p(a).  p(b).  q(a).
```

of program P_3 from Example 4. Any answer set S of this program must contain its facts. Hence $\{X : p(X) \in S\} = \{a, b\}$. S satisfies the body of the first rule and must also contain r. Indeed, the aggregate reduct of P_3 with respect to $S = \{p(a), p(b), q(a), r\}$ consists of the facts of P_3 and the rules

```
r :- p(a),p(b),q(a).
r :- p(a),p(b),q(b).
```

Hence S is the answer set of P_3.

Neither of the two examples above required the application of VCP. The next example shows how this principle influences our definition of answer sets and hence our reasoning.

Example 7 (*Example 2 Revisited*)
Consider a program P_1 from Example 2. The program, consisting of a rule

```
p(a) :- card{X : p(X)}=1
```

is grounded. It has two candidate answer sets, $S_1 = \{\ \}$ and $S_2 = \{p(a)\}$. The aggregate reduct of the program with respect to S_1 is the empty program. Hence, S_1 is an answer set of P_1. The program's aggregate reduct with respect to S_2 however is

```
p(a) :- p(a).
```

https://doi.org/10.1017/S1471068414000222 Published online by Cambridge University Press

The answer set of this reduct is empty and hence S_1 is the only answer of P_1.

Example 7 shows how the attempt to define $p(a)$ in terms of totality of p turns the defining rule into a tautology. The next example shows how it can lead to inconsistency of a program.

Example 8 (*Vicious Circles through Aggregates and Inconsistency*)
Consider a program P_4:

```
p(a).
p(b) :- card{X:p(X)} > 0.
```

Since every answer set of the program must contain $p(a)$, the program has two candidate answer sets: $S_1 = \{p(a)\}$ and $S_2 = \{p(a), p(b)\}$. The aggregate reduct of P_4 with respect to S_1 is

```
p(a).
p(b) :- p(a).
```

The answer set of the reduct is $\{p(a), p(b)\}$ and hence S_1 is not an answer set of P_4. The reduct of P_4 with respect to S_2 is

```
p(a).
p(b) :- p(a),p(b).
```

Again its answer set is not equal to S_2 and hence P_4 is inconsistent (i.e., has no answer sets). The inconsistency is the direct result of an attempt to violate the underlying principle of the semantics. Indeed, the definition of $p(b)$ refers to the set of objects satisfying p that can contain b which is prohibited by our version of VCP. One can, of course, argue that S_2 can be viewed as a reasonable collection of beliefs which can be formed by a rational reasoner associated with P_4. After all, we do not need the totality of p to satisfy the body of the rule defining $p(b)$. It is sufficient to know that p contains a. This is indeed true but this reasoning depends on the knowledge which is not directly incorporated in the definition of $p(b)$. If one were to replace P_4 by

```
p(a).
p(b) :- card{X:p(X), X != b} > 0.
```

then, as expected, the vicious circle principle will not be violated and the program will have unique answer set $\{p(a), p(b)\}$.

We end this section by a simple but practical example of a program which allows recursion through aggregates but avoids vicious circles.

Example 9 (*Defining Digital Circuits*)
Consider part of a logic program formalizing propagation of binary signals through simple digital circuits. We assume that the circuit does not have a feedback, i.e., a wire receiving a signal from a gate cannot be an input wire to this gate. The program may contain a simple rule

https://doi.org/10.1017/S1471068414000222 Published online by Cambridge University Press

```
val(W,0) :-
      gate(G, and),
      output(W, G),
      card{W: val(W,0), input(W, G)} > 0.
```

(partially) describing propagation of symbols through an *and* gate. Here $val(W, S)$ holds iff the digital signal on a wire W has value S. Despite its recursive nature the definition of *val* avoids vicious circle. To define the signal on an output wire W of an *and* gate G one needs to only construct a particular subset of input wires of G. Since, due to absence of feedback in our circuit, W can not belong to the latter set our definition is reasonable. To illustrate that our definition of answer set produces the intended result let us consider program P_5 consisting of the above rule and a collection of facts:

```
gate(g, and).
output(w0, g).
input(w1, g).
input(w2, g).
val(w1,0).
```

The grounding, $ground(P_5)$, of P_5 consists of the above facts and the three rules of the form

```
val(w,0) :-
      gate(g, and),
      output(w, g),
      card{W: val(W,0), input(W, g)} > 0.
```

where w is w_0, w_1 ,and w_2.
Let $S = \{gate(g, and), val(w1, 0), val(w0, 0), output(w0, g), input(w1, g), input(w2, g)\}$. The aggregate reduct of $ground(P_5)$ with respect to S is the collection of facts and the rules

```
val(w,0) :-
      gate(g, and),
      output(w, g),
      input(w1, g),
      val(w1, 0).
```

where w is w_0, w_1, and w_2.
The answer set of the reduct is S and hence S is an answer set of P_5. As expected it is the only answer set. (Indeed it is easy to see that other candidates do not satisfy our definition.)

3 Properties of $\mathscr{A}log$ programs

In this section we give some basic properties of $\mathscr{A}log$ programs. Propositions 1 and 2 ensure that, as in regular ASP, answer sets of $\mathscr{A}log$ program are formed using

https://doi.org/10.1017/S1471068414000222 Published online by Cambridge University Press

the program rules together with the rationality principle. Proposition 3 is the $\mathscr{A}log$ version of the basic technical tool used in theoretical investigations of ASP and its extensions. Proposition 4 shows that complexity of entailment in $\mathscr{A}log$ is the same as that in regular ASP.

We will use the following terminology: e-literals p and *not* p are called *contrary*; *not* l denotes a literal contrary to e-literal l; a *partial interpretation* I over signature Σ is a consistent set of e-literals of this signature; an e-literal l is *true* in I if $l \in I$; it is *false* if *not* $l \in I$; otherwise l is *undefined* in I. An aggregate atom $f\{X : q(X)\} \odot n$ is *true* in I if $f\{t : q(t) \in I\} \odot n$ is true, i.e., the value of f on the set $\{t : q(t) \in I\}$ and the number n satisfy property $\odot$. Otherwise, the atom is *false* in I. The head of a rule is *satisfied* by I if at least one of its literals is true in I; the body of a rule is *satisfied* by I if all of its aggregate atoms and e-literals are true in I. A *rule is satisfied* by I if its head is satisfied by I or its body is not satisfied by I.

Proposition 1 (*Rule Satisfaction and Supportedness*)
Let A be an answer set of a ground $\mathscr{A}log$ program Π. Then

1. A satisfies every rule r of Π.
2. If $p \in A$ then there is a rule r from Π such that the body of r is satisfied by A and p is the only atom in the head of r which is true in A. (It is often said that rule r supports atom p.)

Proposition 2 (*Anti-chain Property*)
Let A_1 be an answer set of an $\mathscr{A}log$ program Π. Then there is no answer set A_2 of Π such that A_1 is a proper subset of A_2.

Proposition 3 (*Splitting Set Theorem*)
Let Π_1 and Π_2 be programs of $\mathscr{A}log$ such that no atom occurring in Π_1 is a head atom of Π_2. Let S be a set of atoms containing all head atoms of Π_1 but no head atoms of Π_2. A set A of atoms is an answer set of $\Pi_1 \cup \Pi_2$ iff $A \cap S$ is an answer set of Π_1 and A is an answer set of $(A \cap S) \cup \Pi_2$.

Proposition 4 (*Complexity*)
The problem of checking if a ground atom a belongs to all answer sets of an $\mathscr{A}log$ program is Π_2^p complete.

4 An Algorithm for Computing Answer Sets

In this section we briefly outline an algorithm, called $\mathscr{A}solver$, for computing answer sets of $\mathscr{A}log$ programs. We follow the tradition and limit our attention to programs without classical negation. Hence, in this section we consider only programs of this type. By an *atom* we mean an e-atom or an aggregate atom.

Definition 3 (*Strong Satisfiability and Refutability*)

- An atom is *strongly satisfied* (*strongly refuted*) by a partial interpretation I if it is true (false) in every partial interpretation containing I; an atom which is neither strongly satisfied nor strongly refuted by I is *undecided* by I.

https://doi.org/10.1017/S1471068414000222 Published online by Cambridge University Press

- A set S of atoms is *strongly satisfied* by I if all atoms in S are strongly satisfied by I;
- S is *strongly refuted* by I if for every partial interpretation I' containing I, some atom of S is false in I'.

For instance, an e-atom is strongly satisfied (refuted) by I iff it is true (false) in I; an atom $card\{X : p(X)\} > n$ which is true in I is strongly satisfied by I; an atom $card\{X : p(X)\} < n$ which is false in I is strongly refuted by I; and a set $\{f\{X : p(X)\} > 5, f\{X : p(X)\} < 3\}$ is strongly refuted by any partial interpretation.

$\mathscr{A}solver$ consists of three functions: $Solver$, $Cons$, and $IsAnswerSet$. The main function, $Solver$, is similar to that used in standard ASP algorithms (See, for instance, $Solver1$ from (Gelfond and Kahl 2014)). But unlike these functions which normally have two parameters - partial interpretation I and program Π - $Solver$ has two additional parameters, TA and FA containing aggregate atoms that must be true and false respectively in the answer set under construction. $Solver$ returns $\langle I, true\rangle$ where I is an answer set of Π compatible with its parameters and $false$ if no such answer set exists. The $Solver$'s description will be omitted due to space limitations. The second function, $Cons$, computes the consequences of its parameters - a program Π, a partial interpretation I, and two above described sets TA and FA of aggregates atoms. Due to the presence of aggregates the function is sufficiently different from a typical $Cons$ function of ASP solvers so we describe it in some detail. The new value of I, containing the desired consequences is computed by application of the following **inference rules**:

1. If the body of a rule r is strongly satisfied by I and all atoms in the head of r except p are false in I then p must be in I.
2. If an atom $p \in I$ belongs to the head of exactly one rule r of Π then every other atom from the head of r must have its complement in I, the e-atoms from the body of r must be in I and its aggregate atoms must be in TA.
3. If every atom of the head of a rule r is false in I, and l is the only premise of r which is either an undefined e-atom or an aggregate atom not in FA, and the rest of the body is strongly satisfied by I, then
   (a) if l is an e-atom, then the complement of l must be in I,
   (b) if l is an aggregate atom, then it must be in FA.
4. If the body of every rule with p in the head is strongly refuted by I, then $(not\ p)$ must be in I.

Given an interpretation I, a program Π, inference rule $i \in [1..4]$ and $r \in \Pi$, let function $iCons(i, I, \Pi, r)$ return $< \delta I, \delta TA, \delta FA >$ where δI, δTA and δFA are the results of applying inference rule i to r. (Note, that inference rule 4 does not really use r). We also need the following terminology. We say that I is *compatible* with TA if TA is not strongly refuted by I; I is *compatible* with FA if no atom from FA is strongly satisfied by I. A set A of regular atoms is *compatible* with TA and FA if the set $compl(A) = \{p : p \in A\} \cup \{not\ a : a \notin A\}$ is *compatible* with TA and FA; A is compatible with I if $I \subseteq compl(A)$. The algorithm $Cons$ is listed below.

https://doi.org/10.1017/S1471068414000222 Published online by Cambridge University Press

function Cons

input: partial interpretation I_0, sets TA_0 and FA_0 of aggregate atoms compatible with I_0, and program Π_0 with signature Σ_0;

output:

$\langle \Pi, I, TA, FA, true \rangle$ where I is a partial interpretation such that $I_0 \subseteq I$, TA and FA are sets of aggregate atoms such that $TA_0 \subseteq TA$ and $FA_0 \subseteq FA$,
I is compatible with TA and FA, and Π is a program with signature Σ_0 such that for every A,
A is an answer set of Π_0 that is compatible with I_0 iff A is an answer set of Π that is compatible with I.

$\langle \Pi_0, I_0, TA_0, FA_0, false \rangle$ if there is no answer set of Π_0 compatible with I_0;

var I, T: set of e-atoms; TA, FA: set of aggregate atoms; Π: program;

1. Initialize I, Π, TA and FA to be I_0, Π_0, TA_0 and FA_0 respectively;
2. **repeat**
3. $T := I$;
4. Remove from Π all the rules whose bodies are strongly falsified by I;
5. Remove from the bodies of rules of Π all negative e-atoms true in I and aggregate atoms strongly satisfied by I;
6. Non-deterministically select an inference rule i from (1)–(4);
8. **for** every $r \in \Pi$
9. $< \delta I, \delta TA, \delta FA > := iCons(I, \Pi, i, r)$;
10. $I := I \cup \delta I$, $TA := TA \cup \delta TA$, $FA := FA \cup \delta FA$;
11. **until** $I = T$;
12. **if** I is consistent, TA and FA are compatible with I **then**
13. **return** $< \Pi, I, TA, FA, true >$;
14. **else return** $< \Pi_0, I_0, TA_0, FA_0, false >$;

The third function, *IsAnswerSet* of our solver $\mathscr{A}solver$ checks if interpretation I is an answer set of a program Π. It computes the aggregate reduct of Π with respect to I and applies usual checking algorithm (see, for instance, (Koch *et al.* 2003)).

Proposition 5 (*Correctness of the Solver*)
If, given a program Π_0, a partial interpretation I_0, and sets TA_0 and FA_0 of aggregate atoms Solver(I_0, TA_0, FA_0, Π_0) returns $\langle I, true \rangle$ then I is an answer set of Π_0 compatible with I_0, TA_0 and FA_0. If there is no such answer set, the solver returns $false$.

To illustrate the algorithm consider a program Π

```
:- p(a).
p(a) :- card{X:q(X)} > 0.
q(a) or p(b).
```

and trace $Solver(\Pi, I, TA, FA)$ where I, TA, and FA are empty. *Solver* starts by calling *Cons* which computes the consequence *not* $p(a)$ (from the first rule

https://doi.org/10.1017/S1471068414000222 Published online by Cambridge University Press

of the program), $FA = \{card\{X : q(X)\} > 0\}$ (from the second rule of the program) and *not* $q(b)$ (from the fourth inference rule), and returns *true*, $I = \{not\ q(b), not\ p(a)\}$ and new FA; TA is unchanged. $Solver$ then guesses $q(a)$ to be true, i.e., $I = \{not\ q(b), not\ p(a), q(a)\}$, and calls $Cons$ again. $Cons$ does not produce any new consequences but finds that FA is not compatible with I (line 12 of the algorithm). So, it returns $false$, which causes $Solver$ to set $q(a)$ to be false, i.e., $I = \{not\ q(b), not\ p(a), not\ q(a)\}$. $Solver$ then calls $Cons$ again which returns $I = \{not\ q(b), not\ p(a), not\ q(a), p(b)\}$. $Solver$ finds that I is complete and calls $IsAnswerSet$ which returns true. Finally, $Solver$ returns I as an answer set of the program.

5 Comparison with Other Approaches

There are a large number of approaches to the syntax and semantics of extensions of ASP by aggregates. In this section we concentrate on languages from (Son and Pontelli 2007) and (Faber *et al.* 2011) which we refer to as $\mathscr{S}log$ and $\mathscr{F}log$ respectively. Due to multiple equivalence results discussed in these papers this is sufficient to cover most of the approaches. The main difference between the syntax of aggregates in $\mathscr{A}log$ and $\mathscr{F}log$ is in treatment of variables occurring in *aggregate terms*. $\mathscr{A}log$ uses usual logical concept of bound and free occurrence of a variable (the occurrence of X within $S = \{X : p(X, Y)\}$ is bound while the occurrence of Y is free). $\mathscr{F}log$ uses very different concepts of global and local variable of a *rule*. A *variable is local in rule r if it occurs solely in an aggregate term of r; otherwise, the variable is global*. As the result, in $\mathscr{A}log$, every aggregate term $\{X : p(X)\}$ can be replaced by a term $\{Y : p(Y)\}$ while it is not the case in $\mathscr{F}log$. In our opinion the approach of $\mathscr{F}log$ (and many other languages and systems which adopted this syntax) makes declarative reading of aggregate terms substantially more difficult[1]. To see the semantic ramifications of the $\mathscr{F}log$ treatment of variables consider the following example:

Example 10 (*Variables in Aggregate Terms: Global versus Bound*)
Consider program P_3 from Example 4. According to $\mathscr{F}log$ the meaning of an occurrence of an expression $\{X : p(X)\}$ in the body of the program's first rule changes if X is replaced by a different variable. In $\mathscr{A}log$, where X is understood as bound this is not the case. This leads to substantial difference in grounding and in the semantics of the program. In $\mathscr{A}log$ P_3 has one answer set, $\{p(a), p(b), q(a), r\}$. In $\mathscr{F}log$ answer sets of P_3 are those of $ground_f(P_3)$. The answer set of the latter is $\{p(a), p(b), q(a)\}$.

Other semantic differences are due to the multiplicity of informal (and not necessarily clearly spelled out) principles underlying various semantics.

[1] The other difference in reading of S is related to the treatment of variable Y. In $\mathscr{F}log$ the variable is bound by an unseen existential quantifier. If all the variables are local then $S = \{X : p(X, Y)\}$ is really $S_1 = \{X : \exists Y\ p(X, Y)\}$. In $\mathscr{A}log$ Y is free. Both approaches are reasonable but we prefer to deal with the different possible readings by introducing an explicit existential quantifier as in Prolog. It is easy semantically and we do not discuss it in the paper.

https://doi.org/10.1017/S1471068414000222 Published online by Cambridge University Press

Example 11 (*Vicious Circles in $\mathscr{F}log$*)
Consider the following program, P_6, adopted from (Son and Pontelli 2007):

```
p(1) :- p(0).
p(0) :- p(1).
p(1) :- count{X: p(X)} != 1.
```

which, if viewed as $\mathscr{F}log$ program, has one answer set $A = \{p(0), p(1)\}$. Informal argument justifying this result goes something like this: Clearly, A satisfies the rules of the program. To satisfy the minimality principle no proper subset of A should be able to do that, which is easily checked to be true. Faber et al use so called *black box principle*: "when checking stability they [aggregate literals] are either present in their entirety or missing altogether", i.e., the semantics of $\mathscr{F}log$ does not consider the process of derivation of elements of the aggregate parameter. Note however, that the program's definition of $p(1)$ is given in terms of fully defined term $\{X : p(X)\}$, i.e., the definition contains a vicious circle. This explains why A is not an answer set of P_6 in $\mathscr{A}log$. In this particular example we are in agreement with $\mathscr{S}log$ which requires that the value of an aggregate atom can be computed before the rule with this atom in the body can be used in the construction of an answer set.

The absence of answer set of P_6 in $\mathscr{S}log$ may suggest that it adheres to our formalization of the VCP. The next example shows that it is not the case.

Example 12 (*VCP and Constructive Semantics of aggregates*)
Let us consider a program P_7.

```
p(a) :- count{X:p(X)} > 0.
p(b) :- not q.
q :- not p(b).
```

As shown in (Son and Pontelli 2007) the program has two $\mathscr{S}log$ answer sets, $A = \{q\}$ and $B = \{p(a), p(b)\}$. If viewed as a program of $\mathscr{A}log$, P_7 will have one answer set, A. This happens because the $\mathscr{S}log$ construction of B uses knowledge about properties of the *aggregate atom* of the first rule; the semantics of $\mathscr{A}log$ only takes into account the *meaning of the parameter of the aggregate term.* Both approaches can, probably, be successfully defended but, in our opinion, the constructive semantics has a disadvantage of being less general (it is only applicable to non-disjunctive programs), and more complex mathematically.

A key difference between our algorithm and those in the existing work (Faber *et al.* 2008; Gebser *et al.* 2009) is that the other work needs rather involved methods to ground the aggregates while our algorithm does not need to ground the aggregate atoms. As a result, the ground program used by our algorithm may be smaller, and our algorithm is simpler.

There is also a close connection between the above semantics of aggregates all of which are based on some notion of a reduct or a fixpoint computation and approaches in which aggregates are represented as special cases of more general constructs, such as propositional formulas (Ferraris 2005; Harrison *et al.* 2013) and abstract constraint atoms (Marek *et al.* 2004; Liu *et al.* 2010; Wang *et al.* 2012) (Our

https://doi.org/10.1017/S1471068414000222 Published online by Cambridge University Press

semantics can be easily extended to the latter). Some of the existing equivalence results allow us to establish the relationship between these approaches and $\mathscr{A}log$. Others require further investigation.

6 Conclusion and Future Work

We presented an extension, $\mathscr{A}log$, of ASP which allows for the representation of and reasoning with aggregates. We believe that the language satisfies design criteria of simplicity of syntax and formal and informal semantics. There are many ways in which this work can be continued. The first, and simplest, step is to expand $\mathscr{A}log$ by allowing choice rules similar to those of (Niemela *et al.* 2002). This can be done in a natural way by combining ideas from this paper and that from (Gelfond 2002). We also plan to investigate mapping of $\mathscr{A}log$ into logic programs with arbitrary propositional formulas. There are many interesting and, we believe, important questions related to optimization of the $\mathscr{A}log$ solver from Section 4. After clarity is reached in this area one will, of course, try to address the questions of implementation.

7 Acknowledgment

We would like to thank Amelia Harrison, Patrick Kahl, Vladimir Lifschitz, and Tran Cao Son for useful comments. The authors' work was partially supported by NSF grant IIS-1018031.

Supplementary material

To view supplementary material for this article, please visit http://dx.doi.org/10.1017/S1471068414000222.

References

Balai, E., Gelfond, M., and Zhang, Y. 2013. Towards answer set programming with sorts. In *LPNMR*. 135–147.

Brewka, G., Eiter, T., and Truszczynski, M. 2011. Answer set programming at a glance. *Commun. ACM 54,* 12, 92–103.

Erdem, E., Lee, J., and Lierler, Y. 2012. Theory and practice of answer set programming. AAAI-2012 Tutorial (`http://peace.eas.asu.edu/aaai12tutorial/asp-tutorial-aaai.pdf`).

Faber, W., Pfeifer, G., and Leone, N. 2011. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence 175,* 1, 278–298.

Faber, W., Pfeifer, G., Leone, N., Dell'Armi, T., and Ielpa, G. 2008. Design and implementation of aggregate functions in the dlv system. *TPLP 8,* 5-6, 545–580.

Feferman, S. 2002. Predicativity. `http://math.stanford.edu/ feferman/papers/`.

Ferraris, P. 2005. Answer sets for propositional theories. In *LPNMR*. 119–131.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. 2009. On the implementation of weight constraint rules in conflict-driven asp solvers. In *ICLP*. 250–264.

Gebser, M., Kaufman, B., Neumann, A., and Schaub, T. 2007. Conflict-driven answer set enumeration. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, C. Baral, G. Brewka, and J. Schlipf, Eds. lnai, vol. 3662. Springer, 136–148.

Gelfond, M. 2002. Representing Knowledge in A-Prolog. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, A. C. Kakas and F. Sadri, Eds. Vol. 2408. Springer Verlag, Berlin, 413–451.

Gelfond, M. and Kahl, Y. 2014. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents*. Cambridge University Press.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of ICLP-88*. 1070–1080.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9,* 3/4, 365–386.

Harrison, A., Lifschitz, V., and Yang, F. 2013. On the semantics of gringo. In *Working Notes of the Workshop on Answer Set Programming and Other Computing Paradigms*.

Kemp, D. B. and Stuckey, P. J. 1991. Semantics of logic programs with aggregates. In *ISLP*. Vol. 91. Citeseer, 387–401.

Koch, C., Leone, N., and Pfeifer, G. 2003. Enhancing disjunctive logic programming systems by sat checkers. *Artif. Intell. 151,* 1-2, 177–212.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic 7*, 499–562.

Liu, L., Pontelli, E., Son, T. C., and Truszczynski, M. 2010. Logic programs with abstract constraint atoms: The role of computations. *Artif. Intell. 174,* 3-4, 295–315.

Marek, V. W., Truszczynski, M., et al. 2004. Logic programs with abstract constraint atoms. In *AAAI*. Vol. 4. 86–91.

Niemela, I., Simons, P., and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138,* 1–2 (Jun), 181–234.

Pelov, N., Denecker, M., and Bruynooghe, M. 2007. Well-fouded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming 7*, 355–375.

Poincare, H. 1906. Les mathematiques et la logique. *Review de metaphysique et de morale 14*, 294–317.

Son, T. C. and Pontelli, E. 2007. A constructive semantic characterization of aggregates in answer set programming. *TPLP 7,* 3, 355–375.

Wang, Y., Lin, F., Zhang, M., and You, J.-H. 2012. A well-founded semantics for basic logic programs with arbitrary abstract constraint atoms. In *AAAI*.