Ultimate Well-founded and Stable Semantics for Logic Programs with Aggregates

Marc Denecker, Nikolay Pelov, and Maurice Bruynooghe

Dept. of Computer Science, K.U.Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
{marcd,pelov,maurice}@cs.kuleuven.ac.be

Abstract. In this paper, we propose an extension of the well-founded and stable model semantics for logic programs with aggregates. Our approach uses Approximation Theory, a fixpoint theory of stable and well-founded fixpoints of non-monotone operators in a complete lattice. We define the syntax of logic programs with aggregates and define the immediate consequence operator of such programs. We investigate the well-founded and stable semantics generated by Approximation Theory. We show that our approach extends logic programs with stratified aggregation and that it correctly deals with well-known benchmark problems such as the shortest path program and the company control problem.

1 Introduction

Programs with aggregates have been investigated in the context of database applications [1, 14, 17, 20]. One important approach to programs with aggregates is *Monotone aggregation* [17, 20]. This approach concentrates on programs where the 2-valued immediate consequence operator is monotone with respect to some given lattice order. In such cases, one can take the least fixpoint as the intended model of the program. Another approach is by using stratification. *Aggregate stratified programs* [1, 14] can be split up in a sequence of different *strata* such that each aggregate expression only refers to predicates defined in previous strata. A disadvantage of both approaches is that they impose serious syntactic restrictions on the programs. Therefore for many problems, one has to carefully tune the program in such a way that it satisfies the syntactic restrictions. For this reason, a number of attempts have been made for extending the well-founded semantics [21] and the stable semantics [10] for unstratified programs with aggregates.

Well-founded semantics is a natural semantics for deductive databases. As shown in [2, 4] it captures a general principle of non-monotone inductive definition. For this reason, [3] used the well-founded semantics as the semantic principle of ID-logic, a logic for knowledge representation which integrates classical logic assertions and inductive definitions. ID-logic can be seen as an extension and provides an epistemological foundation for Abductive logic Programming [11]. Also the stable semantics has been shown to be important for knowledge representation, in particular for nonmonotonic reasoning. Since recently, it is

used as the foundation of an emerging paradigm of *stable logic programming* [13] where problems are solved by computing stable models of a logic program. Since recently, aggregates are attracting increasing interest in the context of these extensions. Experiments in solving combinatorial search problems showed that many of these problems cannot be adequately expressed without aggregation. Recently, one of the prominent systems for stable logic programming **smodels** [15] has been extended successfully with a limited form of aggregate constraints [16]. [22] describes an extension of an abductive system with aggregates and some experiments with this system in the context of scheduling and puzzles.

As will be shown in section 5, current extensions of well-founded semantics [12, 18, 19] and stable model semantics [12, 16] of programs with aggregates are still weak in some ways. In this paper, we propose a stronger extension of the well-founded and stable model semantics for logic programs with aggregates. Our approach uses Approximation Theory developed in [5]. With each nonmonotone operator, this theory associates a Kripke-Kleene, a Well-founded and a set of stable fixpoints. In the case of the immediate consequence operator of a logic program, these fixpoints identify exactly the models in the corresponding types of Logic Programming semantics. We define the syntax of logic programs with aggregates and define an immediate consequence operator of such programs. We investigate the well-founded and stable semantics generated by Approximation Theory, and show this approach extends logic programs with stratified aggregation and that it correctly deals with recursive benchmark problems such as the shortest path program and the company control problem. An important property is that if the 2-valued immediate consequence operator is monotone then its ultimate well-founded model coincides with the least fixpoint of this operator.

2 Aggregate Logic Programs: Syntax and T_P

A sorted signature Σ is a set of sort symbols and sorted function, predicate and variable symbols. The arity of a function symbol is a tuple $(s_1, \dots, s_n) : s$ where $s_1, \dots, s_n, s$ are sort symbols; the arity of a predicate symbol is a tuple $(s_1, \dots, s_n)$. We introduce a set $Agg \subseteq \Sigma$ of sorted second order predicate symbols. These will be called the aggregate symbols. The arity of a symbol F of Agg is specified as a tuple $(\tau_1, \dots, \tau_n)$ where τ_i is either a tuple $(s_1, \dots, s_m)$ specifying the sort of a relation argument, a tuple $(s_1, \dots, s_m) : s$ specifying the sort of a function argument or a first order sort symbol s . Sort symbols *nat*, *int* and *real* denote the natural, respectively integer and real numbers.

Example 2.1. The following aggregate symbols are used below:

- $Card_s$: has arity $((s), nat)$ denotes the cardinality relation of sets of sort s .
- Min_s, Max_s : have arity $((s), s)$ and denote the minimality, respectively maximality relation of sets of the sort s representing a partial order.
- $Sum_{(s_1, \dots, s_n)}$: has arity $((s_1, \dots, s_n), (s_1, \dots, s_n) : nat, nat)$ and denotes the sum function that summates a function over a set. I.e. for a relation R , function F and integer n , $Sum(R, F, n)$ holds iff $n = \sum_{(x_1, \dots, x_n) \in R} F(x_1, \dots, x_n)$ and R is finite.

All these aggregate symbols are indexed by the sorts over which they range. In particular, *Agg* may contain many cardinality, minimality, maximality or summation aggregates, each ranging over other sorts. However, in the sequel, we will drop the annotations with sorts and will write *Card*, *Min*, *Max*, *Sum*, ...

A signature Σ is interpreted over a Σ -structure I ; this is an assignment of:

- a set D_s to each sort symbol s ,
- a function $f_I : D_{s_1} \times \dots \times D_{s_n} \rightarrow D_s$ to each function symbol f with arity $(s_1, \dots, s_n) : s$,
- a relation $p_I \subseteq D_{s_1} \times \dots \times D_{s_n}$ to each predicate symbol p with arity $(s_1, \dots, s_n)$.
- an appropriately typed second order relation F_I to each aggregate symbol $F \in \text{Agg}$.

Next we define the syntax of aggregate formulas and aggregate programs based on Σ as a restricted subset of second order formulas. We define the notions of *lambda expression* and of *set expression*, *aggregate atom* and *aggregate formula* by simultaneous induction:

- A *lambda expression* is of the form $\lambda(X_1, \dots, X_n)t$ where $X_1, \dots, X_n$ are different variables and t is a first order term. Variables appearing in t and not among $X_1, \dots, X_n$ are called the free variables of the lambda expression.
- A *set expression* is of the form $\{(X_1, \dots, X_n) | \phi\}$ where again $X_1, \dots, X_n$ are different variables and ϕ an aggregate formula. Variables appearing in ϕ and not among $X_1, \dots, X_n$ are called the free variables of the set expression.
- An *aggregate atom* is a well-typed second order atom where each function argument contains a lambda expression, each relation argument contains a set expression and each other argument contains a term.
- An *aggregate formula* is either a (first order) atom, an aggregate atom or is a composed formula obtained by applying the normal composition rules for $\wedge, \vee, \leftarrow, \neg, \forall, \exists$.

Variables of aggregate atoms are first order, hence quantification is never over second order variables. Also, a set expression which is part of an aggregate atom in turn contains an aggregate formula, hence, nested aggregation is allowed.

Example 2.2. The formula

$$\forall T. \text{Sum} \left(\begin{array}{c} \{(U, C) | \text{bucket}(U) \wedge \neg \text{leaking}(U) \wedge \text{capacity}(U, C)\}, \\ \lambda(U, C)C, \\ T \end{array} \right) \rightarrow T \geq 1000$$

expresses that the sum of the capacity of non-leaking buckets is at least 1000. The first argument denotes the set of pairs of a non-leaking bucket and its capacity while the second argument represents the projection function. The argument U (the *bucket*) in the set and lambda expression is necessary as a capacity has to be counted as many times as there are non-leaking buckets with it. Using a function *capac* mapping buckets on their capacity, an alternative formulation is:

$$\forall T. \text{Sum} \left(\begin{array}{c} \{(U) | \text{bucket}(U) \wedge \neg \text{leaking}(U)\}, \\ \lambda(U) \text{capac}(U), \\ T \end{array} \right) \rightarrow T \geq 1000$$

Its meaning is: $\forall T. (T = \sum_{U \in \{U \mid \text{bucket}(U) \wedge \neg \text{leaking}(U)\}} \text{capac}(U)) \rightarrow T \geq 1000$

Aggregate expressions have been denoted in different ways. The most common (e.g [12]) is of the form $\text{group_by}(p(\overline{X}, \overline{Y}), [\overline{X}], C = F(t[\overline{X}, \overline{Y}]))$ where $\overline{X}$ are the grouping variables and C is the result of computing the aggregate function F . It is equivalent in our notation to: $F(\{\overline{Y} \mid p(\overline{X}, \overline{Y})\}, \lambda(\overline{Y})t[\overline{X}, \overline{Y}], C)$. For example, the atom $\text{group_by}(d(A, B, M), [A], N = \text{Sum}(M))$ is written in the syntax defined here as $\text{Sum}(\{(B, M) \mid d(A, B, M)\}, \lambda(B, M)M, N)$ and denotes that $N = \sum_{(B, M) \in \{(B, M) \mid d(A, B, M)\}} M$.

Next we define aggregate programs. Let Σ be a sorted signature of symbols and $\Sigma' = \Sigma \cup \Sigma_d$ an extension of Σ with a set Σ_d of sorted predicate symbols, called the *defined predicate symbols*. An aggregate program based on Σ' is a set of rules of the form $p(t_1, \dots, t_n) \leftarrow B$ where p is a defined predicate, $t_1, \dots, t_n$ are appropriately typed first order terms and B is an aggregate formula based on Σ' . The resulting syntax extends the standard logic programming syntax by allowing aggregate expressions and general formulas in the body of rules.

2.1 Examples

Example 2.3 (Shortest path). Given are two sorts *edge* and *weight* and an interpreted predicate symbol *edge* of arity $(\text{edge}, \text{edge}, \text{weight})$ representing a directed weighted graph. The defined predicate *sp* has the same sort; $\text{sp}(X, Y, Z)$ means that the shortest path from X to Y has total weight Z . It is defined as:

$$\text{sp}(X, Y, W) \leftarrow \text{Min} \left(\left\{ W \mid \begin{array}{l} \text{edge}(X, Y, W) \vee \\ \exists Z, W_1, W_2. (\text{sp}(X, Z, W_1) \wedge \\ \text{edge}(Z, Y, W_2) \wedge W = W_1 + W_2) \end{array} \right\}, W \right).$$

Example 2.4 (Company control). A sort *comp* represents companies and a sort *shares* interpreted over the real interval $[0..1]$ represents shares. The interpreted predicate *owns_stock* has arity $(\text{comp}, \text{comp}, \text{shares})$ and $\text{owns_stock}(X, Y, S)$ means that a company X owns a fraction S of the stock of a company Y . The defined predicate *Controls* has arity $(\text{comp}, \text{comp})$; $\text{Controls}(X, Y)$ denotes that company X has a controlling interest in a company Y . This is the case if X owns (by its own shares in Y augmented with the shares of other companies Z controlled by X) more than 50% of the stock of Y . It is defined as:

$$\text{Controls}(X, Y) \leftarrow \text{Sum} \left(\left\{ (Z, S) \mid \begin{array}{l} \text{owns_stock}(X, Y, S) \wedge Z = X \vee \\ \text{controls}(X, Z) \wedge \text{owns_stock}(Z, Y, S) \end{array} \right\}, \right. \\ \left. \lambda(Z, S)S, \right. \\ \left. T \right. \\ \left. \wedge T > 0.5 \right)$$

2.2 Immediate Consequence Operator

First, we give a standard definition of truth assignment of aggregate formulas. Given is a signature Σ and a Σ -structure I . A variable assignment σ is a (well-typed) mapping from the variables of Σ to domain elements. For a given variable assignment σ , a tuple of variables $\overline{X} = (X_1, \dots, X_n)$ and a tuple of domain elements $\overline{x} = (x_1, \dots, x_n)$, we denote by $\sigma[\overline{X}/\overline{x}]$ the variable assignment σ' such that $\sigma'(X) = \sigma(X)$ for each variable X not in $\overline{X}$, and $\sigma'(X_i) = x_i$. We define satisfiability $\models$ of a formula in a structure and the evaluation operator $eval_{I,\sigma}$ which maps first order terms to domain elements, lambda expressions to functions and set expressions to relations. The definition is by simultaneous induction:

- $eval_{I,\sigma}(X) = \sigma(X)$
- $eval_{I,\sigma}(f(t_1, \dots, t_n)) = f_I(eval_{I,\sigma}(t_1), \dots, eval_{I,\sigma}(t_n))$
- $eval_{I,\sigma}(\lambda(\overline{X})t) = \{(\overline{x}, y) \mid (\overline{x}, y) \in D_{s_1} \times \dots \times D_{s_n} \times D_s \text{ and } y = eval_{I,\sigma}[\overline{X}/\overline{x}](t)\}$ where $(s_1, \dots, s_n) : s$ is the arity of the lambda expression.
- $eval_{I,\sigma}(\{\overline{X}|\phi\}) = \{\overline{x} \mid \overline{x} \in D_{s_1} \times \dots \times D_{s_n} \text{ and } I, \sigma[\overline{X}/\overline{x}] \models \phi\}$ where $(s_1, \dots, s_n)$ is the arity of the set expression.
- for each first order or aggregate atom $F(t_1, \dots, t_n)$, $I, \sigma \models F(t_1, \dots, t_n)$ iff $(eval_{I,\sigma}(t_1), \dots, eval_{I,\sigma}(t_n)) \in F_I$
- the standard truth recursion for $\wedge, \vee, \neg, \rightarrow, \dots$

Given is a signature $\Sigma' = \Sigma \cup \Sigma_d$ (symbols of Σ (Σ_d) are called the interpreted symbols (respectively the defined predicate symbols)), we assume the existence of a Σ -structure I_o . It may interpret some of the sorts from Σ over the integers or reals. If Σ contains the standard arithmetic function and predicate symbols such as $+$, $-$, $\leq$ they are interpreted in the standard way. Other sorts represent user-defined sorts and are interpreted appropriately. Aggregate symbols are also part of the signature Σ and are interpreted by some fixed domain independent relation (e.g. the minimum relation over a poset, the cardinality of a set, sum, product, etc.). An analogy with database terminology can be drawn here: defined predicates correspond to intensional database predicates (IDB) while interpreted predicates correspond to extensional database predicates (EDB).

Let P be a Σ' -aggregate program and $\mathcal{I}_{I_o}$ be the set of all Σ' -structures extending I_o and interpreting the defined symbols. With these definitions, the immediate consequence operator of P can be defined in a straightforward way.

Definition 2.1 (2-valued T_P operator for aggregate programs). *We define the immediate consequence operator $T_P : \mathcal{I}_{I_o} \rightarrow \mathcal{I}_{I_o}$ of P as the operator mapping an interpretation I to $T_P(I) = I'$ such that for each defined symbol p , $p_{I'} = \{(x_1, \dots, x_n) \mid \text{there is a rule } p(t_1, \dots, t_n) \leftarrow B \text{ in } P \text{ and a variable assignment } \sigma \text{ such that } eval_{I,\sigma}(t_1, \dots, t_n) = (x_1, \dots, x_n) \text{ and } I, \sigma \models B\}$.*

Typically, the intended interpretation of a logic program P is given by one or some subset of fixpoints of the T_P operator. The semantics defined in the next sections is a fixpoint semantics. For each pair $I_1, I_2 \in \mathcal{I}_{I_o}$, define $I_1 \sqsubseteq I_2$ iff for each defined symbol p , $p_{I_1} \subseteq p_{I_2}$. A well-known phenomenon is that aggregates lead to T_P operators that are nonmonotone with respect to this order, even when the program p doesn't contain a negation symbol.

Example 2.5. Consider the program: $p(a) \leftarrow \text{Card}(\{X \mid p(X)\}, N) \wedge N < 2$.

The T_P of this program is non-monotone as it maps any interpretation of p with at least two elements to an interpretation assigning the empty set to p . The unique fixpoint of T_P interprets p by $\{a\}$. This seems a natural solution.

For aggregate programs with a $\sqsubseteq$ -monotone T_P operator, T_P has a least fixpoint. As argued in [12], it is a natural solution for the semantics of P^1 .

Example 2.6. Reconsider the company control program from Example 2.4. In the natural situation where the sort *shares* is interpreted with positive numbers, the T_P is monotone. Indeed, if we add new *Controls* facts to the structure, then the result of summing the shares of controlling companies can only increase and hence more *Controls* facts can be derived. It is well-known that the least fixpoint of this operator is the intended solution of this problem.

The examples above illustrate cases where the intended interpretation corresponds to a minimal fixpoint of the immediate consequence operator. In general, nonmonotone operators may have no or multiple (minimal) fixpoints. For pure logic programs, not all minimal fixpoints are intended interpretations. This can also happen with aggregate programs:

Example 2.7. Consider the program:

$$p(0) \leftarrow \text{Card}(\{X \mid p(X)\}, 1) \qquad q \leftarrow \neg p(0)$$

This program has two fixpoints. One in which p is interpreted by the singleton 0, and q is false, and another in which p is empty and q is true. Both are minimal, but clearly only the second is intended. The first fixpoint is not acceptable because $p(0)$ only depends positively on itself, hence, the model is not well-founded.

Approximation theory [5, 6] is a fixpoint theory that assigns fixpoints to any nonmonotone operator in a complete lattice and was used to describe the semantics of logic programs, default logic and autoepistemic logic. The next section recalls its basics and uses it to define a semantics for aggregate programs.

3 Approximation Theory

Let $\langle L, \leq \rangle$ be a complete² lattice with least element $\perp$ and largest element $\top$. The bilattice of $\langle L, \leq \rangle$ is the structure $\langle L^2, \leq, \leq_i \rangle$ where we define $(x, y) \leq (x_1, y_1)$ iff $x \leq x_1$ and $y \leq y_1$, and $(x, y) \leq_i (x_1, y_1)$ iff $x \leq x_1$ and $y \geq y_1$. Both $\leq$ and $\leq_i$ are complete lattice orders in L^2 . The order $\leq$ has least element $(\perp, \perp)$ and largest element $(\top, \top)$ whereas $\leq_i$ has least element $(\perp, \top)$ and largest element $(\top, \perp)$.

¹ Note that the approach of monotone aggregation is far more general than the case of programs with a $\sqsubseteq$ -monotone T_P operator. Indeed, in monotone aggregation, programs are considered for which T_P is monotone with respect to some user defined order. Such operators are not necessarily $\sqsubseteq$ -monotone.

² Each subset has a least upper bound and a greatest lower bound.

In approximation theory, the intuition underlying bilattice elements (x, y) is that they are *approximations* of lattice elements. In particular (x, y) approximates a lattice element z if $x \leq z \leq y$. The elements x and y constitute respectively an underestimate and overestimate of z . The set of lattice elements approximated by (x, y) is denoted $[x, y]$. Note that this interpretation is only possible for tuples (x, y) such that $x \leq y$. We call such tuples *consistent*. Inconsistent tuples do not approximate any lattice element ($[x, y]$ is empty). The order $\leq_i$ is called the information order; $(x, y) \leq_i (x_1, y_1)$ implies that (x_1, y_1) is a more precise approximation than (x, y) (in particular, $[x, y] \supseteq [x_1, y_1]$). Note that for tuples (x, x) , $[x, x] = \{x\}$. Therefore, the tuples of identical elements constitute the natural embedding of L in L^2 .

We define a framework for studying the fixpoints of any operator on a complete lattice by investigating the fixpoints of approximations.

Definition 3.1. Let $O : L \rightarrow L$ be an operator on L . An approximation $A : L^2 \rightarrow L^2$ of O is a $\leq_i$ -monotone operator such that:

- $A(x, x) = (O(x), O(x))$, that is, A coincides with O on the embedding of L in L^2 .
- A is symmetric, that is, if $A(x, y) = (x_1, y_1)$ then $A(y, x) = (y_1, x_1)$.

Every operator $A : L^2 \rightarrow L^2$ can be defined by a unique pair A_1, A_2 of functions of type $L^2 \rightarrow L$ such that for each (x, y) , $A(x, y) = (A_1(x, y), A_2(x, y))$. It is straightforward to see that A is an approximation of some operator iff (1) for each pair (x, y) , $A_1(x, y) = A_2(y, x)$ and (2) A_1 is monotone in its first argument and antimonotone in its second argument. Vice versa, each operator $A^1 : L^2 \rightarrow L$ which is monotone in its first and anti-monotone in its second argument can be used to construct an approximation A . A maps a tuple (x, y) to the tuple $(A^1(x, y), A^1(y, x))$ and approximates the operator $O : L \rightarrow L$ where for each x , $O(x) = A^1(x, x)$.

Let A be an approximation of O . Then A has a least fixpoint $\text{KK}(A)$ in the information order $\leq_i$, called the Kripke-Kleene fixpoint of A . The Kripke-Kleene fixpoint approximates each fixpoint of O . This is not good enough if we want to approximate minimal fixpoints of O (or some subset of them).

In order to obtain better approximations, we define a *stable operator* $ST_A : L \rightarrow L$. Observe that with each lattice element x , we can associate an $L \rightarrow L$ operator, denoted $A^1(., x)$ which maps elements z to $A^1(z, x)$. The operator $A^1(., x)$ is monotone and has a least fixpoint. Define $ST_A(x) = \text{lfp}(A^1(., x))$. ST_A is an anti-monotone operator. Fixpoints of ST_A are called *stable fixpoints* of A . It can be shown that they are minimal fixpoints of O . The *extended stable operator* $ST_A : L^2 \rightarrow L^2$ maps (x, y) to $(ST_A(y), ST_A(x))$ and is $\leq_i$ -monotone. It has a $\leq_i$ -least fixpoint $WF(A)$, called the well-founded fixpoint of A .

An operator O may have many different approximations. It was investigated in [6] how the fixpoints of these different approximations relate and how approximations can be constructed from O . Let L^c be the restriction of L^2 to the consistent pairs. We say that an approximation A_1 is *less precise* than approximation A_2 and denote by $A_1 \leq_i A_2$ if for each $(x, y) \in L^c$, $A_1(x, y) \leq_i A_2(x, y)$. The following properties have been proven:

Proposition 3.1. [6] *Let $A_1 \leq_i A_2$. Then: (1) A stable fixpoint of A_1 is a stable fixpoint of A_2 . (2) The Kripke-Kleene and well-founded fixpoint of A_1 are less precise (w.r.t $\leq_i$) than those of A_2 .*

Two important observations can be derived. First, two approximations coinciding on L^c have the same stable, Kripke-Kleene and well-founded fixpoints. The behaviour of the approximation on the inconsistent elements is not relevant! Hence, it suffices to define an approximation only on the consistent elements.

Definition 3.2 (Partial Approximation). *An operator $A : L^c \rightarrow L^c$ is a partial approximation of an operator $O : L \rightarrow L$ if A is $\leq_i$ -monotone and $A(x, x) = (O(x), O(x))$ for each lattice element x .*

It can be proven that any partial approximation can be extended to an approximation defined on L^2 . Hence, the sequel focuses on partial approximations.

Improving an approximation yields increasing sets of stable fixpoints and more precise Kripke-Kleene and well-founded fixpoints. There must be a limit to this process. One can construct a most precise partial approximation:

Definition 3.3 (Ultimate Approximation). *Let $O([x, y]) = \{O(z) | z \in [x, y]\}$. The ultimate partial approximation $Ult(O) : L^c \rightarrow L^c$ of $O : L \rightarrow L$ is:*

$$Ult(O)(x, y) = (glb(O([x, y])), lub(O([x, y])))$$

It is easy to show that $Ult(O)$ is $\leq_i$ monotone, coincides with O on the pairs (x, x) , and is more precise than any other partial approximation of O . Moreover, its fixpoints are determined only by O . Thus, for each operator O on $\langle L, \leq \rangle$, we define the *ultimate stable* (respectively *ultimate Kripke-Kleene* and *ultimate well-founded*) fixpoints of O as the stable (respectively Kripke-Kleene and well-founded) fixpoints of its ultimate partial approximation $Ult(O)$.

3.1 Approximations of Logic Programs

Approximation theory was used in [7] for describing the semantics of Logic Programming, Default logic and Autoepistemic logic. In the case of Logic Programming, the underlying lattice is the lattice of Herbrand interpretations. The bilattice corresponds to 4-valued interpretations. In particular, any pair (I, J) of 2-valued interpretations defines the following truth function:

- p is true in (I, J) iff $p \in I \cap J$; i.e. under- and overestimate agree on the truth of p .
- p is false in (I, J) iff $p \notin I \cup J$; i.e. under- and overestimate agree on the falsity of p .
- p is undefined in (I, J) iff $p \in J \setminus I$; i.e. underestimate of p is false and overestimate is true.
- p is inconsistent in (I, J) iff $p \in I \setminus J$; i.e. underestimate of p is true and overestimate is false.

The consistent bilattice elements (I, J) correspond exactly to the 3-valued interpretations (since $I \setminus J$ is empty).

In terms of the bilattice representation of 4-valued interpretations, the four-valued immediate consequence operator $\mathcal{T}_P$ of a program P [9] can be defined as the operator that maps any pair (I, J) to $(\mathcal{T}_P^1(I, J), \mathcal{T}_P^1(J, I))$. Here, the operator $\mathcal{T}_P^1(I, J)$ is defined as the 2-valued interpretation I' such that a ground atom p is true in I' iff there exists a ground instance of a ground rule $p \leftarrow B$ such that each positive literal of B is true in I and each negative literal is true in J .

The main result about the relationship between approximation theory and LP-semantics is that the 4-valued immediate consequence operator $\mathcal{T}_P$ of a program P is an approximation of the 2-valued immediate consequence operator T_P , that the stable operator of P is the stable operator of $\mathcal{T}_P$ and hence that the Kripke-Kleene, stable and well-founded models of P are the Kripke-Kleene, stable and well-founded fixpoints of $\mathcal{T}_P$.

In general, $\mathcal{T}_P$ is not the ultimate approximation³ of T_P , and consequently, the ultimate fixpoints of T_P are not the Kripke-Kleene, stable and well-founded models of P . Below are shown some important relationships between the standard models and the ultimate versions. Let P be a logic program.

Proposition 3.2. (a) *The well-founded (resp. Kripke Kleene) model of P is less precise than the ultimate well-founded (resp. Kripke Kleene) model of P .* (b) *If the well-founded model of P is 2-valued then it is the ultimate well-founded model of P .* (c) *If P_1, P_2 are two different programs such that their 2-valued immediate consequence operators coincide, then their ultimate Kripke-Kleene, ultimate well-founded and ultimate stable model coincide.*

Example 3.1. A simple example of a logic program for which the standard and ultimate semantics differ is the following:

$$P_1 = \{ p \leftarrow p \quad p \leftarrow \neg p \}$$

Its Kripke-Kleene and well-founded fixpoint is $(\{\}, \{p\})$ (i.e. p is undefined) and it has no stable models. On the other hand, note that the 2-valued immediate consequence operators of this program is monotone and that it coincides with the immediate consequence operator of the program

$$P_2 = \{ p \leftarrow \}$$

Hence, they have the same ultimate models. Since the Kripke-Kleene model of P_2 is the 2-valued interpretation $\{p\}$, this is also the ultimate Kripke-Kleene, ultimate well-founded and the unique ultimate stable model of P_1 .

³ It can be seen easily that the 3-valued ultimate approximation is obtained by evaluating rule bodies, not according to the standard 3-valued truth function but using the technique of *supervaluations*. In this scheme, a formula is true(false) in a 3-valued interpretation I if it is true(false) in each 2-valued interpretation approximated by I . Otherwise, the formula is undefined.

3.2 Ultimate Semantics for Programs with Aggregates

Given is a signature $\Sigma' = \Sigma \cup \Sigma_d$, a Σ -structure I_o for the interpreted symbols Σ , and an aggregate program with defined predicates Σ_d . The structure $\langle \mathcal{I}_{I_o}, \sqsubseteq \rangle$ of Σ' -structures extending I_o is a complete lattice. The operator T_P is an operator in this lattice. Hence, we can apply ultimate approximation theory.

Definition 3.4. *The ultimate Kripke-Kleene, ultimate well-founded and ultimate stable models of P are the ultimate Kripke-Kleene, ultimate well-founded and ultimate stable fixpoints of T_P .*

4 Analysis and Results

In this section, we investigate the ultimate semantics of aggregate programs. Everywhere in this section, we assume the existence of a given signature Σ , the Σ -structure I_o and an aggregate program P based on $\Sigma' = \Sigma \cup \Sigma_d$.

The first result shows that if the immediate consequence operator is monotone, then the ultimate well-founded model is the least fixpoint of this operator.

Theorem 4.1. *[6] If T_P is $\sqsubseteq$ -monotone then its ultimate well-founded and unique stable fixpoint are equal to the least fixpoint of T_P .*

Note that this property is not satisfied by the well-founded semantics. Example 3.1 illustrates this. The 2-valued T_P operator of the program P_1 is constant and hence monotone, yet its least fixpoint is not the well-founded model of P_1 .

Aggregate stratified programs are studied in [1, 14]. P is an aggregate stratified program iff the body B of any rule is a conjunction of atoms, negative literals and aggregate atoms. Moreover, for each predicate symbol p there is a level number i_p such that for each rule $p \leftarrow B$, if predicate symbol q occurs in an atom of B then $i_q \leq i_p$, if q appears in a negative literal in B or in a set expression in an aggregate atom in B , then $i_q < i_p$. In other words, predicate symbols appearing negatively or in an aggregate atom in the body of a rule defining p must be defined in lower levels.

The next theorem expresses that the ultimate well-founded (and stable) semantics of aggregate programs generalises the perfect model semantics of aggregate stratified programs.

Theorem 4.2. *If P is a aggregate stratified program then its ultimate well-founded model is its perfect model [1, 14].*

Another theorem shows that each aggregate program has a natural translation into a ground, infinitary logic program, such that the ultimate well-founded model of the aggregate program is the well-founded model of its translation.

Let $\mathcal{B}_o$ be the set $\{p(x_1, \dots, x_n) \mid p \in \Sigma_d \text{ and } x_1, \dots, x_n \in D_{s_1} \times \dots \times D_{s_n} \text{ (with } (s_1, \dots, s_n) \text{ the arity of } p)\}$. For any 2-valued interpretation I extending I_o and any set B of positive or negative literals of $\mathcal{B}_o$, define that $I \models B$ iff $(x_1, \dots, x_n) \in p_I$ if $p(x_1, \dots, x_n) \in B$ and $(x_1, \dots, x_n) \notin p_I$ if $\neg p(x_1, \dots, x_n) \in B$.

Definition 4.1. Define $lp(P)$ as the (infinitary) ground logic program based on $\mathcal{B}_o$ and consisting of all ground clauses $p(x_1, \dots, x_n) \leftarrow B$ for which there exists a variable assignment σ and a rule $p(t_1, \dots, t_n) \leftarrow F$ in P such that:

- $p(x_1, \dots, x_n) = p(eval_{I, \sigma}(t_1), \dots, eval_{I, \sigma}(t_n))$ and
- B is any set of literals of $\mathcal{B}_o$ such that for each 2-valued Σ -interpretation I extending I_o : if $I \models B$ then $I, \sigma \models F$.

Theorem 4.3. The ultimate well-founded model of P is the well-founded model of $lp(P)$.

Finally, we return to the examples of Section 2.1. Since the immediate consequence operator of company control program is monotone (if shares are positive), the following proposition follows.

Proposition 4.1. For any finite structure I_o interpreting `owns_stock` and interpreting the sort `shares` by positive real numbers, the ultimate well-founded model is 2-valued and is the unique stable model. Moreover, a fact `Controls`(c_1, c_2) belongs to this model iff a company c_1 controls a company c_2 .

The ultimate well-founded model of the shortest path problem correctly represents shortest path distances.

Proposition 4.2. Let I_o be a structure for the shortest path program P_{sp} defining a weighted directed graph with positive weights. Then the ultimate well-founded model of P_{sp} is 2-valued and contains a fact `sp`(x, y, w) iff there exists a path between x and y with total weight w and the total weight of all other paths between x and y is greater or equal to w .

5 Related Work

The following toy-problems illustrate some issues in the relationship between ultimate well-founded semantics and monotone aggregation [17].

In the example below, $p/1$ is a predicate interpreted over the domain $\{0, 1, 2\}$.

$$p(X) \leftarrow Min(\{Y | Y = 1 \vee p(Y)\}, X)$$

Intuitively, the intended meaning is $p_I = \{p(1)\}$. However, note that $p_{I'} = \{p(0)\}$ is a fixpoint too. Note that the immediate consequence operator of this program is not monotone with respect to the standard order $\sqsubseteq$ on interpretations. However, it is monotone with respect to other orders. In approaches of monotone aggregation, one must find such an appropriate order. In monotone aggregation, only structures that assign a singleton set to p are considered. In this collection of structures, one can define the following order: $I \leq J$ if it holds that if $p_I = \{x\}$ and $p_J = \{y\}$, then $x \leq y$. T_P is monotone with respect to this order. However, its minimal fixpoint assigns $p_I = \{0\}$. T_P is also monotone with respect to the inverse order $\geq$, and its least fixpoint in this order is the intended solution $p_I =$

$\{1\}$. This shows that the selection of the right order is important in monotone aggregation. Consider now the following extension:

$$\begin{aligned} p(X) &\leftarrow \text{Min}(\{Y | Y = 1 \vee p(Y)\}, X) \\ r(X) &\leftarrow \text{Min}(\{Y | Y = 1 \vee r(Y)\}, Z) \wedge X = 2 - Z \end{aligned}$$

The immediate consequence operator is not longer monotone with respect to $\geq$. The above program has the same T_P operator as the following program (obtained by removing redundant rules from $\text{lp}(P)$):

$$\begin{aligned} p(0) &\leftarrow p(0) & r(2) &\leftarrow r(0) \\ p(1) &\leftarrow \neg p(0) & r(1) &\leftarrow \neg r(0) \end{aligned}$$

This program is stratified and has a 2-valued well-founded model $\{p(1), r(1)\}$. It is the ultimate well-founded model of the original program.

The above examples suggest that the approach developed in this paper might be both more general and simpler than monotone aggregation. A formal investigation of the relationship between ultimate semantics and monotone aggregation needs to be conducted.

Kemp and Stuckey [12] define a well-founded semantics of logic programs with aggregates. However, their definition doesn't deal well with positive recursion through aggregation. Hence, their semantics is strictly weaker than the ultimate well-founded semantics defined in this paper. For example, in the well-founded model of [12] of the following program

$$p(0) \leftarrow \text{Card}(\{X | p(X)\}, 1)$$

the atom p is undefined while it is false in the ultimate well-founded model.

In [18] was proposed another extension of the well-founded and valid semantics for programs with aggregates. The extension is more in the spirit of the ultimate well-founded semantics. The authors rely on having aggregate functions defined for 3-valued multisets and satisfying certain properties of monotonicity. However, we have shown in this paper (the definition of ultimate approximations) that for defining 3-valued (or even 4-valued) semantics of programs with aggregates one can still work with aggregate functions defined on standard sets. Further investigation is needed about the precise relation of these two approaches.

Dix and Osorio [8] have also observed that the standard well-founded semantics is not strong enough for programs with aggregation. They have pointed out that the WFS^+ (introduced independently by Dix and Schlipf) is well-suited, however its complexity is on the first level of the polynomial hierarchy. In their paper [8] they propose two weaker extensions of the well-founded semantics WFS^1 and WFS^2 which are polynomially computable. The common property for these two semantics is that a is true for a program containing the single rule $a \leftarrow \neg a$. For this program, a is undefined in the ultimate well-founded semantics. Whether the ultimate well-founded semantics is weaker in general than WFS^1 and WFS^2 is an open question.

[12] also defines a stable semantics of programs with aggregates where the aggregate atoms are treated like negative literals during the stability transformation. As the authors have pointed out, this has the effect that a program may have non minimal stable models. In the ultimate well-founded and stable semantics for aggregate programs, such problems do not appear.

Another extension of the stable model semantics for aggregate programs [16] was done in the context of the **smodels** system [15]. The authors consider weight constraints of the form $L \leq \{l_1 = w_1, \dots, l_n = w_n\} \leq U$ where l_i are literals and w_i are real numbers representing weights. Such constraint is true in an interpretation if the sum of the weights of the literals satisfied by the interpretation is between L and U . For details about how stable models of programs with weight constraints are computed we refer to [16].

To show the relationship with our semantics we can translate a weight constraint to an aggregate formula of the form

$$\exists S. \text{Sum}(\{I \mid l_1 \wedge I = 1 \vee \dots \vee l_n \wedge I = n\}, \lambda(I)w(I), S) \wedge L \leq S \wedge S \leq U$$

where $w(I)$ is a new interpreted function which corresponds to the weights w_i . For programs with weight constraints only in the body we can show that stable models are ultimate stable models of the aggregate program obtained by applying the above transformation. In most cases the inverse will also hold but not always, as Example 3.1 shows. The advantage of using a special form of aggregate expressions, like weight constraints, is that more efficient algorithms can be developed. In the case of the **smodels** system, the complexity of computing stable models remains in the same class as for logic programs without aggregates [16]. The reason for this is the use of a less precise approximation underlying **smodels**.

In [3], ID-logic is defined as a logic extending first order logic with (*general nonmonotone inductive*) *definitions*. Each definition can be thought of as one logic program defining a subset of predicates. Using the ideas presented here, one can extend ID-logic with aggregates. A theory in this extended logic consists of *aggregate formulas* and *aggregate programs* as defined here. A model of such a theory is a 2-valued interpretation I that satisfies all aggregate formulas and is an ultimate well-founded model of each of its aggregate programs.⁴

6 Conclusion and Future Work

In this paper we have used the Approximation Theory to develop a well-founded and stable model semantics for programs with aggregates. We do this by defining a 2-valued T_P operator for programs with aggregates in an intuitive way. Approximation theory then shows how to define a most precise approximation of this operator, called an ultimate approximation. We argued that the well-founded and stable fixpoints of this ultimate approximation are well suited for

⁴ Note that an aggregate program may have many ultimate well-founded models if the interpretation I_o is not fixed but may vary.

programs with aggregation. In particular, if the T_P operator of a programs is monotone then the ultimate well-founded model is 2-valued and coincides with the least fixpoint of the T_P operator.

Another important topic for further research is to investigate the complexity of computing the ultimate well-founded semantics for programs with aggregates. We expect that if we restrict to Datalog programs, we will obtain exponential complexity in terms of the number of ground atoms. This is because a one step computation of the ultimate approximation on a 3-valued interpretation (I, J) requires to apply the T_P operator to all possible 2-valued interpretations I' such that $I \subseteq I' \subseteq J$. If the size of the Herbrand base is n then for the first step, there are 2^n possible interpretations. However, Approximation Theory is an interesting setting for reducing the complexity. Indeed, note that the ultimate well-founded model is approximated by the well-founded model of any approximating operator. Less precise alternative approximation operators with lower complexity can be searched for such that their well-founded fixpoints coincide with the ultimate well-founded model for certain collections of well-behaved programs. An example where this is possible is the class of aggregate stratified programs.

References

1. M. P. Consens and A. O. Mendelzon. Low-complexity aggregation in GraphLog and Datalog. *Theoretical Computer Science*, 116(1):95–116, 1993.
2. M. Denecker. The well-founded semantics is the principle of inductive definition. In J. Dix, L. F. del Cerro, and U. Furbach, editors, *Logics in Artificial Intelligence, European Workshop*, volume 1489 of *Lecture Notes in Computer Science*, pages 1–16, Dagstuhl, Germany, 1998. Springer.
3. M. Denecker. Extending classical logic with inductive definitions. In J. Lloyd et al., editors, *1st International Conference on Computational Logic*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 703–717, London, U.K., July 2000. Springer.
4. M. Denecker, M. Bruynooghe, and V. W. Marek. Logic programming revisited: logic programs as inductive definitions. *ACM Transactions on Computational Logic*, 2001. accepted.
5. M. Denecker, V. Marek, and M. Truszczyński. Approximating operators, stable operators, well-founded fixpoints and applications in non-monotonic reasoning. In J. Minker, editor, *Logic-based Artificial Intelligence*, pages 127–144. Kluwer Academic Publishers, 2000.
6. M. Denecker, V. Marek, and M. Truszczyński. Ultimate approximations. Report CW 320, Department of Computer Science, K.U.Leuven, Belgium, Sept. 2001.
7. M. Denecker, V. W. Marek, and M. Truszczyński. Uniform semantic treatment of default and autoepistemic logics. In A. Cohn, F. Giunchiglia, and B. Selman, editors, *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning*, pages 74–84, 2000.
8. J. Dix and M. Osorio. On well-behaved semantics suitable for aggregation. Technical Report TR 11/97, University of Koblenz, Department of Computer Science, Rheinau 1, Apr. 1997. Accepted at ILPS '97 as poster paper.
9. M. Fitting. Bilattices and the semantics of logic programming. *The Journal of Logic Programming*, 11(1,2):91–116, 1991.

10. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming, Proceedings of the 5th International Conference and Symposium*, pages 1070–1080, Seattle, Washington, 1988. MIT Press.
11. A. C. Kakas, R. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, Dec. 1992.
12. D. B. Kemp and P. J. Stuckey. Semantics of logic programs with aggregates. In V. A. Saraswat and K. Ueda, editors, *Proceedings of the International Logic Programming Symposium*, pages 387–401. MIT Press, 1991.
13. V. W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K. R. Apt, V. W. Marek, M. Truszczyński, and D. S. Warren, editors, *The Logic Programming Paradigm: A 25-Year Perspective*, pages 375–398. Springer, 1999.
14. I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In D. McLeod, R. Sacks-Davis, and H.-J. Schek, editors, *16th International Conference on Very Large Data Bases*, pages 264–277. Morgan Kaufmann, 1990.
15. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.
16. I. Niemelä, P. Simons, and T. Soininen. Stable model semantics of weight constraint rules. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *Proceedings of the Fifth International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 1730 of *Lecture Notes in Computer Science*, pages 317–331. Springer-Verlag, 1999.
17. K. A. Ross and Y. Sagiv. Monotonic aggregation in deductive databases. *Journal of Computer and System Sciences*, 54(1):79–97, 1997.
18. S. Sudarshan, D. Srivastava, R. Ramakrishnan, and C. Beeri. Extending the well-founded and valid semantics for aggregation. In D. Miller, editor, *Proceedings of the International Logic Programming Symposium*, pages 590–608. MIT Press, 1993.
19. A. Van Gelder. The well-founded semantics of aggregation. In *Proceedings of the Eleventh ACM Symposium on Principles of Database Systems*, pages 127–138. ACM Press, 1992.
20. A. Van Gelder. Foundations of aggregation in deductive databases. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Third International Conference on Deductive and Object-Oriented Databases*, volume 760 of *Lecture Notes in Computer Science*, pages 13–34. Springer-Verlag, 1993.
21. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
22. B. Van Nuffelen and M. Denecker. Problem solving in ID-logic with aggregates: some experiments. In A. K. Mark Denecker, editor, *Eight International Workshop on Nonmonotonic Reasoning, special track on Abductive Reasoning*, Breckenridge, Colorado, USA, 2000.