Implementing Operator-Based Semantics for Normal Logic Programs using Meta-Programming in ASP

Jesse Heyninck[1,3], Spencer Killen[2] and Jeroen Spaans[3]

[1]University of Cape Town, South Africa
[2]University of Alberta, Canada
[3]Open Universiteit, Heerlen, Limburg, the Netherlands

Abstract

Approximation Fixpoint Theory (AFT) provides a unifying semantic framework for a wide range of non-monotonic logics, including many dialects of logic programming. Despite its conceptual elegance, practical implementations of AFT-based semantics remain limited. In this paper, we present an implementation of AFT using meta-programming in Answer Set Programming (ASP). We show how approximators and different kinds of fixpoints can be encoded at the meta-level, allowing existing ASP solvers to compute AFT-based semantics for normal logic programs without modifying the solver itself.

1. Introduction

Semantics of various formalisms for knowledge representation can often be described by fixpoints of corresponding operators. For example, in logic programming, default logic or formal argumentation, all the major semantics can be formulated as different types of fixpoints of the same operator [1, 2]. For example, in logic programming, the immediate consequence operator can be used as a basis to derive most common semantics. Such operators are usually non-monotonic, and so their fixpoints are not guaranteed to exist or be found constructively.

In order to deal with this 'elusive nature' of the fixpoints, [1] introduced a method for *approximating* each value x of the underlying operator by a pair of elements (h, t). These elements intuitively represent lower and upper bounds on z, and so a corresponding *approximator* for the original, non-monotonic operator, is constructed. If the approximator that is obtained is precision-monotonic, intuitively meaning that more precise inputs of the operator give rise to more precise outputs, then the approximator has fixpoints that can be constructively computed, and which in turn approximate the fixpoints of the approximated operator, if such fixpoints exist. It was shown that many formalisms admit a characterisation in terms of AFT [3, 2, 4, 5, 6, 7], including logic programming [8, 9, 10, 11, 12, 13, 14, 15, 16].

Until now, research into AFT has been largely theoretical, even though theoretical insights have also percolated in the realm of implementations [17, 18]. However, a unified library for AFT-based semantics has, to the best of our knowledge, not been developed hitherto. In this paper, we take a first step towards the goal of an implementation of approximation fixpoint theory by developing an ASP-based library for AFT-based semantics for normal logic programs. We do this by making use of the meta-programming capabilities of clingo [19], as this allows to make use of the grounder gringo and formulate logic programs in the original ASP-syntax. We also provide a tool that automatically generates several semantic transformations underlying AFT-based semantics and combines several calls to clingo.

The main contribution of this paper is thus a library and tool based on answer set programming in which all major semantics based on AFT using the Fitting and ultimate operator for normal logic programs are implemented. Both the library and tool are modular, so that alternative operators can be straightforwardly implemented. This has as benifits that: (1) it allows researchers in AFT to implement

Joint Proceedings of the Workshops and Doctoral Consortium of the 42nd International Conference on Logic Programming (ICLP 2026), co-located with the Federated Logic Conference (FLoC 2026), July 18–19 and 24–25, 2026, Lisbon, Portugal.



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-03

CEUR-WS.org/Vol-4311/ASPOCP_regular_08.pdf

concepts and thus validate their theoretical work; (2) it opens up a host of new semantics for solving logic programs, which might be more apt than the stable semantics in certain application contexts; (3) it can be used for people getting into AFT, as they can use the library to learn AFT from another, more practical angle or validate examples.
Outline and Main Contributions: Necessary preliminaries on logic programming are introduced in Section 2. We explain how we implemented semantics based on the Fitting operator in Section 3, and move to an implementation of the ultimate operator in Section 4. We report on some preliminary experiments in Section 5. We conclude in Section 6.

2. Background and Preliminaries

In this section, we recall the necessary preliminaries on the syntax of logic programming (Section 2.1), AFT (Section 2.2), AFT-based semantics for normal logic programs (Section 2.3) and meta-programming (Section 2.4). Since the goal of this paper is to implement semantics for (normal) logic programs using answer set programming, which, syntactically speaking, has normal logic programs as a subfragment. We therefore have chosen to first present the full syntax of answer set programs (Section 2.1), and then present the semantics for the fragment of normal logic programs in Section 2.3 (for which we need background on AFT in Section 2.2).

2.1. Syntax of Answer Set Programs

Answer Set Programming (ASP) [20] is a declarative problem solving approach targeted at difficult search problems. In ASP, a program $\mathcal{P}$ consists of rules of the form:

$$\mathtt{b_1} | \ldots | \mathtt{b_o} \texttt{:-} \mathtt{a_1}, \ldots, \mathtt{a_m}, \texttt{not } \mathtt{a_{m+1}}, \ldots, \texttt{not } \mathtt{a_n}$$

where each $\mathtt{a_i}$ and $\mathtt{b_i}$ is an atom, "not" represents negation as failure, and | represents disjunction. An atom has the form $p(\mathtt{t}_1, \ldots, \mathtt{t}_n)$, where p is a predicate symbol, and $\mathtt{t}_1, \ldots, \mathtt{t}_n$ are terms. Terms are either constants, variables, arithmetic terms (i. e., $-\mathtt{t}_1$ or $\mathtt{t}_1 \circ \mathtt{t}_2$ with $\circ \in \{+, -, *, /\}$ wrt. some terms $\mathtt{t}_1, \mathtt{t}_2$), or functional terms (i. e., $\varphi(\mathtt{t}_1, \ldots, \mathtt{t}_m)$ with φ a functor, $\mathtt{t}_1, \ldots, \mathtt{t}_m$ terms, and $m > 0$) [21]. Moreover, we express the arity n of a predicate or function φ as φ/n, and an ASP literal ℓ is either an atom or a default-negated atom. Intuitively, a head atom of the rule ($\mathtt{b}_1 \ldots \mathtt{b}_o$) is derived if all positive literals in the body ($\mathtt{a}_1, \ldots, \mathtt{a}_m$) are true and all negated literals ($\texttt{not } \mathtt{a}_{m+1}, \ldots, \texttt{not } \mathtt{a}_n$) are not proven true. Informally, an answer set can be seen as a minimal set of literals satisfying all program rules without cyclic support (we will come to the exact definition in Section 2.3). Multiple answer sets represent different possible solutions to the encoded problem. Solvers like clingo [19] can automatically calculate these answer sets. If a rule has an empty body, it is called a *fact*, and if its head is empty, it is called a *constraint*. An atom/rule/program is called *ground* if it contains no variables, and a rule/program is called normal if o=1.

We furthermore make use of some additional constructs. *choice rule*, are rules of the form:

$$\mathtt{l}\{\mathtt{a_1}, \ldots, \mathtt{a_m}\}\mathtt{u} \texttt{:-} \mathtt{a_{m+1}}, \ldots, \mathtt{a_n}, \texttt{not } \mathtt{a_{n+1}}, \ldots, \texttt{not } \mathtt{a_o}$$

with $0 \leq m \leq n \leq o$, $a_1, \ldots, a_o$ being atoms, $0 \leq l \leq u$, and l, u being (optional) lower and upper bounds, respectively. Intuitively, any subset of the head atoms (which complies with the upper and lower bound, if specified) can be included in the answer set (if the body is true). We use *aggregates* and *optimization statements*. The former are used to reason about minima, maxima, sums, and counts over sets of literals. First, an *aggregate element* g is defined as $g = \mathtt{t}_1, \ldots, \mathtt{t}_m : \ell_1, \ldots, \ell_n$ with $\mathtt{t}_1, \ldots, \mathtt{t}_m$ terms, and $\ell_1, \ldots, \ell_n$ literals. An *aggregate* is then defined as $\#\mathtt{agg}\{g_1, \ldots, g_n\} \prec \mathtt{t}$, with $\#\mathtt{agg} \in \{\#\mathtt{count}, \#\mathtt{min}, \#\mathtt{max}, \#\mathtt{sum}\}$ an aggregate function name, $g_1, \ldots, g_n$ aggregate elements, $\prec \in \{<, >, \leq, \geq, =, \neq\}$ an aggregate relation, and $\mathtt{t}$ a term [21]. Optimization statements allow to select answer sets that are minimal w.r.t. a defined cost function. We only use a specific form of *minimize statements*, which are of the form $\texttt{\#minimize}\{\mathtt{t}_1, \ldots, \mathtt{t}_m : \ell_1, \ldots, \ell_n\}.$, where $\mathtt{t}_1, \ldots, \mathtt{t}_m$ are terms and $\ell_1, \ldots, \ell_n$ are literals. A set compliant with the minimization is *optimal*.

2.2. Approximation Fixpoint Theory

The basic idea behind approximation fixpoint theory is that the semantics of a (non-monotonic) formalism can be expressed in terms of (selected) fixpoints of an operator $\mathcal{O}$. For example, in logic programming, most common semantics can be characterised in terms of fixpoints of the Fitting-operator $\mathcal{IC}_\mathcal{P}$ (see Section 2.3). Depending on the application, certain kinds of fixpoints are of interest (e.g. the *stable* fixpoints), as we will explain below.

We recall some necessary preliminaries on set theory and (sub-)lattices. A lattice is a partially ordered set $L = \langle \mathcal{L}, \leq \rangle$ where every two elements $x, y \in \mathcal{L}$ have a least upper $x \sqcup y$ and a greatest lower bound $x \sqcap y$. A lattice is complete if every set $X \subseteq \mathcal{L}$ has a least upper (denoted $\sqcup X$) and a greatest lower bound (denoted $\sqcap X$).

We now recall basic notions from approximation fixpoint theory (AFT), as described by [1]. We will illustrate these notions in Section 2.3.

Given a lattice $L = \langle \mathcal{L}, \leq \rangle$, we let $L^2 = \langle \mathcal{L}^2, \leq_i, \leq_t \rangle$ be the structure (called *bilattice*), in which $\mathcal{L}^2 = \mathcal{L} \times \mathcal{L}$, and for every $h_1, t_1, h_2, t_2 \in \mathcal{L}$, (1) $(h_1, t_1) \leq_i (h_2, t_2)$ if $h_1 \leq h_2$ and $t_1 \geq t_2$; and $(h_1, t_1) \leq_t (h_2, t_2)$ if $h_1 \leq h_2$ and $t_1 \leq t_2$.

An *approximator* $\mathcal{O} : \mathcal{L}^2 \to \mathcal{L}^2$ of an operator $O : \mathcal{L} \to \mathcal{L}$ is an operator that maps every approximation (h, t) of an element z to an approximation (h', t') of another element $O(z)$, thus approximating the behavior of the approximated operator O.

Definition 1. *Let $O : \mathcal{L} \to \mathcal{L}$ and $\mathcal{O} : \mathcal{L}^2 \to \mathcal{L}^2$. (1) $\mathcal{O}$ is* $\leq_i$-monotonic*, if when $(h_1, t_1) \leq_i (h_2, t_2)$, also $\mathcal{O}(h_1, t_1) \leq_i \mathcal{O}(h_2, t_2)$; (2) $\mathcal{O}$ is* approximating*, if it is $\leq_i$-monotonic and for any $z \in \mathcal{L}$, $(\mathcal{O}(z, z))_1 = (\mathcal{O}(z, z))_2$; (3) $\mathcal{O}$ is an* approximation *of O, if it is $\leq_i$-monotonic and $\mathcal{O}$* extends *O, that is: $(\mathcal{O}(z, z))_1 = (\mathcal{O}(z, z))_2 = O(z)$.*

To avoid clutter, we will also denote $(\mathcal{O}(h, t))_1$ by $\mathcal{O}_l(h, t)$ and $(\mathcal{O}(h, t))_2$ by $\mathcal{O}_u(h, t)$.

The *stable operator*, defined next, is used for expressing the semantics of many non-monotonic formalisms and filters out self-supporting conclusions (see e.g. Example 1). Given a complete lattice $L = \langle \mathcal{L}, \leq \rangle$, let $\mathcal{O} : \mathcal{L}^2 \to \mathcal{L}^2$ be an approximator. $\mathcal{O}_l(\cdot, t) = \lambda h.\mathcal{O}_l(h, t)$, i.e.: $\mathcal{O}_l(\cdot, t)(h) = \mathcal{O}_l(h, t)$ (and similarly for the upper bound operator $\mathcal{O}_u$). The *stable operator for* $\mathcal{O}$ is: $S(\mathcal{O})(h, t) = (\text{lfp}(\mathcal{O}_l(\cdot, t)), \text{lfp}(\mathcal{O}_u(h, \cdot)))$. We also denote the components $\text{lfp}(\mathcal{O}_l(\cdot, t))$ and $\text{lfp}(\mathcal{O}_u(h, \cdot))$ of the stable operator by $C(\mathcal{O}_l)(t)$ respectively $C(\mathcal{O}_u)(h)$.

Stable operators capture the idea of minimizing truth, since for any $\leq_i$-monotonic operator $\mathcal{O}$ on $\mathcal{L}^2$, fixpoints of the stable operator $S(\mathcal{O})$ are $\leq_t$-minimal fixpoints of $\mathcal{O}$ [1, Theorem 4]. Altogether, we obtain the following notions (in addition to the notion of fixpoint):

Definition 2. *Given a complete lattice $L = \langle \mathcal{L}, \leq \rangle$, let $\mathcal{O} : \mathcal{L}^2 \to \mathcal{L}^2$ be an approximating operator. We call: (1) (x, y) a* Kripke-Kleene fixpoint *of $\mathcal{O}$ if $(x, y) = \text{lfp}_{\leq_i}(\mathcal{O})$; (2) (x, y) a* three-valued stable *fixpoint of $\mathcal{O}$ if $(x, y) = S(\mathcal{O})(x, y)$; (3) (x, x) a* total stable fixpoints *of $\mathcal{O}$ if $(x, x) = S(\mathcal{O})(x, x)$; (4) (x, y) the* well-founded fixpoint *of $\mathcal{O}$ if it is the $\leq_i$-minimal (three-valued) stable fixpoint of $\mathcal{O}$.*

2.3. Semantics of Logic Programming

In this section we recall the AFT-based semantics for ground normal logic programs. We assume that the atoms may include the propositional constants T (representing truth), F (falsity), U (unknown), and C (contradictory information). The set of atoms occurring in $\mathcal{P}$ is denoted $\mathcal{A}_\mathcal{P}$.

Central is the four-valued bilattice in Figure 1. We also assume a $\leq_t$-involution $-$ on $\leq_t$ (i.e, $-\mathsf{F} = \mathsf{T}$, $-\mathsf{T} = \mathsf{F}$, $-\mathsf{U} = \mathsf{U}$ and $-\mathsf{C} = \mathsf{C}$). A *four-valued interpretation* of a program $\mathcal{P}$ is a pair (h, t), where $h \subseteq \mathcal{A}_\mathcal{P}$ is the set of the atoms that are assigned a value in $\{\mathsf{T}, \mathsf{C}\}$ and $t \subseteq \mathcal{A}_\mathcal{P}$ is the set of atoms assigned a value in $\{\mathsf{T}, \mathsf{U}\}$. The intuition here is that h (t) is a lower (upper) approximation of the true atoms. Interpretations are compared by the *information order* $\leq_i$, where $(h_1, t_1) \leq_i (h_2, t_2)$ iff $h_1 \subseteq h_2$ and $t_2 \subseteq t_1$ (sometimes called "precision" order), and by the *truth order* $\leq_t$, where $(h_1, t_1) \leq_t (h_2, t_2)$ iff $h_1 \subseteq h_2$ and $t_1 \subseteq t_2$ (increased 'positive' evaluations). Truth assignments to complex formulas are then recursively defined as follows:

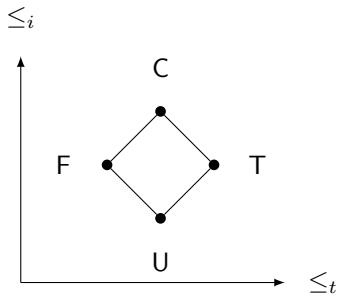


Figure 1: Bilattice and truth assignments.

- $(h,t)(p) = \begin{cases} \mathsf{T} & \text{if } p \in h \text{ and } p \in t, \\ \mathsf{U} & \text{if } p \notin h \text{ and } p \in t, \\ \mathsf{F} & \text{if } p \notin h \text{ and } p \notin t, \\ \mathsf{C} & \text{if } p \in h \text{ and } p \notin t. \end{cases}$
- $(h,t)(\texttt{not}\ \phi) = -(h,t)(\phi)$,
- $(h,t)(\psi,\phi) = \sqcup_{\leq_t}\{(h,t)(\phi),(h,t)(\psi)\}$,

A four-valued interpretation of the form (h,h) may be associated with *total* (sometimes also called *two-valued*) interpretation h, in which for an atom p, $h(p) = \mathsf{T}$ if $p \in h$ and $h(p) = \mathsf{F}$ otherwise. We say that (h,t) is a *three-valued* (or *consistent*) interpretation, if $h \subseteq t$ (i.e. there are no C-assignments).

We consider semantics for normal logic programs (NLP's), by first recalling the two-valued immediate consequence operator $IC_{\mathcal{P}}$ [22] and the four-valued Fitting-operator $\mathcal{IC}_{\mathcal{P}}$ [23] approximating it.

Definition 3. *Given an NLP $\mathcal{P}$ and a total interpretation t, we define:*

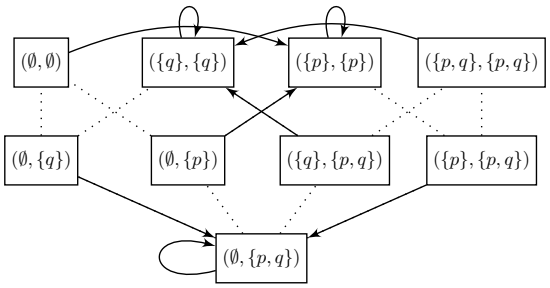
$$IC_{\mathcal{P}}(t) = \{p \in \mathcal{A}_{\mathcal{P}} \mid p \leftarrow \psi \in \mathcal{P}, (t,t)(\psi) = \mathsf{T}\}.$$

For a four-valued interpretation (h,t), we define $\mathcal{IC}_{\mathcal{P}}(h,t) = (\mathcal{IC}^l_{\mathcal{P}}(h,t), \mathcal{IC}^u_{\mathcal{P}}(h,t))$ with:

$$\mathcal{IC}^l_{\mathcal{P}}(h,t) = \{p \mid p \leftarrow \psi \in \mathcal{P}, (h,t)(\psi) \in \{\mathsf{T},\mathsf{C}\}\} \text{ and } \mathcal{IC}^u_{\mathcal{P}}(h,t) = \{p \mid p \leftarrow \psi \in \mathcal{P}, (h,t)(\psi) \in \{\mathsf{U},\mathsf{T}\}\}$$

Thus, $IC_{\mathcal{P}}$ is an operator on the lattice $\langle 2^{\mathcal{A}}, \subseteq\rangle$ that derives all heads of rules with true bodies. The operator $IC_{\mathcal{P}}$ (which is potentially non-monotonic), is approximated by the approximator $\mathcal{IC}_{\mathcal{P}}$ over the bilattice of four-valued interpretations.

Example 1. *Consider the logic program $\mathcal{P} = \{p\texttt{:-not}\ q;\ q\texttt{:-}q.\}$. Then the two-valued operator $IC_{\mathcal{P}}$ operates as follows: $IC_{\mathcal{P}}(\emptyset) = IC_{\mathcal{P}}(\{p\}) = \{p\}$ and $IC_{\mathcal{P}}(\{q\}) = IC_{\mathcal{P}}(\{p,q\}) = \{q\}$. The Fitting operator, on the other hand, displays the following behavior:*



We see that the Kripke-Kleene model is $(\emptyset, \{p, q\})$, which is, of course, also a fixpoint of $\mathcal{IC}_\mathcal{P}$. $\mathcal{IC}_\mathcal{P}$ has two other fixpoints, $(\{p\}, \{p\})$ and $(\{q\}, \{q\})$, which are in fact total fixpoints.

Constructing e.g. $\mathcal{S}(\mathcal{O})(\emptyset, \{p, q\}) = (\emptyset, \{p\})$ is done as follows: (1) $C(\mathcal{IC}^l_\mathcal{P})(\{p, q\}) = \emptyset$ since $\mathcal{IC}^l_\mathcal{P}(\emptyset, \{p, q\}) = \emptyset$ (and thus a fixpoint is reached). (2) $C(\mathcal{IC}^u_\mathcal{P})(\emptyset) = \{p\}$ since $\mathcal{IC}^u_\mathcal{P}(\emptyset, \emptyset) = \mathcal{IC}^l_\mathcal{P}(\emptyset, \emptyset) = \{p\}$, and $\mathcal{IC}^u_\mathcal{P}(\{p\}, \emptyset) = \mathcal{IC}^l_\mathcal{P}(\emptyset, \{p\}) = \{p\}$ (and thus a fixpoint is reached).

Likewise, $\mathcal{S}(\mathcal{O})(\emptyset, \{p\}) = (\{p\}, \{p\})$.We thus see that this program has one stable fixpoint, $(\{p\}, \{p\})$, which coincides with the well-founded fixpoint.

Pelov et al. [15] show that for normal logic programs, the fixpoints based on the four-valued immediate consequence operator $\mathcal{IC}_\mathcal{P}$ for a logic program give rise to the following correspondences: the total fixpoints correspond to supported models [24]; the Kripke-Kleene fixpoint corresponds to the Kripke-Kleene semantics [23]; the three-valued stable fixpoints coincides with the three-valued semantics as defined by [25], the well-founded model coincides with the homonymous semantics [25, 26], and the total stable fixpoints coincide with the total (or two-valued) stable models of a logic program (which are typically defined using a reduct [27]). Other semantics can be formulated by using different operators, and we'll see an example of this in Section 4. The semantics AFT gives rise to relative to an approximator (e.g. $IC_\mathcal{P}$), are summarised below:

	fixpoint ($(h,t) = \mathcal{O}(h,t)$)	total fixpoint ($(h,h) = \mathcal{O}(h,h)$)	least fixpoint ($\text{lfp}(\mathcal{O})$)
$\mathcal{O}$	Fixpoint	Total Fixpoint	Kripke-Kleene FP
$S(\mathcal{O})$	Stable Fixpoint	Total Stable Fixpoint	Well-founded FP

2.4. Reification

We take advantage of the *reification*-functionalities of clingo. This functionality allows to represent a logic program within the language of ASP using a format based on the ASPIF format [28]. In more detail, the grounder gringo can output a grounded logic program as a set of facts, which can then be used by a *meta-program* that defines semantics for this logic program. For example, the following logic program $\mathcal{P}_1 = \{\texttt{p:-r}, \texttt{not q.}, \texttt{r:-not q.}; \texttt{q:-not p.}\}$ is reified as in Listing 1.

Listing 1: Reification of $\mathcal{P}_1$

```
1 atom_tuple(0). atom_tuple(0,1). literal_tuple(0).       literal_tuple(0,-2).
      rule(disjunction(0),normal(0)).
2 atom_tuple(1). atom_tuple(1,3). rule(disjunction(1),normal(0)).
3 atom_tuple(2). atom_tuple(2,2). literal_tuple(1).       literal_tuple(1,3).
      literal_tuple(1,-1).        rule(disjunction(2),normal(1)).
4 literal_tuple(2). literal_tuple(2,3).   output(r,2).
5 literal_tuple(3). literal_tuple(3,1).   output(q,3).
6 literal_tuple(4). literal_tuple(4,2).   output(p,4).
```

The first rule `p:- r, not q` is encoded as `rule(disjunction(2),normal(1))` which intuitively encodes that it consists of a disjunction of atoms in the head, and a normal body (i.e. a conjunction of literals). The number 2 in `disjunction(2)` links the head of the rule to the facts `atom_tuple(2)` and `atom_tuple(2,2)`. The first of these facts specifies the identifier 2 for the tuple of atoms making up the rule head. The second encodes the inclusion of an atom with ID 2 in this tuple. The absence of any other occurrences of `atom_tuple(2,·)` makes this a one-tuple. Likewise, the 1 in `normal(1)` links the body of the rule with `literal_tuple(1)`, `literal_tuple(1,3)`, and `literal_tuple(1,-1)`. These encode, in turn, the presence of a tuple of literals with ID 1, the inclusion of an atom (i.e. a positive literal) with ID 3 in this tuple, and the tuple-inclusion of the default negation of an atom with ID 1 (i.e. a negated literal). Finally, the aforementioned atom IDs are matched to the original program atoms by the `output`/2-atoms. These link an atom in the first argument to the ID of a literal tuple in the second argument. For example, `output(q,3)` associates atom q with literal tuple 3; `literal_tuple(3)` and `literal_tuple(3,1)` then further tell us that the atom ID of q is 1. Note that the indirection through a literal tuple makes the identification of atoms less clear, but is required by the general aspif format. In the appendix, we give a formal description of the reification proces.

3. Semantics Based on the Fitting Operator

To implement (declaratively) the AFT-based fixpoint semantics of the operator $\mathcal{IC}_\mathcal{P}$, we proceed as follows: we represent the basic mathematical structure of a three-valued interpretation (h, t) in a format amenable to meta-programming in ASP (Section 3.1), then we implement the Fitting-operator $\mathcal{IC}_\mathcal{P}$ (Section 3.2), implement the search for (least) fixpoints (Section 3.3 and 3.4), and finally we implement the stable operator (Section 3.5).

3.1. Representing Three-valued interpretations

The first thing we have to represent are three-valued interpretations. We do this by first establishing that there are two bounds or models: `model(h;t)`. We use the terminology common in logic programming based on equilibrium logic, using `h`ere for the lower bound and `t`here for the upper bound. Furthermore, it will sometimes be useful to keep track of which step (e.g. of applying an operator) we are. Altogether, to denote an atom `a` is in the upper bound `t` at step `s`, we use the atom `hold(a,model(t),step(s))`. E.g. the interpretation $(x_0, y_0) = (\{p\}, \{p, q\})$ is represented as: `hold(p,model(h),step(0); p,model(t),step(0); q,model(t),step(0))`.

As we restrict attention to consistent interpretations, everything true in the lower bound `h` is also true in the upper bound `t`, encoded in the following rule:

Listing 2: Implementation of consistency of interpretations.

```
1 hold(L,model(t),step(S)):-  hold(L,model(h),step(S)).
```

3.2. Fitting Operator

Listing 3, inspired by Kaminski et al. [29], contains a program that takes as input an interpretation (h_0, t_0) encoded as described above, and outputes $\mathcal{IC}_\mathcal{P}(h_0, t_0)$.

Listing 3: Implementation of Fitting Operator $\mathcal{IC}_\mathcal{P}$.

```
1 conjunction(B, model(h), step(S)) :- literal_tuple(B), step(S),
       hold(L, model(h), step(S)) : literal_tuple(B, L), L > 0;
       not hold(L, model(t), step(S)): literal_tuple(B, -L), L > 0.
2 conjunction(B, model(t), step(S)) :- literal_tuple(B), step(S),
       hold(L, model(t), step(S)) : literal_tuple(B, L), L > 0;
       not hold(L, model(h), step(S)): literal_tuple(B, -L), L > 0.
3 hold(A,M,step(S+1)) : atom_tuple(H,A)   :- rule(disjunction(H),B),
      conjunction(B,M,step(S)), model(M).
```

First, we check whether a conjunction `B` occuring as a rule body holds by checking, for every positive literal occuring in `B` that conjunction, whether it holds in the lower bound `h`; and for every negative literal occuring `B` that it does not hold in the upper bound `t`. If that is the case, the conjunction `B` holds in the lower bound `h`. Likewise, the second rule checks whether a conjunction holds in the upper bound `t`. The third rule is then used to derive that the head of a rule whose body is true in the lower respectively upper bound is true in that bound as well. Notice that this rule is slightly more general than need be (as we restrict attention to normal logic programs): it derives that *all* atoms `A` in the head `H` are true in bound `M`. This is to retain generalisability.

3.3. Fixpoint Semantics

Having encoded three-valued interpretations and the Fitting operator $\mathcal{IC}_\mathcal{P}$, we can now encode fixpoint semantics. In more detail, Listing 4 contains a program that, when combined with an operator (e.g. Listing 3), looks for all (h, t) s.t. $\mathcal{IC}_\mathcal{P}(h, t) = (h, t)$. We do this by using the *guess-and-check*-methodolgy common in ASP-applications, where a choice rule is used to generate potential solutions (the *guess*-step), while constraints are used to filter out actual from potential solutions (the *check*-step). In this case, we guess an interpretation (h_0, t_0) and then check whether $\mathcal{IC}_\mathcal{P}(h_0, t_0) = (h_0, t_0)$. This is done in the program in Listing 4. We use choice-rules (line 1) to guess which atoms are true in the lower bound h_0

and upper bound t_0. In combination with the consistency-rule (Listing 2), this generates all possible three-valued interpretations. By combining this with the logic program for the operator (Listing 3), we can check what follows from our initial guess. Finally, the *check*-part (lines 2-6) ensures that (h_0, t_0) is a fixpoint. In more detail, we check whether when a step adds new atoms (i.e. $(h_i, t_i) \not\supseteq \mathcal{IC}^m_{\mathcal{P}}(h_i, t_i)$ where $m \in \{h, t\}$) on line 3 and when a step removes atoms (i.e. $(h_i, t_i) \not\subseteq \mathcal{IC}^m_{\mathcal{P}}(h_i, t_i)$ where $m \in \{h, t\}$) on line 4. Then we define a (partial) fixpoint as being neither a strict post-fixpoint nor a strict prefixpoint (line 4 and 5), and add as a constraint that both h_i and t_i should be a fixpoint.

Listing 4: Implementation of Fixpoint Semantics

```
1 {hold(A,model(M),step(0))}:- model(M), rule(disjunction(H),_), atom_tuple(H,A).
2 strict_atoms_added(model(M), step(S)) :- step(S), step(S+1),
      hold(V, model(M), step(S+1)), not hold(V, model(M), step(S)).
3 strict_atoms_removed(model(M), step(S)) :-step(S), step(S+1),
     not hold(V, model(M),step(S+1)), hold(V, model(M), step(S)).
4 fp(model(M), step(S)) :- model(M), step(S+1),
      not strict_atoms_added(model(M), step(S)),
      not strict_atoms_removed(model(M), step(S)), step(S).
5 fp(model(M)) :- fp(model(M), step(_)).
6 :- not fp(model(M)), model(M).  step(0;1).
```

3.4. Kripke-Kleene Semantics

We now look at a special case of the fixpoint semantics that can be found constructively: the Kripke-Kleene semantics, defined as the $\leq_i$-least fixpoint of the approximator.

The program is found in Listing 5. As an upper bound on the number of iterations of the $\mathcal{IC}_{\mathcal{P}}$-operator to obtain a fixpoint, we start by counting the number of literals in the logic program under consideration, and encode every step as a `step` (line 1). We then encode the idea of iterating $\mathcal{IC}_{\mathcal{P}}$ until a fixpoint is reached in , by first encoding $(h_0, t_0) = (\emptyset, \mathcal{A})$ in lines 2-4, i.e. assuming that every atom holds in the upper bound (`hold(L,model(t),step(0))` whereas no atom holds in the lower bound (which does not need to be specified due to the closed world assumption being used in ASP). Derivations using the Fitting-operator (i.e. performing $(h_{i+1}, t_{i+1}) = \mathcal{IC}_{\mathcal{P}}(h_i, t_i)$) are already defined in Listing 3, and lines 2-6 of Listing 4 ensure that a fixpoint is reached.

Listing 5: Implementation of Kripke-Kleene Semantics

```
1 step(0..N+2):- #count{L: literal_tuple(_, L)}=N.
2 hold(A, model(t), step(0)) :- atom_tuple(_, A).
3 hold(L, model(t), step(0)) :- literal_tuple(_, L), L >= 0.
4 hold(L, model(t), step(0)) :- literal_tuple(_, -L), L >= 0.
```

3.5. Stable Semantics and Well-Founded Semantics

Recall that the stable version of an operator is obtained by constructing the least fixpoint of $\mathcal{IC}^l_{\mathcal{P}}(\cdot, t)$ (respectively $\mathcal{IC}^u_{\mathcal{P}}(h, \cdot)$), i.e. by constructing $\bigcup(\mathcal{IC}^l)^\infty(\emptyset, t)$. Thus, within one single step of the operator $S(\mathcal{IC}^l_{\mathcal{P}})$, we now have to consider several steps of the operator $\mathcal{IC}^l_{\mathcal{P}}(\cdot, t)$ (until a fixpoint is reached). To do this, we extend the term `step/1` to the term `step/2`, which intuitively allows to express that something holds at step i of the operator $\mathcal{IC}^l_{\mathcal{P}}(\cdot, y_i)$ to reach the j^{th} application of the stable operator $S(\mathcal{IC}^l_{\mathcal{P}})$. In other words, `hold(L,model(h),step(i,j))` represents a literal is in $(\mathcal{IC}^l_{\mathcal{P}})^j(\emptyset, y_i)$. This results in the implementation in Listing 4. The rule on line 1 expresses that the body of a rule holds at stage `step(S1,S2)` if all positive literals hold in the lower bound of the current "internal" step `step(S1,S2)` (i.e. $(\mathcal{IC}^l_{\mathcal{P}})^{\mathtt{S2}}(\emptyset, y_{\mathtt{S1}})$), and none of the negative body literals hold in the previous "external" step `step(S1-1)` (i.e. $y_{\mathtt{S1}-1}$). Similarly for the upper bound. Derivation of true and undecided atoms (line 3-4) proceeds exactly as in Listing 3, whereas (least) fixpoint construction (lines 5-7) proceeds exactly as in Listing 5.

Listing 6: Implementation of the Stable Operator

```
1 conjunction(B,model(h),step(S1,S2)) :- step(S1,S2); literal_tuple(B);
      hold(L,model(h),step(S1,S2)): literal_tuple(B,L), L > 0; not
      hold(L,model(t),step((S1-1))): literal_tuple(B,-L), L > 0.
2 conjunction(B,model(t),step(S1,S2)) :- step(S1,S2); literal_tuple(B);
      hold(L,model(t),step(S1,S2)): literal_tuple(B,L), L > 0; not
      hold(L,model(h),step((S1-1))): literal_tuple(B,-L), L > 0.
3 hold(A,model(t),step(S1,(S2+1))) :- step(S1,(S2+1)); atom_tuple(Head,A);
      rule(disjunction(Head),normal(Body)); conjunction(Body,model(t),step(S1,S2)).
4 hold(A,model(h),step(S1,(S2+1))) :- step(S1,(S2+1)); atom_tuple(Head,A);
      rule(disjunction(Head),normal(Body)); conjunction(Body,model(h),step(S1,S2)).
5 fp(model(M), step(S1, S2)) :- model(M), not strict_atoms_added(model(M), step(S1, S2)),
      step(S1, S2), not strict_atoms_removed(model(M), step(S1, S2)), step(S1, S2).
6 strict_atoms_added(model(M), step(S1, S2)) :- step(S1, S2), step(S1, S2+1), hold(V,
      model(M), step(S1, S2+1)), not hold(V, model(M), step(S1, S2)).
7 strict_atoms_removed(model(M), step(S1, S2)) :- step(S1, S2), step(S1, S2+1), not
      hold(V, model(M), step(S1, S2+1)), hold(V, model(M), step(S1, S2)).
```

Semantics based on this operator, such as the partial stable fixpoints and the well-founded fixpoints, are then obtained just as the fixpoint semantics respectively Kripke-Kleene semantics were obtained on the basis of the Fitting-operator in Listing 4 respectively Listing 5. We do not reproduce these programs in this paper due to spatial restrictions, but they can be found in the supplementary material.

4. Ultimate Operators

The semantics we have discussed so far where all based on the Fitting operator $\mathcal{IC}_\mathcal{P}$. One benefit of AFT is that one can easily define a family of alternative semantics by simply defining another approximator. There are different ways to obtain an approximator, and these include constructive definitions. One such constructive definition is the *ultimate* operator [30], which is coincidentally the *most precise* approximator. In this section, we implement the ultimate operator in our library. We first survey the necessary background on the ultimate operator Section 4.1 and then explain our implementation Section 4.2.

4.1. Background on the Ultimate Operator

We first recall that the ultimate approximator for $IC_\mathcal{P}$ for normal logic programs $\mathcal{P}$ is defined by $\mathcal{IC}^\mathcal{U}_\mathcal{P}(h,t) = (\mathcal{IC}^{\mathcal{U},l}_\mathcal{P}(h,t),\ \mathcal{IC}^{\mathcal{U},u}_\mathcal{P}(h,t))$ with: $\mathcal{IC}^{\mathcal{U},l}_\mathcal{P}(h,t) = \bigcap_{h\subseteq z\subseteq t}\{\alpha \mid \alpha\texttt{:-}\phi \in \mathcal{P} \text{ and } z(\phi) = \mathsf{T}\}$, and $\mathcal{IC}^{\mathcal{U},u}_\mathcal{P}(h,t) = \bigcup_{h\subseteq z\subseteq t}\{\alpha \mid \alpha\texttt{:-}\phi \in \mathcal{P} \text{ and } z(\phi) = \mathsf{T}\}$. We will use the following characterisation of the ultimate operator. $B_\mathcal{P}(a)$ denotes a disjunction of the bodies of all rules in $\mathcal{P}$ that have a as a head. Given a formula ϕ in DNF, $h \subseteq t \subseteq \mathcal{A}$, $[\phi]_{h,t}$ is the formula obtained by replacing in ϕ every atom in h by $\top$, and every atom not in t by $\bot$.

Proposition 1 (Lemma 6.9 [30]). *Given two sets of atoms $h \subseteq t$, (1) $a \in \mathcal{IC}^{\mathcal{U},l}_\mathcal{P}(h,t)$ iff $[B_p(a)]_{h,t}$ is a tautology, and (2) $a \in \mathcal{IC}^{\mathcal{U},u}_\mathcal{P}(h,t)$ iff $[B_p(a)]_{h,t}$ is satisfiable.*

In other words, the lower bound of $\mathcal{IC}^\mathcal{U}_\mathcal{P}$ can be obtained by a call to an UNSAT-oracle, whereas the upper bound can be obtained by a call to a SAT-oracle. Notice that, for a formula in DNF the former is in NP whereas the later can be done in polytime. This means the complexity of inference based on the Kripke-Kleene and well-founded fixpoint of $\mathcal{IC}^\mathcal{U}_\mathcal{P}$ is in Δ^P_2 whereas determining existence of the (stable) fixpoints of $\mathcal{IC}^\mathcal{U}_\mathcal{P}$ is Σ^P_2-complete [30]. These complexity results mean that we will need to use disjunction in our implementation of the ultimate operator.

Example 2. *Consider the program $\mathcal{P} = \{p\texttt{:-}p.;\ p\texttt{:-not }p\}$. Then $IC_\mathcal{P}(\emptyset) = IC_\mathcal{P}(\{p\}) = \{p\}$. We see the following behaviour for the operators $\mathcal{IC}_\mathcal{P}$ and $\mathcal{IC}^\mathcal{U}_\mathcal{P}$:*

(h,t)	$\mathcal{IC}_\mathcal{P}(h,t)$	$\mathcal{IC}^\mathcal{U}_\mathcal{P}(h,t)$	(h,t)	$\mathcal{IC}_\mathcal{P}(h,t)$	$\mathcal{IC}^\mathcal{U}_\mathcal{P}(h,t)$	(h,t)	$\mathcal{IC}_\mathcal{P}(h,t)$	$\mathcal{IC}^\mathcal{U}_\mathcal{P}(h,t)$
$(\emptyset,\{p\})$	$(\emptyset,\{p\})$	$(\{p\},\{p\})$	$(\emptyset,\emptyset)$	$(\{p\},\{p\})$	$(\{p\},\{p\})$	$(\{p\},\{p\})$	$(\{p\},\{p\})$	$(\{p\},\{p\})$

We see that e.g. the Kripke-Kleene fixpoint of $\mathcal{IC}_\mathcal{P}$ is $(\emptyset, \{p\})$ and that of $\mathcal{IC}^\mathcal{U}_\mathcal{P}$ is $(\{p\}, \{p\})$.

4.2. Implementation

We have implemented the ultimate operator $\mathcal{IC}^\mathcal{U}_\mathcal{P}$ in Listing 7, that takes as an input a program and information in what holds in the input state (h, t), and calculates $\mathcal{IC}^\mathcal{U}_\mathcal{P}(h_i, t_i)$ (in the form of predicates `hold(X,M,i+1)`). Notice that in Listing 7, we have left out the term `step(S)` and replaced it by `I` to avoid some clutter, and similarly for `model(M)`.

Listing 7: Implementation of the Ultimate Operator

```
1  relevantFor(A,T):- rule(disjunction(X), normal(T)), atom_tuple(X,A).
2  isInconsistent(A,T):- relevantFor(A,T), posin(T,X),negin(T,X).
3  isInconsistent(A,T):- relevantFor(A,T), posin(T,X), not hold(X,t,I), step(I).
4  isInconsistent(A,T):- relevantFor(A,T), negin(T,X), hold(X,h,I).
5  isConsistent(A) :- relevantFor(A,T), not isInconsistent(A,T).
6  hold(A,t,I+1) :- step(I),  atom_tuple(_,A), isConsistent(A).
7  true(|X|,A),false(|X|,A):-  relevantFor(A,T), literal_tuple(T,X).
8  true(|X|,A):-relevantFor(A,T), literal_tuple(T,X), hold(|X|,h,0).
9  false(|X|,A):-relevantFor(A,T), literal_tuple(T,X), not hold(|X|,t,0).
10 posin(T,A):- literal_tuple(T,A), A>0.
11 negin(T,A):- literal_tuple(T,-A), A>0.
12 unsatNeg(T,X):- relevantFor(X,T), true(A,X):   posin(T,A);
       relevantFor(X,T), false(A,X): negin(T,A).
13 bot(A):- unsatNeg(_,A).
14 true(|X|,A):- bot(A), relevantFor(A,T), literal_tuple(T,X).
15 false(|X|,A):- bot(A), relevantFor(A,T), literal_tuple(T,X).
16 hold(A,h,I+1):- step(I),  bot(A).
```

We first construct $B_\mathcal{P}(a)$, i.e. the DNF formed by the bodies of all rules that have a as a head, by delineating all relevant clauses for an atom A (lines 1). Intuitively, we now obtain $B_\mathcal{P}(a) = \bigvee\{\bigwedge_{\texttt{literal_tuple}(l,t)} \texttt{output}(l) \mid \texttt{relevantFor}(a,t)\}$.

For the upper bound (according to the second item of Proposition 1) checking whether $a \in \mathcal{IC}^{\mathcal{U},u}_\mathcal{P}(h_i, t_i)$ is easy. As the formula $[B_p(a)]_{h_i,t_i}$ is in DNF, we simply check (lines 2-6) whether every clause is consistent, given the information about the previous lower and upper bound: In more detail, a clause is inconsistent if it (a) contains the same atom positively and negatively, or (b) contains an atom positively that was not contained in the upper bound (and thus certainly false), or (c) contains an atom negatively that was contained in the lower bound (and thus certainly true). An atom a has a consistent DNF $[B_p(a)]_{h,t}$ if it has at least one consistent clause (line 5). In that case, it is derived in the new upper bound (line 6).

We now move to the lower bound (line 7-13) $\mathcal{IC}^{\mathcal{U},l}_\mathcal{P}$. To determine whether $a \in \mathcal{IC}^{\mathcal{U},l}_\mathcal{P}(h, t)$, we have to check satisfiability of $[B_\mathcal{P}(a)]_{h,t}$. We do this by implementing a sat-solver using the so-called *saturation technique*. Due to spatial limitations, we cannot explain this technique in full detail, but we refer to the work of [31] for a more complete description. In a nutshell, the solver will look, for every atom a, for an interpretation z_a (encoded by `true(X,a)` and `false(X,a)` that makes true `not` $[B_\mathcal{P}(a)]_{h,t}$. If such an interpretation exists, it means $[B_\mathcal{P}(a)]_{h,t}$ is not a tautology. z_a is obtained (on line 7) by guessing, using disjunctions instead of choice rules (this is necessary for the saturation-technique to work). Literals in h have to be in z_a (line 8), whereas literals not in t cannot be in z_a (line 9). The rule on line 12 defines the negation of a literal tuple T relevant for a as being unsatisfiable with the help of 2 helper predicates (line 10-11).

As soon as the negation of an atom is unsatisfiable, we derive bot, which leads to explosion (lines 13-15) this is necessary for the saturation-technique to work). Due to the way disjunction works in ASP, this will lead to the solver looking for answer sets where `bot` is only derived if it is absolutely necessary, i.e. this is the only way to satisfy all rules. For our situation, this means that `bot` will be derivable only if there is no interpretation that falsifies $[B_p(a)]_{h,t}$.

Having implemented our SAT-solver for every $[B_p(a)]_{h,t}$, which will derive `bot` only if $[B_p(a)]_{h,t}$ is a tautology, we derive an atom being in $h_{i+1} = \mathcal{IC}^{\mathcal{U},l}_\mathcal{P}(h_i, t_i)$ on line 16.

5. Experimental Exploration

In order to both validate our encodings and quantitatively study the AFT-semantics, we ran some preliminary computational experiments. Indeed, a benefit of our encodings is that we can now investigate some relationships between semantics about which we had *qualitative* knowledge via theoretical results in a more quantitative way. For example, we know that the Kripke-Kleene model is always less precise than the well-founded model (i.e. $\mathrm{lfp}_{\leq_i}(\mathcal{IC}_\mathcal{P}) \leq_i \mathrm{lfp}_{\leq_i}(S(\mathcal{IC}_\mathcal{P}))$), but this relationship can now be studied more quantitatively. Likewise, it is well-known that the set of stable fixpoints is a subset of the set of fixpoints, and that the set of total (stable) fixpoints is a subset of the set of partial (stable) fixpoints.

As a preliminary study, we looked at three NLP-encodings of graph-based problems applied to synthetically generated directed graphs. The first of these problems interprets edges as attacks, and its encoding `accept(Y):- edge(X,Y), not accept(X).` accepts nodes when their attackers are not accepted. For this encoding, all models are necessarily stable. The second problem encoding adds self-support with `accept(X):- accept(X).`, permitting (partial/total) fixpoints that are not stable. The third is a graph colouring problem, serving mainly as an encoding that induces larger reified programs.

We focused on the following three questions.

`q1` : What is the computational overhead caused by reification and the meta-encodings?

`q2` : what is the relation between the number of partial/total (stable) fixpoints?

`q3` : what is the relationship between the size of the Kripke-Kleene and well-founded fixpoint?

With regard to `q1`, we measured the time needed to compute all models for every reification of a graph–problem pair and each of our meta programs. Figure 2 shows these measurements, graphed against the size of the reified logic program that produced them. Owing to its simplifications and optimizations, we see that the default Clingo solver finishes orders of magnitude sooner than our meta programs across all experiments. Specifically, compare Clingo's default performance to that using the total stable fixpoint meta program, which computes the same answer sets and averagely among meta programs. Then too, look at the well-founded meta program—which requires nested least fixpoint computation, and is therefore considerably slower than the other meta programs. Finally, note the partial fixpoint meta program—which requires no constructions of least fixpoints and finishes notably

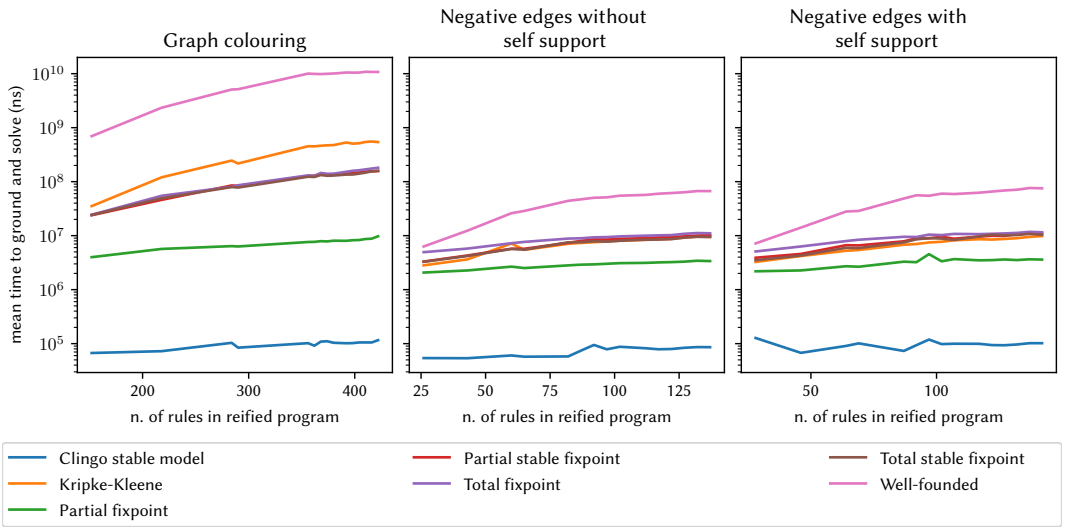


Figure 2: The time in nanoseconds needed to ground and solve a problem graphed against the size of its reification, grouped by graph problem. (note: vertical log-scale)

sooner than the other meta programs. As we did not focus on efficiency in our encodings, this is not surprising.

For `q2` and `q3`, the supplementary material to this work includes figures where the number of fixpoints per semantics and the precisions of the Kripke-Kleene and well-founded are graphed per problem instance. As predicted, we find a greater number of (non-stable) partial/total fixpoints than stable ones for the attack encoding with self-support, and find only stable fixpoinst for the encodings for graph colouring and attacks without self-support. Finally we find that, for these problem encodings, the Kripke-Kleene and well-founded fixpoints coincide. In other words, we see that the well-founded fixpoint is at least as precise the Kripke-Kleene fixpoint in these cases, but not moreso.

6. Conclusion, in View of Related Work

The main contribution of our work is the implementation of AFT-based semantics for normal logic programs using meta-programming. One of the benefits of this encoding is that it allows to perform experiments on AFT-based semantics. We also developed a tool that transforms and combines the different modules depending on which semantics one wants to calculate, unifies calls to Clingo and transforms the output to a more readable object-level format. The tool also allows users to specify their own operator, on the basis of which encodings for all semantics can be automatically generated. More details can be found in the online repository[1]. Furthermore, the appendix also contains formal proofs of the correctness of our encodings.

Our work takes inspiration from the work of [29], where total fixpoints were defined using meta-programming. In terms of related work, we mention the IDP-system [32, 33]: a finite model generator that supports first-order logic supplemented with, among others, inductive definitions. As the semantics of inductive definitions are akin to the well-founded fixpoint, this can be seen as a solver making use of ideas that also inspired AFT. However, the IDP-system does not cover other semantics (e.g. those based on the ultimate operator, or (stable) fixpoints). On the other hand, the IDP-system does allow for a richer language than ours, is geared towards efficiency, and covers other reasoning tasks such as model expansion. Some semantics covered by our library were also implemented independently. For example, the DLV-system also implements the well-founded semantics [34]. Likewise, [35] provide an implementation of the well-founded semantics for logic programs with aggregates. In future work, we will compare the performance of our library with that of these systems, even though we conjecture these systems, since they have been implemented with an eye on efficiency, will probably fare better than our library.

Bogaerts and Cruz-Filipe [36] have formalised the main concepts from approximation theory in the automated theorem prover coq. Their goal is quite different: ours is to compute "object level" semantics, while the implementation in coq is mainly used to verify meta-theoretical results.

Avenues for future work include the implementation of other AFT-based formalisms [12, 37, 13], including LPs with aggregates [38, 15, 11]. Furthermore, making the encodigns more computationally efficient is another interesting avenue for future work.

Acknowledgments

This work was partially supported by the project *LogicLM: Combining Logic Programs with Language Models* with file number NGF.1609.241.010 of the research programme NGF AiNed XS Europa 2024-1 and *SaTSNASP: Safe and Transparent Scheduling with Neuro-Symbolic Answer Set Programming* with file number NGF.1609.243.040 and https://doi.org/10.61686/CZLDA50536 of the research programme NGF AiNed XS Europa 2024-3, which are (partly) financed by the Dutch Research Council (NWO).

[1] https://github.com/jesseheyninck/metaft

Declaration on Generative AI

During the preparation of this work, the author(s) used ChatGPT in order to: Grammar and spelling check. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] M. Denecker, V. Marek, M. Truszczyński, Approximations, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning, in: Logic-based Artificial Intelligence, volume 597 of *The Springer International Series in Engineering and Computer Science*, Springer, 2000, pp. 127–144.

[2] H. Strass, Approximating operators and semantics for abstract dialectical frameworks, Artificial Intelligence 205 (2013) 39–70.

[3] M. Denecker, V. Marek, M. Truszczyński, Uniform semantic treatment of default and autoepistemic logics, Artificial Intelligence 143 (2003) 79–122.

[4] F. Liu, J.-H. You, Alternating fixpoint operator for hybrid mknf knowledge bases as an approximator of aft, Theory and Practice of Logic Programming 22 (2022) 305–334.

[5] B. Bogaerts, M. Jakubowski, Fixpoint semantics for recursive SHACL, in: A. Formisano, Y. A. Liu, B. Bogaerts, A. Brik, V. Dahl, C. Dodaro, P. Fodor, G. L. Pozzato, J. Vennekens, N. Zhou (Eds.), Proceedings 37th International Conference on Logic Programming (Technical Communications), ICLP Technical Communications 2021, Porto (virtual event), 20-27th September 2021, volume 345 of *EPTCS*, 2021, pp. 41–47. URL: https://doi.org/10.4204/EPTCS.345.14. doi:10.4204/EPTCS.345.14.

[6] B. Bogaerts, Weighted abstract dialectical frameworks through the lens of approximation fixpoint theory, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 33, 2019, pp. 2686–2693.

[7] J. Heyninck, O. Arieli, Semantics for non-flat assumption-based argumentation, revisited, in: Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, Jeju, South Korea, August 3-9, 2024, ijcai.org, 2024, pp. 3413–3420. URL: https://www.ijcai.org/proceedings/2024/378.

[8] C. Antić, T. Eiter, M. Fink, Hex semantics via approximation fixpoint theory, in: Proceedings of LPNMR'13, LNCS 8148, Springer, 2013, pp. 102–115.

[9] A. Charalambidis, P. Rondogiannis, I. Symeonidou, Approximation fixpoint theory and the well-founded semantics of higher-order logic programs, Theory and Practice of Logic Programming 18 (2018) 421–437.

[10] M. Denecker, M. Bruynooghe, J. Vennekens, Approximation fixpoint theory and the semantics of logic and answers set programs, in: Correct reasoning, Springer, 2012, pp. 178–194.

[11] J. Heyninck, B. Bogaerts, Non-deterministic approximation operators: Ultimate operators, semi-equilibrium semantics, and aggregates, Theory Pract. Log. Program. 23 (2023) 632–647. URL: https://doi.org/10.1017/s1471068423000236. doi:10.1017/S1471068423000236.

[12] J. Heyninck, O. Arieli, B. Bogaerts, Non-deterministic approximation fixpoint theory and its application in disjunctive logic programming, Artif. Intell. 331 (2024) 104110. URL: https://doi.org/10.1016/j.artint.2024.104110. doi:10.1016/J.ARTINT.2024.104110.

[13] J. Heyninck, Operator-based semantics for choice programs: Is choosing losing?, in: P. Marquis, M. Ortiz, M. Pagnucco (Eds.), Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning, KR 2024, Hanoi, Vietnam. November 2-8, 2024, 2024. URL: https://doi.org/10.24963/kr.2024/42. doi:10.24963/KR.2024/42.

[14] P. Kettmann, J. Heyninck, H. Strass, Approximation fixpoint theory as a unifying framework for fuzzy logic programming semantics, in: Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2025, Montreal, Canada, August 16-22, 2025, ijcai.org, 2025, pp. 4544–4552. URL: https://doi.org/10.24963/ijcai.2025/506. doi:10.24963/IJCAI.2025/506.

[15] N. Pelov, M. Denecker, M. Bruynooghe, Well-founded and stable semantics of logic programs with aggregates, Theory and Practice of Logic Programming 7 (2007) 301–353.
[16] J. P. Spaans, J. Heyninck, A unifying framework for semiring-based constraint logic programming with negation, in: Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2025, Montreal, Canada, August 16-22, 2025, ijcai.org, 2025, pp. 2684–2692. URL: https://doi.org/10.24963/ijcai.2025/299. doi:10.24963/IJCAI.2025/299.
[17] R. Kaminski, T. Schaub, On the foundations of grounding in answer set programming, Theory and Practice of Logic Programming 23 (2023) 1138–1197.
[18] P. Hou, M. Denecker, A deductive system for fo (id) based on least fixpoint logic, in: International Conference on Logic Programming and Nonmonotonic Reasoning, Springer, 2009, pp. 129–141.
[19] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot asp solving with clingo, Theory and Practice of Logic Programming 19 (2019) 27–82.
[20] V. Lifschitz, Answer set programming, Springer Heidelberg, 2019.
[21] 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 and Practice of Logic Programming 20 (2020) 294–309.
[22] M. H. van Emden, R. A. Kowalski, The semantics of predicate logic as a programming language, J. ACM 23 (1976) 733–742.
[23] M. Fitting, A kripke-kleene semantics for logic programs, The Journal of Logic Programming 2 (1985) 295–312.
[24] S. Brass, J. Dix, Characterizations of the stable semantics by partial evaluation, in: International Conference on Logic Programming and Nonmonotonic Reasoning, Springer, 1995, pp. 85–98.
[25] T. C. Przymusinski, The well-founded semantics coincides with the three-valued stable semantics, Fundamenta Informaticae 13 (1990) 445–463.
[26] A. Van Gelder, K. A. Ross, J. S. Schlipf, The well-founded semantics for general logic programs, Journal of the ACM 38 (1991) 619–649.
[27] M. Gelfond, V. Lifschitz, Classical negation in logic programs and disjunctive databases, New generation computing 9 (1991) 365–385.
[28] R. Kaminski, T. Schaub, P. Wanko, A tutorial on hybrid answer set solving with clingo, in: G. Ianni, D. Lembo, L. E. Bertossi, W. Faber, B. Glimm, G. Gottlob, S. Staab (Eds.), Reasoning Web. Semantic Interoperability on the Web - 13th International Summer School 2017, London, UK, July 7-11, 2017, Tutorial Lectures, volume 10370 of *Lecture Notes in Computer Science*, Springer, 2017, pp. 167–203. URL: https://doi.org/10.1007/978-3-319-61033-7_6. doi:10.1007/978-3-319-61033-7_6.
[29] R. Kaminski, J. Romero, T. Schaub, P. Wanko, How to build your own asp-based system?!, Theory and Practice of Logic Programming 23 (2023) 299–361.
[30] M. Denecker, V. W. Marek, M. Truszczynski, Ultimate approximations in nonmonotonic knowledge representation systems, in: Proceedings of the Eights International Conference on Principles of Knowledge Representation and Reasoning, 2002, pp. 177–190.
[31] T. Eiter, A. Polleres, Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications, Theory Pract. Log. Program. 6 (2006) 23–60. URL: https://doi.org/10.1017/S1471068405002577. doi:10.1017/S1471068405002577.
[32] B. D. Cat, B. Bogaerts, M. Bruynooghe, G. Janssens, M. Denecker, Predicate logic as a modeling language: the IDP system, in: M. Kifer, Y. A. Liu (Eds.), Declarative Logic Programming: Theory, Systems, and Applications, volume 20 of *ACM Books*, ACM / Morgan & Claypool, 2018, pp. 279–323. URL: https://doi.org/10.1145/3191315.3191321. doi:10.1145/3191315.3191321.
[33] M. Bruynooghe, H. Blockeel, B. Bogaerts, B. De Cat, S. De Pooter, J. Jansen, A. Labarre, J. Ramon, M. Denecker, S. Verwer, Predicate logic as a modeling language: modeling and solving some machine learning and data mining problems with idp3, Theory and Practice of Logic Programming 15 (2015) 783–817.
[34] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello, The DLV system for knowledge representation and reasoning, ACM Trans. Comput. Log. 7 (2006) 499–562. URL: https://doi.org/10.1145/1149114.1149117. doi:10.1145/1149114.1149117.

[35] M. Alviano, F. Calimeri, W. Faber, N. Leone, S. Perri, Unfounded sets and well-founded semantics of answer set programs with aggregates, Journal of Artificial Intelligence Research 42 (2011) 487–527.

[36] B. Bogaerts, L. Cruz-Filipe, Approximation fixpoint theory in coq - with an application to logic programming, in: V. Capretta, R. Krebbers, F. Wiedijk (Eds.), Logics and Type Systems in Theory and Practice - Essays Dedicated to Herman Geuvers on The Occasion of His 60th Birthday, volume 14560 of *Lecture Notes in Computer Science*, Springer, 2024, pp. 84–99. URL: https://doi.org/10.1007/978-3-031-61716-4_5. doi:10.1007/978-3-031-61716-4_5.

[37] S. Killen, J. You, J. Heyninck, An alternative theory of stable revision for nondeterministic approximation fixpoint theory and the relationships, in: T. Walsh, J. Shah, Z. Kolter (Eds.), AAAI-25, Sponsored by the Association for the Advancement of Artificial Intelligence, February 25 - March 4, 2025, Philadelphia, PA, USA, AAAI Press, 2025, pp. 15033–15040. URL: https://doi.org/10.1609/aaai.v39i14.33648. doi:10.1609/AAAI.V39I14.33648.

[38] N. Pelov, M. Truszczynski, Semantics of disjunctive programs with monotone aggregates: an operator-based approach, in: Proceedings of NMR'04, 2004, pp. 327–334.