TPLP: Page 1–14. © The Author(s), 2024. Published by Cambridge University Press 2024
doi:10.1017/xxxxx

The Stable Model Semantics for Higher-Order Logic Programming[*]

BOGAERTS BART[†‡], CHARALAMBIDIS ANGELOS[⊛], CHATZIAGAPIS GIANNOS[⁎], KOSTOPOULOS BABIS[⊛], POLLACI SAMUELE[†‡] and RONDOGIANNIS PANOS[⁎]
† *Vrije Universiteit Brussel, Belgium*
‡ *Katholieke Universiteit Leuven, Belgium*
⊛ *Harokopio University of Athens, Greece*
⁎ *National and Kapodistrian University of Athens, Greece*

submitted xx xx xxxx; revised xx xx xxxx; accepted xx xx xxxx

Abstract

We propose a stable model semantics for higher-order logic programs. Our semantics is developed using Approximation Fixpoint Theory (AFT), a powerful formalism that has successfully been used to give meaning to diverse non-monotonic formalisms. The proposed semantics generalizes the classical two-valued stable model semantics of (Gelfond and Lifschitz 1988) as-well-as the three-valued one of (Przymusinski 1990), retaining their desirable properties. Due to the use of AFT, we also get for free alternative semantics for higher-order logic programs, namely *supported model*, *Kripke-Kleene*, and *well-founded*. Additionally, we define a broad class of *stratified higher-order logic programs* and demonstrate that they have a unique two-valued higher-order stable model which coincides with the well-founded semantics of such programs. We provide a number of examples in different application domains, which demonstrate that higher-order logic programming under the stable model semantics is a powerful and versatile formalism, which can potentially form the basis of novel ASP systems.

KEYWORDS: Higher-Order Logic Programming, Stable Model Semantics, Approximation Fixpoint Theory.

1 Introduction

Recent research (Charalambidis et al. 2013; 2018a;b) has demonstrated that it is possible to design higher-order logic programming languages that have powerful expressive capabilities and simple and elegant semantic properties. These languages are genuine extensions of classical (first-order) logic programming: for example, Charalambidis et al. (2013) showed that positive higher-order logic programs have a Herbrand model intersection property and this least Herbrand model can also be produced as the least fixpoint of a continuous immediate consequence operator. In other words, crucial semantic results of classical (positive) logic programs transfer directly to the higher-order setting.

* This work was partially supported by Fonds Wetenschappelijk Onderzoek – Vlaanderen (project G0B2221N) and by a research project which is implemented in the framework of H.F.R.I call "Basic research Financing (Horizontal support of all Sciences)" under the National Recovery and Resilience Plan "Greece 2.0" funded by the European Union - NextGenerationEU (H.F.R.I. Project Number: 16116).

The above positive results, created the hope and expectation that all major achievements of first-order logic programming could transfer to the higher-order world. Despite this hope, it was not clear until now whether it is possible to define a *stable model semantics* for higher-order logic programs that would generalize the seminal work of Gelfond and Lifschitz (1988). For many extensions of standard logic programming, it is possible to generalize the *reduct* construction of Gelfond and Lifschitz to obtain a stable model semantics, as illustrated for instance by Faber et al. (2011) for an extension of logic programs with aggregates. For higher-order programs, however, it is not clear whether a reduct-based definition makes sense. The most important reason why it is challenging to define a higher-order reduct, is that using the powerful abstraction mechanisms that higher-order languages provide, one can *define* negation inside the language, for instance by the rule `neg X` $\leftarrow$ $\sim$`X` and use `neg` everywhere in the program where otherwise negation would be used, rendering syntactic definitions based on occurrences of negation difficult to apply.

Apart from scientific curiosity, the definition of a stable model semantics for higher-order logic programs also serves solid practical goals: there has been a quest for extending the power of ASP systems (Bogaerts et al. 2016; Amendola et al. 2019; Fandinno et al. 2021), and higher-order logic programming under the stable model semantics may prove to be a promising solution.

In this paper we define a stable model semantics for higher-order logic programs. Our semantics is developed using Approximation Fixpoint Theory (AFT) (Denecker et al. 2004). AFT is a powerful lattice-theoretic formalism that was originally developed to unify semantics of logic programming, autoepistemic logic (AEL) and default logic (DL) and was used to resolve a long-standing open question about the relation between AEL and DL semantics (Denecker et al. 2003). Afterwards, it has been applied to several other fields, including abstract argumentation (Strass 2013), active integrity constraints (Bogaerts and Cruz-Filipe 2018), stream reasoning (Antic 2020), and constraint languages for the semantic web (Bogaerts and Jakubowski 2021). In these domains, AFT has been used to define new semantics without having to reinvent the wheel (for instance, if one uses AFT to define a stable semantics, well-known properties such as minimality results will be automatic), to study the relation to other formalisms, and even to discover bugs in the original semantics (Bogaerts 2019). To apply AFT to a new domain, what we need to do is define a suitable semantic operator on a suitable set of "partial interpretations". Once this operator is identified, a family of well-known semantics and properties immediately rolls out of the abstract theory. In this paper, we construct such an operator for higher-order logic programs. Since our operator coincides with Fitting's (2002) three-valued immediate consequence operator for the case of standard logic programs, we immediately know that our resulting stable semantics generalizes the classical two-valued stable model semantics of Gelfond and Lifschitz (1988) as well as the three-valued one of Przymusinski (1990).

The main idea of our construction is to interpret the higher-order predicates of our language as three-valued relations over two-valued objects, i.e., as functions that take classical relations as arguments and return *true*, *false*, or *undef*. We demonstrate that such relations are equivalent to appropriate pairs of (classical) two-valued relations. The pair-representation gives us the basis to apply AFT, and to obtain, in a simple and transparent manner, the stable model semantics. At the same time, thanks to the versatility of AFT, without any additional effort, we obtain several alternative semantics for higher-order logic programs, namely *supported model*, *Kripke-Kleene*, and *well-founded* semantics. In particular, we argue that our well-founded semantics remedies certain deficiencies that have been observed in other attempts to define such a semantics for higher-order formalisms (Dasseville et al. 2015; 2016; Charalambidis et al. 2018a). We study properties of our novel semantics and to do so, we define a broad class of *stratified higher-order logic programs*. This is a non-trivial task mainly due to the fact that in the higher-order setting non-monotonicity can be well-hidden (Rondogiannis and Symeonidou 2017) and stratification will hence have to take more than just occurrences of negation into account. We demonstrate that stratified programs, as expected, indeed have a unique two-valued higher-order stable model, which coincides with the well-founded model of such programs. We feel that these results create a solid and broad foundation for the semantics of higher-order logic programs with negation. Finally, from a practical perspective, we showcase our semantics on three different examples. In Section 2, we start with max-clique, a simple graph-theoretic problem which we use to familiarize the reader with our notation and to demonstrate the power of abstraction. In Section 8, we study more intricate applications, namely semantics for abstract argumentation and Generalized Geography, which is a PSPACE-complete problem. These examples illustrate that higher-order logic programming under the stable model semantics is a powerful and versatile formalism, which can potentially form the basis of novel ASP systems.

2 A Motivating Example

In this section, we illustrate our higher-order logic programming language on the max-clique problem. A complete solution is included in Listing 1. We will assume an undirected graph is given by means of a unary predicate `v` (containing all the nodes of the graph) and a binary predicate `e` representing the edge-relation (which we assume to be symmetric). Lines 2 and 3 contain the standard trick that exploits an even loop of negation for simulating a choice, which in modern ASP input formats (Calimeri et al. 2020) would be abbreviated by a choice rule construct `{pick X : v X}`. Line 6 defines what it means to be a clique. In this line the (red) variable `P` is a first-order variable; it ranges over all *sets* of domain elements, whereas (blue) zero-order variables such as `X` in Line 2 range over actual elements of the domain. A set of elements is a clique if it *(i)* is a subset of `v`, and *(ii)* contains no two nodes without an edge between them. Failures to satisfy the second condition are captured by the predicate `hasNonEdge`. The predicate `clique` is a second-order predicate. Formally, we will say its type is $(\iota \to o) \to o$: it takes as input a relation of type $\iota \to o$, i.e., a set of base domain elements of type ι and it returns a Boolean (type o). In other words, the interpretation of `clique` will be a set of sets. Next, line 8 defines the second-order predicate `maxclique`, which is true precisely for those sets `P` that are subset-maximal among the set of all cliques, and line 10 asserts, using the standard trick with an odd loop over negation that `pick` must indeed be in `maxclique`.

This definition of `maxclique` makes use of a third-order predicate `maximal` which works with an arbitrary binary relation for comparing sets (here: the subset relation), as well as an arbitrary unary predicate over sets (here: the `clique` predicate). Listing 2 provides definitions of `maximal`, `equal`, and other generic predicates. Note that equality between predicates is not a primitive of the language: we define it in Line 4 of Listing 2. On the other hand, equality between atomic objects ($\approx$), which we use in Line 5 of Listing 1, is a primitive of the language. These generic definitions, which can be reused in different applications, illustrate the power of higher-order modelling: it enables reuse and provides great flexibility, e.g., if we are interested in cardinality-maximal cliques, we only need to replace subset by an appropriate relation comparing the size of two predicates. Also note that our solution has only a single definition of what it means to be a clique. This definition is used both to state that `pick` is a clique (the first atom of the rule defining `maximal` guarantees this) and to check that there are no larger cliques (in the rule defining `nonmaximal`).

Listing 1: Max-clique problem using stable semantics for higher-order logic programs.

```
% Pick a set of vertices (emulate choice rule)
pick X  ← v X, ∼(npick X).
npick X ← v X, ∼(pick X).
% Define what it means for a set of vertices to be a clique
hasNonEdge P ← P X, P Y, ∼(X ≈ Y), ∼(e X Y).
clique P ← subset P v,  ∼(hasNonEdge P).
% Define what it means to be a max−clique:
maxclique P ← maximal subset clique P.
% The selected set should be a max−clique
f ← ∼f, ∼(maxclique pick).
```

Listing 2: Definitions of generic higher-order predicates.

```
% Define generic higher−order predicates: subset, equal, maximal
nonsubset P Q ← P X, ∼(Q X).
subset P Q ←  ∼(nonsubset P Q).
equal P Q ← subset P Q, subset Q P.
% maximal Ord Prop  P means: P is Ord−maximal among sets satisfying Prop
maximal Ord Prop P    ←  Prop P, ∼(nonmaximal Ord Prop P).
nonmaximal Ord Prop P ←  Prop Q, Ord P Q, ∼(equal P Q).
```

3 $\mathcal{HOL}$: A Higher-Order Logic Programming Language

In this section we define the syntax of the language $\mathcal{HOL}$ that we use throughout the paper. For simplicity reasons, the syntax of $\mathcal{HOL}$ does not include function symbols; this is a restriction that can easily be lifted. $\mathcal{HOL}$ is based on a simple type system with two base types: o, the Boolean domain, and ι, the domain of data objects. The composite types are partitioned into *predicate* ones (assigned to predicate symbols) and *argument* ones (assigned to parameters of predicates).

Definition 3.1
Types are either *predicate* or *argument*, denoted by π and ρ respectively, and defined as:

$$\pi := o \mid (\rho \to \pi)$$
$$\rho := \iota \mid \pi$$

As usual, the binary operator $\to$ is right-associative. It can be easily seen that every predicate type π can be written in the form $\rho_1 \to \cdots \to \rho_n \to o$, $n \geq 0$ (for $n = 0$ we assume that $\pi = o$). We proceed by defining the syntax of $\mathcal{HOL}$.

Definition 3.2
The *alphabet* of $\mathcal{HOL}$ consists of the following: *predicate variables* of every predicate type π (denoted by capital letters such as $\mathsf{P}, \mathsf{Q}, \ldots$); *predicate constants* of every predicate

type π (denoted by lowercase letters such as $\mathsf{p}, \mathsf{q}, \ldots$); *individual variables* of type ι (denoted by capital letters such as $\mathsf{X}, \mathsf{Y}, \ldots$); *individual constants* of type ι (denoted by lowercase letters such as $\mathsf{a}, \mathsf{b}, \ldots$); the *equality* constant $\approx$ of type $\iota \to \iota \to o$ for comparing individuals of type ι; the *conjunction* constant $\wedge$ of type $o \to o \to o$; the *rule operator* constant $\leftarrow$ of type $o \to o \to o$; and the *negation* constant $\sim$ of type $o \to o$.

Arbitrary variables (either predicate or individual ones) will usually be denoted by R.

Definition 3.3
The *terms* and *expressions* of $\mathcal{HOL}$ are defined as follows. Every predicate variable/constant and every individual variable/constant is a term; if E_1 is a term of type $\rho \to \pi$ and E_2 a term of type ρ then $(\mathsf{E}_1\ \mathsf{E}_2)$ is a term of type π. Every term is also an expression; if E is a term of type o then $(\sim\mathsf{E})$ is an expression of type o; if E_1 and E_2 are terms of type ι, then $(\mathsf{E}_1 \approx \mathsf{E}_2)$ is an expression of type o.

We will omit parentheses when no confusion arises. To denote that an expression E has type ρ we will often write $\mathsf{E} : \rho$.

Definition 3.4
A *rule* of $\mathcal{HOL}$ is a formula $\mathsf{p}\ \mathsf{R}_1 \cdots \mathsf{R}_n \leftarrow \mathsf{E}_1 \wedge \ldots \wedge \mathsf{E}_m$, where p is a predicate constant of type $\rho_1 \to \cdots \to \rho_n \to o$, $\mathsf{R}_1, \ldots, \mathsf{R}_n$ are distinct variables of types $\rho_1, \ldots, \rho_n$ respectively and the E_i are expressions of type o. The term $\mathsf{p}\ \mathsf{R}_1 \cdots \mathsf{R}_n$ is the *head* of the rule and $\mathsf{E}_1 \wedge \ldots \wedge \mathsf{E}_m$ is the *body* of the rule. A *program* P of $\mathcal{HOL}$ is a finite set of rules.

We will often follow the common logic programming notation and write $\mathsf{E}_1, \ldots, \mathsf{E}_m$ instead of $\mathsf{E}_1 \wedge \cdots \wedge \mathsf{E}_m$ for the body of a rule. For brevity reasons, we will often denote a rule as $\mathsf{p}\ \overline{\mathsf{R}} \leftarrow \mathsf{B}$, where $\overline{\mathsf{R}}$ is a shorthand for a sequence of variables $\mathsf{R}_1 \cdots \mathsf{R}_n$ and B represents a conjunction of expressions of type o.

4 The Two-Valued Semantics of $\mathcal{HOL}$

In this section we define an immediate consequence operator for $\mathcal{HOL}$ programs, which is an extension of the classical T_{P} operator for first-order logic programs. We start with the semantics of the types of our language. In the following, we denote by U_{P} the *Herbrand universe* of P, namely the set of all constants of the program.

The semantics of the base type o is the classical Boolean domain $\{\mathit{true}, \mathit{false}\}$ and that of the base type ι is U_{P}. The semantics of types of the form $\rho \to \pi$ is the set of all functions from the domain of type ρ to that of type π. We define, simultaneously with the meaning of every type, a partial order on the elements of the type.

Definition 4.1
Let P be an $\mathcal{HOL}$ program. We define the (two-valued) meaning of a type with respect to U_{P}, as follows:

- $[\![o]\!]_{U_{\mathsf{P}}} = \{\mathit{true}, \mathit{false}\}$. The partial order $\leq_o$ is the usual one induced by the ordering $\mathit{false} <_o \mathit{true}$
- $[\![\iota]\!]_{U_{\mathsf{P}}} = U_{\mathsf{P}}$. The partial order $\leq_\iota$ is the trivial one defined as $d \leq_\iota d$ for all $d \in U_{\mathsf{P}}$

- $[\![\rho \to \pi]\!]_{U_{\mathsf{P}}} = [\![\rho]\!]_{U_{\mathsf{P}}} \to [\![\pi]\!]_{U_{\mathsf{P}}}$, namely the set of all functions from $[\![\rho]\!]_{U_{\mathsf{P}}}$ to $[\![\pi]\!]_{U_{\mathsf{P}}}$. The partial order $\leq_{\rho\to\pi}$ is defined as: for all $f, g \in [\![\rho \to \pi]\!]_{U_{\mathsf{P}}}$, $f \leq_{\rho\to\pi} g$ iff $f(d) \leq_\pi g(d)$ for all $d \in [\![\rho]\!]_{U_{\mathsf{P}}}$.

The subscripts from the above partial orders will be omitted when they are obvious from context. Moreover, we will omit the subscript U_{P} assuming that our semantics is defined with respect to a specific program P.

As we mentioned before, each predicate type π can be written in the form $\rho_1 \to \cdots \to \rho_n \to o$. Elements of $[\![\pi]\!]$ can be thought of, alternatively, as subsets of $[\![\rho_1]\!] \times \cdots \times [\![\rho_n]\!]$ (the set contains precisely those n-tuples mapped to *true*). Under this identification, it can be seen that $\leq_\pi$ simply becomes the subset relation.

Proposition 4.1
For every predicate type π, $([\![\pi]\!], \leq_\pi)$ is a complete lattice.

In the following, we denote by $\bigvee_{\leq_\pi}$ and $\bigwedge_{\leq_\pi}$ the corresponding lub and glb operations of the above lattice. When viewing elements of π as *sets*, $\bigvee_{\leq_\pi}$ is just the union operator and $\bigwedge_{\leq_\pi}$ the intersection. We now proceed to define Herbrand interpretations and states.

Definition 4.2
A *Herbrand interpretation I of a program* P assigns to each individual constant c of P, the element $I(\mathsf{c}) = \mathsf{c}$, and to each predicate constant $\mathsf{p} : \pi$ of P, an element $I(\mathsf{p}) \in [\![\pi]\!]$.

We will denote the set of Herbrand interpretations of a program P with H_{P}. We define a partial order on H_{P} as follows: for all $I, J \in H_{\mathsf{P}}$, $I \leq J$ iff for every predicate constant $\mathsf{p} : \pi$ that appears in P, $I(\mathsf{p}) \leq_\pi J(\mathsf{p})$. The following proposition demonstrates that the space of interpretations is a complete lattice. This is an easy consequence of Proposition 4.1.

Proposition 4.2
Let P be a program. Then, $(H_{\mathsf{P}}, \leq)$ is a complete lattice.

Definition 4.3
A *Herbrand state s* of a program P is a function that assigns to each argument variable R of type ρ, an element $s(\mathsf{R}) \in [\![\rho]\!]$. We denote the set of Herbrand states with S_{P}.

In the following, $s[\mathsf{R}_1/d_1, \ldots, \mathsf{R}_n/d_n]$ is used to denote a state that is identical to s the only difference being that the new state assigns to each R_i the corresponding value d_i; for brevity, we will also denote it by $s[\overline{\mathsf{R}}/\overline{d}]$.

We proceed to define the (two-valued) semantics of $\mathcal{HOL}$ expressions and bodies.

Definition 4.4
Let P be a program, I a Herbrand interpretation of P, and s a Herbrand state. Then, the semantics of expressions and bodies is defined as follows:

1. $[\![\mathsf{R}]\!]_s(I) = s(\mathsf{R})$
2. $[\![\mathsf{c}]\!]_s(I) = I(\mathsf{c}) = \mathsf{c}$
3. $[\![\mathsf{p}]\!]_s(I) = I(\mathsf{p})$
4. $[\![(\mathsf{E}_1\ \mathsf{E}_2)]\!]_s(I) = [\![\mathsf{E}_1]\!]_s(I)\ [\![\mathsf{E}_2]\!]_s(I)$
5. $[\![(\mathsf{E}_1 \approx \mathsf{E}_2)]\!]_s(I) = \begin{cases} \mathit{true}, & \text{if } [\![\mathsf{E}_1]\!]_s(I) = [\![\mathsf{E}_2]\!]_s(I) \\ \mathit{false}, & \text{otherwise} \end{cases}$

6. $[\![(\sim \mathsf{E})]\!]_s(I) = \begin{cases} \mathit{true}, & \text{if } [\![\mathsf{E}]\!]_s(I) = \mathit{false} \\ \mathit{false}, & \text{otherwise} \end{cases}$
7. $[\![(\mathsf{E}_1 \wedge \cdots \wedge \mathsf{E}_m)]\!]_s(I) = \bigwedge_{\leq_o}\{[\![\mathsf{E}_1]\!]_s(I), \ldots, [\![\mathsf{E}_m]\!]_s(I)\}$

We can now formally define the notion of *model* for $\mathcal{HOL}$ programs.

Definition 4.5
Let P be a program and M be a two-valued Herbrand interpretation of P. Then, M is a *two-valued Herbrand model* of P iff for every rule $\mathsf{p}\ \overline{\mathsf{R}} \leftarrow \mathsf{B}$ in P and for every Herbrand state s, $[\![\mathsf{B}]\!]_s(M) \leq_o [\![\mathsf{p}\ \overline{\mathsf{R}}]\!]_s(M)$.

Since we have a mechanism to evaluate bodies of rules, we can define the *immediate consequence operator* for $\mathcal{HOL}$ programs, which generalizes the corresponding operator for classical (first-order) logic programs of (van Emden and Kowalski 1976).

Definition 4.6
Let P be a program. The mapping $T_{\mathsf{P}} : H_{\mathsf{P}} \rightarrow H_{\mathsf{P}}$ is called the *immediate consequence operator for* P and is defined for every predicate constant $\mathsf{p} : \rho_1 \rightarrow \cdots \rightarrow \rho_n \rightarrow o$ and all $d_1 \in [\![\rho_1]\!], \ldots, d_n \in [\![\rho_n]\!]$, as: $T_{\mathsf{P}}(I)(\mathsf{p})\ \overline{d} = \bigvee_{\leq_o}\{[\![\mathsf{B}]\!]_{s[\overline{\mathsf{R}}/\overline{d}]}(I) \mid s \in S_{\mathsf{P}} \text{ and } (\mathsf{p}\ \overline{\mathsf{R}} \leftarrow \mathsf{B}) \text{ in } \mathsf{P}\}$.

Since a program may contain negation, T_{P} is not necessarily monotone. In fact, perhaps somewhat surprisingly, T_{P} can even be non-monotone for negation-free programs such as `p ← r(p)`, where `p` is of type o and `r` is a predicate constant of type $o \rightarrow o$.[1]

As expected, T_{P} characterizes the models of P, as the following proposition suggests.

Proposition 4.3
Let P be a program and $I \in H_{\mathsf{P}}$. Then, I is a model of P iff I is a pre-fixpoint of T_{P} (i.e., $T_{\mathsf{P}}(I) \leq I$).

5 The Three-Valued Semantics of $\mathcal{HOL}$

In this section we define an alternative semantics for $\mathcal{HOL}$ types and expressions, based on a three-valued truth space. As in first-order logic programming, the purpose of the third truth value is to assign meaning to programs that contain circularities through negation. Since we are dealing with higher-order logic programs, we must define three-valued relations at all orders of the type hierarchy. These three-valued relations are functions that take two-valued arguments and return a three-valued truth result.

Due to the three-valuedness of our base domain o, all our domains inherit two distinct ordering relations, namely $\leq$ (the *truth ordering*) and $\preceq$ (the *precision ordering*).

Definition 5.1
Let P be a program. We define the (three-valued) meaning of a type with respect to U_{P}, as follows:

[1] To see the non-monotonicity of T_{P} for this program, consider an interpretation I_0 which assigns to `p` the value *false* and to `r` the negation operation $\mathit{neg} : o \rightarrow o : \mathit{true} \mapsto \mathit{false}, \mathit{false} \mapsto \mathit{true}$. Consider also an interpretation I_1 which is identical to I_0 the only difference being that it assigns to `p` the value *true*. It can be verified that $I_0 \leq I_1$ but $T_{\mathsf{P}}(I_0) \not\leq T_{\mathsf{P}}(I_1)$.

- $[\![o]\!]^*_{U_\mathsf{P}} = \{\mathit{false}, \mathit{undef}, \mathit{true}\}$. The partial order $\leq_o$ is the one induced by the ordering $\mathit{false} <_o \mathit{undef} <_o \mathit{true}$; the partial order $\preceq_o$ is the one induced by the ordering $\mathit{undef} \prec_o \mathit{false}$ and $\mathit{undef} \prec_o \mathit{true}$.
- $[\![\iota]\!]^*_{U_\mathsf{P}} = U_\mathsf{P}$. The partial order $\leq_\iota$ is defined as $d \leq_\iota d$ for all $d \in U_\mathsf{P}$. The partial order $\preceq_\iota$ is also defined as $d \preceq_\iota d$ for all $d \in U_\mathsf{P}$.
- $[\![\rho \to \pi]\!]^*_{U_\mathsf{P}} = [\![\rho]\!]_{U_\mathsf{P}} \to [\![\pi]\!]^*_{U_\mathsf{P}}$. The partial order $\leq_{\rho\to\pi}$ is defined as follows: for all $f, g \in [\![\rho \to \pi]\!]^*_{U_\mathsf{P}}$, $f \leq_{\rho\to\pi} g$ iff $f(d) \leq_\pi g(d)$ for all $d \in [\![\rho]\!]_{U_\mathsf{P}}$. The partial order $\preceq_{\rho\to\pi}$ is defined as follows: for all $f, g \in [\![\rho \to \pi]\!]^*_{U_\mathsf{P}}$, $f \preceq_{\rho\to\pi} g$ iff $f(d) \preceq_\pi g(d)$ for all $d \in [\![\rho]\!]_{U_\mathsf{P}}$.

We omit subscripts when unnecessary. It can be easily verified that for every ρ it holds $[\![\rho]\!] \subseteq [\![\rho]\!]^*$. In other words, every two-valued element is also a three-valued one. Moreover, the $\leq_\rho$ ordering in the above definition is an extension of the $\leq_\rho$ ordering in Definition 4.1.

Proposition 5.1
For every predicate type π, $([\![\pi]\!]^*, \leq_\pi)$ is a complete lattice and $([\![\pi]\!]^*, \preceq_\pi)$ is a complete meet-semilattice (i.e., every non-empty subset of $[\![\pi]\!]^*$ has a $\preceq_\pi$-greatest lower bound).

We denote by $\bigvee_{\leq_\pi}$ and $\bigwedge_{\leq_\pi}$ the lub and glb operations of the lattice $([\![\pi]\!]^*, \leq_\pi)$; it can easily be verified that these operations are extensions of the corresponding operations implied by Proposition 4.1. We denote by $\bigwedge_{\preceq_\pi}$ the glb in $([\![\pi]\!]^*, \preceq_\pi)$. Just like how a two-valued interpretation of a predicate π of type $\rho_1 \to \cdots \to \rho_n \to o$ can be viewed as a *set*, an element of $[\![\pi]\!]^*$ can be viewed as a *partial set*, assigning to each tuple in $[\![\rho_1]\!] \times \cdots \times [\![\rho_n]\!]$ one of three truth values (*true*, meaning the tuple is *in* the set, *false* meaning it is not in the set, or *undef* meaning it is not determined if it is in the set or not). This explains why the *arguments* are interpreted classically: a partial set decides for each *actual* (i.e., two-valued) object whether it is in the set or not; it does not make statements about *partial* (i.e., three-valued) objects. Due to the fact that the arguments of relations are interpreted classically, the definition of Herbrand states that we use below, is the same as that of Definition 4.3. A *three-valued Herbrand interpretation* is defined analogously to a two-valued one (Definition 4.2), the only difference being that the meaning of a predicate constant $\mathsf{p} : \pi$ is now an element of $[\![\pi]\!]^*_{U_\mathsf{P}}$. We will use caligraphic fonts (e.g., $\mathcal{I}, \mathcal{J}$) to differentiate three-valued interpretations from two-valued ones. The set of all three-valued Herbrand interpretations is denoted by $\mathcal{H}_\mathsf{P}$. Since $[\![\pi]\!] \subseteq [\![\pi]\!]^*$ it also follows that $H_\mathsf{P} \subseteq \mathcal{H}_\mathsf{P}$.

Definition 5.2
Let P be a program. We define the partial orders $\leq$ and $\preceq$ on $\mathcal{H}_\mathsf{P}$ as follows: for all $\mathcal{I}, \mathcal{J} \in \mathcal{H}_\mathsf{P}$, $\mathcal{I} \leq \mathcal{J}$ (respectively, $\mathcal{I} \preceq \mathcal{J}$) iff for every predicate type π and for every predicate constant $\mathsf{p} : \pi$ of P, $\mathcal{I}(\mathsf{p}) \leq_\pi \mathcal{J}(\mathsf{p})$ (respectively, $\mathcal{I}(\mathsf{p}) \preceq_\pi \mathcal{J}(\mathsf{p})$).

Definition 5.3
Let P be a program, $\mathcal{I}$ a three-valued Herbrand interpretation of P, and s a Herbrand state. The *three-valued semantics* of expressions and bodies is defined as follows:

1. $[\![\mathsf{R}]\!]^*_s(\mathcal{I}) = s(\mathsf{R})$
2. $[\![\mathsf{c}]\!]^*_s(\mathcal{I}) = \mathcal{I}(\mathsf{c}) = \mathsf{c}$
3. $[\![\mathsf{p}]\!]^*_s(\mathcal{I}) = \mathcal{I}(\mathsf{p})$

4. $[\![(\mathsf{E}_1\ \mathsf{E}_2)]\!]^*_s(\mathcal{I}) = \bigwedge_{\preceq_\pi}\{[\![\mathsf{E}_1]\!]^*_s(\mathcal{I})(d) \mid d \in [\![\rho]\!], [\![\mathsf{E}_2]\!]^*_s(\mathcal{I}) \preceq_\rho d\}$, for $\mathsf{E}_1 : \rho \to \pi$ and $\mathsf{E}_2 : \rho$
5. $[\![(\mathsf{E}_1 \approx \mathsf{E}_2)]\!]^*_s(\mathcal{I}) = \begin{cases} \mathit{true}, & \text{if } [\![\mathsf{E}_1]\!]^*_s(\mathcal{I}) = [\![\mathsf{E}_2]\!]^*_s(\mathcal{I}) \\ \mathit{false}, & \text{otherwise} \end{cases}$
6. $[\![(\sim \mathsf{E})]\!]^*_s(\mathcal{I}) = ([\![\mathsf{E}]\!]^*_s(\mathcal{I}))^{-1}$, with $\mathit{true}^{-1} = \mathit{false}$, $\mathit{false}^{-1} = \mathit{true}$ and $\mathit{undef}^{-1} = \mathit{undef}$
7. $[\![(\mathsf{E}_1 \wedge \cdots \wedge \mathsf{E}_m)]\!]^*_s(I) = \bigwedge_{\leq_o}\{[\![\mathsf{E}_1]\!]^*_s(I), \ldots, [\![\mathsf{E}_m]\!]^*_s(I)\}$

Item 4 is perhaps the most noteworthy. To evaluate an expression $(\mathsf{E}_1\ \mathsf{E}_2)$, we cannot just take $[\![\mathsf{E}_1]\!]^*_s(\mathcal{I})$, which is a function $[\![\rho]\!]_{U_\mathsf{P}} \to [\![\pi]\!]^*_{U_\mathsf{P}}$, and apply it to $[\![\mathsf{E}_2]\!]^*_s(\mathcal{I})$, which is of type $[\![\rho]\!]^*_{U_\mathsf{P}}$. Instead, we apply $[\![\mathsf{E}_1]\!]^*_s(\mathcal{I})$ to all "two-valued extensions" of $[\![\mathsf{E}_2]\!]^*_s(\mathcal{I})$ and take the least precise element approximating all those results. Our definition ensures that if $[\![\mathsf{E}_2]\!]^*_s(\mathcal{I})$ is a partial object, the result of the application is the most precise outcome achievable by using information from all the two-valued extensions of the argument.

Application is always well-defined, i.e., the set of two-valued extensions of a three-valued element is always non-empty, as the following lemma suggests.

Lemma 5.1
For every argument type ρ and $d^* \in [\![\rho]\!]^*$, there exists $d \in [\![\rho]\!]$ such that $d^* \preceq_\rho d$.

Moreover, as the following lemma suggests, the above semantics (Definition 5.3) is compatible with the standard semantics (see, Definition 4.4) when restricted to two-valued interpretations.

Lemma 5.2
Let P be a program, $I \in H_\mathsf{P}$ and $s \in S_\mathsf{P}$. Then, for every expression E, $[\![\mathsf{E}]\!]_s(I) = [\![\mathsf{E}]\!]^*_s(I)$.

This three-valued valuation of bodies, immediately gives us a notion of three-valued model as well as a three-valued immediate consequence operator.

Definition 5.4
Let P be a program and $\mathcal{M}$ be a three-valued Herbrand interpretation of P. Then, $\mathcal{M}$ is a *three-valued Herbrand model* of P iff for every rule $\mathsf{p}\ \overline{\mathsf{R}} \leftarrow \mathsf{B}$ in P and for every Herbrand state s, $[\![\mathsf{B}]\!]^*_s(\mathcal{M}) \leq_o [\![\mathsf{p}\ \overline{\mathsf{R}}]\!]^*_s(\mathcal{M})$.

For the special case where in the above definition $\mathcal{M} \in H_\mathsf{P}$, it is clear from Lemma 5.2 that Definition 5.4 coincides with Definition 4.5.

Definition 5.5
Let P be a program. The *three-valued immediate consequence operator* $\mathcal{T}_\mathsf{P} : \mathcal{H}_\mathsf{P} \to \mathcal{H}_\mathsf{P}$ is defined for every predicate constant $\mathsf{p} : \rho_1 \to \cdots \to \rho_n \to o$ in P and all $d_1 \in [\![\rho_1]\!], \ldots, d_n \in [\![\rho_n]\!]$, as: $\mathcal{T}_\mathsf{P}(\mathcal{I})(\mathsf{p})\ \overline{d} = \bigvee_{\leq_o}\{[\![\mathsf{B}]\!]^*_{s[\overline{\mathsf{R}}/\overline{d}]}(\mathcal{I}) \mid s \in S_\mathsf{P} \text{ and } (\mathsf{p}\ \overline{\mathsf{R}} \leftarrow \mathsf{B}) \text{ in } \mathsf{P}\}$.

The proof of the following proposition is similar to that of Proposition 4.3.

Proposition 5.2
Let P be a program and $\mathcal{I} \in \mathcal{H}_\mathsf{P}$. Then, $\mathcal{I}$ is a three-valued model of P if and only if $\mathcal{I}$ is a pre-fixpoint of $\mathcal{T}_\mathsf{P}$.

6 Approximation Fixpoint Theory and the Stable Model Semantics

We now define the *two-valued* and *three-valued stable models* of a program P. To achieve this goal, we use the machinery of *approximation fixpoint theory (AFT)* (Denecker et al. 2004). In the rest of this section, we assume the reader has a basic familiarity with (Denecker et al. 2004). As mentioned before, the two-valued immediate consequence operator $T_{\mathsf{P}} : H_{\mathsf{P}} \to H_{\mathsf{P}}$ can be non-monotone, meaning it is not clear what its fixpoints of interest would be. The core idea behind AFT is to "approximate" T_{P} with a function A_{P} which is $\preceq$-monotone. We can then study the fixpoints of A_{P}, which shed light to the fixpoints of T_{P}. While we already have such a candidate function, namely $\mathcal{T}_{\mathsf{P}}$, AFT requires a function that works on *pairs* (of interpretations). Therefore, we show that there is a simple isomorphism between three-valued relations and (appropriate) pairs of two-valued ones. This isomorphism also exists between three-valued interpretations and (appropriate) pairs of two-valued ones.

Definition 6.1
Let $(L, \leq)$ be a complete lattice. We define $L^c = \{(x, y) \in L \times L \mid x \leq y\}$. Moreover, we define the relations $\leq$ and $\preceq$, so that for all $(x, y), (x', y') \in L^c$: $(x, y) \leq (x', y')$ iff $x \leq x'$ and $y \leq y'$, and $(x, y) \preceq (x', y')$ iff $x \leq x'$ and $y' \leq y$.

Proposition 6.1
For every predicate type π there exists a bijection $\tau_\pi : [\![\pi]\!]^* \to [\![\pi]\!]^c$ with inverse $\tau_\pi^{-1} : [\![\pi]\!]^c \to [\![\pi]\!]^*$, that both preserve the orderings $\leq$ and $\preceq$ of elements between $[\![\pi]\!]^*$ and $[\![\pi]\!]^c$. Moreover, there exists a bijection $\tau : \mathcal{H}_{\mathsf{P}} \to H_{\mathsf{P}}^c$ with inverse $\tau^{-1} : H_{\mathsf{P}}^c \to \mathcal{H}_{\mathsf{P}}$, that both preserve the orderings $\leq$ and $\preceq$ between $\mathcal{H}_{\mathsf{P}}$ and H_{P}^c.

When viewing elements of $[\![\pi]\!]^*$ as partial sets and elements of $[\![\pi]\!]$ as sets, the isomorphism maps a partial set onto the pair with first component all the *certain* elements of the partial set (those mapped to *true*) and second component all the *possible* elements (those mapped to *true* or *undef*). Using these bijections, we can now define A_{P} which, as we demonstrate, is an approximator of T_{P}. Intuitively, A_{P} is the "pair version of $\mathcal{T}_{\mathsf{P}}$" (instead of handling three-valued interpretations, it handles pairs of two-valued ones).

Definition 6.2
For each program P, $A_{\mathsf{P}} : H_{\mathsf{P}}^c \to H_{\mathsf{P}}^c$ is defined as $A_{\mathsf{P}}(I, J) = \tau(\mathcal{T}_{\mathsf{P}}(\tau^{-1}(I, J)))$.

Lemma 6.1
Let P be a program. In the terminology of Denecker et al. (2004), $A_{\mathsf{P}} : H_{\mathsf{P}}^c \to H_{\mathsf{P}}^c$ is a consistent approximator of T_{P}.

Since A_{P} is "the pair version of $\mathcal{T}_{\mathsf{P}}$", it is not a surprise that it also captures all the three-valued models of P.

Lemma 6.2
Let P be a program and $(I, J) \in H_{\mathsf{P}}^c$. Then, (I, J) is a pre-fixpoint of A_{P} if and only if $\tau^{-1}(I, J)$ is a three-valued model of P.

Due to the above lemma, by stretching notation, when (I, J) is a pre-fixpoint of A_{P} we will also say that (I, J) is a model of P.

The power of AFT comes from the fact that once an approximator is defined, it immediately defines a whole range of semantics. In other words, there is no need to reinvent the wheel. The following definition summarizes the different induced semantics.

Definition 6.3
Let P be a program, A_{P} the induced approximator and $I, J \in H_{\mathsf{P}}$. We call:

- (I, J) a *three-valued supported model of* P if it is a fixpoint of A_{P};
- (I, J) a *three-valued stable model of* P if it is a stable fixpoint of A_{P}; that is, if $I = \text{lfp}\, A_{\mathsf{P}}(\cdot, J)_1$ and $J = \text{lfp}\, A_{\mathsf{P}}(I, \cdot)_2$, where $A_{\mathsf{P}}(\cdot, J)_1$ is the function that maps an interpretation X to the first component of $A_{\mathsf{P}}(X, J)$, and similarly for $A_{\mathsf{P}}(I, \cdot)_2$;
- (I, J) the *Kripke-Kleene model of* P if it is the $\preceq$-least fixpoint of A_{P};
- (I, J) the *well-founded model of* P if it is the well-founded fixpoint of A_{P}, i.e., if it is the $\preceq$-least three-valued stable model.

Following the correspondence indicated by the isomorphism between pairs and three-valued interpretations, we will also call $\mathcal{M}$ a *three-valued stable model* of P if $\tau(\mathcal{M})$ is a *three-valued stable model of* P. If $\mathcal{M}$ is a three-valued stable model and $\mathcal{M} \in H_{\mathsf{P}}$ (i.e., $\mathcal{M}$ is actually two-valued), we will call $\mathcal{M}$ a *stable model* of P.

7 Properties of the Stable Model Semantics

In this section we discuss various properties of the stable model semantics of higher-order logic programs, which demonstrate that the proposed approach is indeed an extension of classical stable models. In the following results we use the term "classical stable models" to refer to stable models in the sense of (Gelfond and Lifschitz 1988), "classical three-valued stable models" to refer to stable models in the sense of (Przymusinski 1990) and the term "(three-valued) stable models" to refer to the present semantics.

Theorem 7.1
Let P be a propositional logic program. Then, $\mathcal{M}$ is a (three-valued) stable model of P iff $\mathcal{M}$ is a classical (three-valued) stable model of P.

A crucial property of classical (three-valued) stable models is that they are *minimal* Herbrand models (Gelfond and Lifschitz 1988, Theorem 1) and (Przymusinski 1990, Proposition 3.1). This property is preserved by our extension.

Theorem 7.2
All (three-valued) stable models of a $\mathcal{HOL}$ program P are $\leq$-minimal models of P.

It is a well-known result in classical logic programming that if the well-founded model of a first-order program is two-valued, then that model is its unique classical stable model (Van Gelder et al. 1988, Corollary 5.6). This property generalizes in our setting.

Theorem 7.3
Let P be a $\mathcal{HOL}$ program. If the well-founded model of P is two-valued, then this is also its unique stable model.

A broadly studied subclass of first-order logic programs with negation, is that of *stratified logic programs* (Apt et al. 1988). It is a well-known result that if a logic program is stratified, then it has a two-valued well-founded model which is also its unique classical stable model (Gelfond and Lifschitz 1988, Corollary 2). We extend the class of stratified programs to the higher-order case and generalize the aforementioned result.

Definition 7.1
A $\mathcal{HOL}$ program P is called *stratified* if there is a function S mapping predicate constants to natural numbers, such that for each rule $\mathsf{p}\ \overline{\mathsf{R}} \leftarrow \mathsf{L}_1 \wedge \cdots \wedge \mathsf{L}_m$ and any $i \in \{1, \ldots, m\}$:

- $S(\mathsf{q}) \leq S(\mathsf{p})$ for every predicate constant q occurring in L_i.
- If L_i is of the form $\sim\mathsf{E}$, then $S(\mathsf{q}) < S(\mathsf{p})$ for each predicate constant q occurring in E.
- For any subexpression of L_i of the form $(\mathsf{E}_1\ \mathsf{E}_2)$, $S(\mathsf{q}) < S(\mathsf{p})$ for every predicate constant q occurring in E_2.

For readers familiar with the standard definitions of stratification in first-order logic programs, the last item might be somewhat surprising. What it says is that the stratification function should not only increase because of negation, but also because of higher-order predicate application. The intuitive reason for this is that (as also noted in the introduction of the present paper) one can define a higher-order predicate which is identical to negation, for example, by writing `neg P` $\leftarrow\sim$`P`. As a consequence, it is reasonable to assume that predicates occurring inside an application of `neg` should be treated similarly to predicates appearing inside the negation symbol.

Theorem 7.4
Let P be a stratified $\mathcal{HOL}$ program. Then, the well-founded model of P is two-valued.

By the above theorem and Theorem 7.3, every stratified $\mathcal{HOL}$ program has a unique two-valued stable model.

8 Additional Examples

In this section we present two examples of how higher-order logic programming can be used. First, we showcase reasoning problems arising from the field of abstract argumentation, next we model a PSPACE-complete problem known as Generalized Geography.

8.1 Abstract Argumentation

In what follows, we present a set of standard definitions from the field of abstract argumentation (Dung 1995). Listing 3 contains direct translations of these definitions into our framework; the line numbers with each definition refer to Listing 3. Listing 4 illustrate how these definitions can be used to solve reasoning problems with argumentation.

An *abstract argumentation framework* (AF) Θ is a directed graph (A, E) in which the nodes A represent arguments and the edges in E represent attacks between arguments. We say that a *attacks* b if $(a, b) \in E$. A set $S \subseteq A$ *attacks* a if some $s \in S$ attacks a (Line 2). A set $S \subseteq A$ *defends* a if it attacks all attackers of a (Line 4). An *interpretation* of an AF $\Theta = (A, E)$ is a subset S of A. There exist many different semantics of AFs that each define different sets of acceptable arguments according to different standards or intuitions. The major semantics for argumentation frameworks can be formulated using two operators: the *characteristic function* F_Θ (Line 5) mapping an interpretation S to

$$F_\Theta(S) = \{a \in A \mid S \text{ defends } a\}$$

and the operator U_Θ (U stands for unattacked; Line 6) that maps an interpretation S to

$$U_\Theta(S) = \{a \in A \mid a \text{ is not attacked by } S\}.$$

The *grounded extension* of Θ is defined inductively as the set of all arguments defended by the grounded extension (Line 8), or alternatively, as the least fixpoint of F_Θ, which is a monotone operator. The operator U_Θ is an anti-monotone operator; its fixpoints are called *stable extensions* of Θ (Line 9). An interpretation S is *conflict-free* if it is a postfixpoint of U_Θ (i.e., if $S \subseteq U_\Theta(S)$; Line 10). A *complete extension* is a conflict-free fixpoint of F_Θ (Line 11). An interpretation is *admissible* if it is a conflict-free postfixpoint of F_Θ (Line 12). A *preferred extension* is a $\subseteq$-maximal complete extension (Line 13).

Listing 4 shows how these definitions can be used for reasoning problems related to argumentation. There, we search for an argumentation framework with five elements where the grounded extension does not equal the intersection of all stable extensions.

Listing 3: Second-order definitions of abstract argumentation concepts.

```
1  % A is a set of arguments; E subset A x A is the attack relation
2  attacks A E S X ← (subset S A), (S Y), (E Y X)
3  nondefends A E S X ← (subset S A), (A Y), (E Y X), ∼(attacks A E S Y)
4  defends A E S X ← (subset S A), (A X), ∼(nondefends A E S X)
5  f A E S X ← (defends A E S X)
6  u A E S X ← (subset S A), (A X), ∼(attacks A E S X)
7  % grounded A E X means: X is an element of the grounded extension
8  grounded A E X  ← f A E (grounded A E) X
9  stable A E S ← (equal S (u A E S))
10 conflFree A E S ← (subset S (u A E S))
11 complete A E S ← (conflFree A E S), (equal S (f A E S))
12 admissable A E S  ← (conflFree A E S), (subset S (f A E S))
13 preferred A E S ←  maximal subset (complete A E) S
```

Listing 4: Toy reasoning problem for abstract argumentation.

```
arg a. arg b. arg c. arg d. arg e.
attacks X Y  ← arg X, arg Y, ∼(nattacks X Y).
nattacks X Y ← arg X, arg Y, ∼(attacks X Y).

ncautiousStable X ← arg X, stable arg attacks S, ∼(S X).
cautiousStable X ← arg X, ∼(ncautiousStable X).
p ← ∼p, equal cautiousStable (grounded arg attacks).
```

8.2 (Generalized) Geography

Generalized geography is a two-player game that is played on a graph. Two players take turn to form a simple path (i.e., a path without cycles) through the graph. The first player who can no longer extend the currently formed simple path loses the game. The question whether a given node in a given graph is a winning position in this game (i.e., whether there is a winning strategy) is well-known to be PSPACE-hard (see, e.g., the proof of Lichtenstein and Sipser (1980)). This game can be modelled in our language very compactly: Line 2 in Listing 5 states that `X` is a winning node in the game `V, E` if there is an outgoing edge from `X` that leads to a non-winning position in the induced graph obtained by removing `X` from `V`. This definition makes use of the notion of an induced subgraph, which has a very natural higher-order definition, which in turn makes use of various other generic predicates about sets (see Listings 2 and 6).

Listing 5: Winning positions in the Generalized Geography game.

```
% X is a winning position in the GG game (V,E)
winning V E X ← (E X Y), ∼(X ≈ Y), equal (remove V X) V',
    ↪ inducedGraph V E V' E',
    ↪ ∼(winning V' E' Y).
% (V',E') is the induced graph by restricting (V,E) to V'
inducedGraph V E V' E' ← subset V' V,
    ↪  equal E' (intersection E (square V'))
```

Listing 6: More generic definitions.

```
% X is in the union of P and Q
union P Q X ← P X.
union P Q X ← Q X.
% X is in the intersection of P and Q
intersection P Q X ← P X, Q X.
% Y is in the set obtained from P by removing X  (P \ {X})
remove P X Y ← P Y, ∼(X ≈ Y).
% (X,Y) is in the square of P (cartesian product of P with itself)
square P X Y ← P X, P Y.
```

9 Related and Future Work

There are many extensions of standard logic programming under the stable model semantics that are closely related to our current work. One of them is the extension of logic programming with *aggregates*, which most solvers nowadays support. Aggregates are special cases of second-order functions and have been studied using AFT (Pelov et al. 2007; Vanbesien et al. 2022) and in fact our semantics of application can be viewed as a generalization of the *ultimate approximating aggregates* of Pelov et al. (2007). Also, higher-order logic programs have been studied through this fixpoint theoretic lens. Dasseville et al. (2015) defined a logic for templates, which are second-order definitions, for which they use a well-founded semantics. This idea was generalized to arbitrary higher-order definitions in the next year (Dasseville et al. 2016). While they apply AFT in the same space of three-valued higher order functions as we do, a notable difference is that they use the so-called *ultimate approximator*, resulting in a semantics that does not coincide with the standard semantics for propositional programs whereas our semantics does (see Theorem 7.1).

In 2018, a well-founded semantics for higher-order logic programs was developed using AFT (Charalambidis et al. 2018a). There are two main ways in which that work differs from ours. The first, and arguably most important one, is how the three-valued semantics of types is defined. While in our framework $[\![\rho \to o]\!]^*$ consists of all functions from $[\![\rho]\!]$ to $[\![o]\!]^*$, in their framework $[\![\rho \to o]\!]^*$ would consist of all $\preceq$-monotonic functions from $[\![\rho]\!]^*$ to $[\![o]\!]^*$. This results, in their case, to more refined, but less precise, and more complicated, approximations. As a result, an extension of AFT needed to be developed to accommodate this. In our current work, we show that we can stay within standard AFT, but to achieve this, we needed to develop a new three-valued semantics of function application; see Item 4 in Definition 5.3. The formal relationship between the two approaches remains to be further investigated. The second way in which our work differs from that of Charalambidis et al. (2018a) is the treatment of existential predicate variables of type π that appear in the bodies of rules; we consider such variables to range over $[\![\pi]\!]$, while Charalambidis et al. allow them to range over $[\![\pi]\!]^*$. A consequence of this choice is that arguably, the well-founded semantics of (Charalambidis et al. 2018a) does not always behave as expected; even for simple non-recursive programs such as `p` $\leftarrow$ `R,` $\sim$`R`, the meaning of the defined predicates is not guaranteed to be two-valued. This is not an issue with their extension of AFT, but rather with the precise way their approximator is defined. While we believe it would be possible to solve this issue by changing the definition of the approximator in Charalambidis et al. (2018a), this issue needs to be further investigated. Importantly, we showed in Theorem 7.4 that issues such as the one just mentioned, cannot arise in our new semantics.

Recently, there have also been some extensions of logic programming that allow second-order quantification *over* answer sets. This idea was first referred to as *stable-unstable semantics* (Bogaerts et al. 2016) and later also as *quantified ASP* (Amendola et al. 2019). A related formalism is *ASP with quantifiers* (Fandinno et al. 2021), which can be thought of as a *prenex* version of quantified ASP, consisting of a single logic program preceded by a list of quantifications. A major advantage of those lines of work is that they come with efficient implementations (Janhunen 2022; Faber et al. 2023) and applications (Amendola et al. 2022; Bellusci et al. 2022; Fandinno et al. 2021). An advantage of true higher-order logic programming (which allows for defining higher-order predicates) is the potential for abstraction and reusability (following the lines of thought of the "templates" work referred to above). As an example, consider our max-clique application from Section 2. While it is perfectly possible to express this in stable-unstable semantics or quantified ASP, such encodings would have *two* definitions of what it means to be a clique: one for the actual clique to be found and one inside the oracle call that checks for the non-existence of a larger clique. In our approach, the definition of clique is given only once and used for these two purposes by giving it as an argument to the higher-order `maximal` predicate. Moreover, the definition of the `maximal` predicate can be reused in future applications where maximal (with respect to some given order) elements of some set are sought.

There are several future directions that we feel are worth pursuing. In particular, it would be interesting to investigate efficient implementation techniques for the proposed stable model semantics. As the examples of the paper suggest, even an implementation of second-order stable models, would give a powerful and expressive system. Another interesting research topic is the characterization of the expressive power of higher-order stable models. As proven in (Charalambidis et al. 2019), positive k-order Datalog programs over ordered databases, capture $(k-1)$-$\mathsf{EXPTIME}$ (for all $k \geq 2$). We believe that the addition of stable negation will result in greater expressiveness (for example, the ordering restriction on the database could be lifted), but this needs to be further investigated.

References

Amendola, G., Cuteri, B., Ricca, F., and Truszczynski, M. Solving problems in the polynomial hierarchy with ASP(Q). In *LPNMR* 2022, volume 13416 of *LNCS*, pp. 373–386. Springer.

Amendola, G., Ricca, F., and Truszczynski, M. 2019. Beyond NP: quantifying over answer sets. *Theory Pract. Log. Program.*, *19*, 5-6, 705–721.

Antic, C. 2020. Fixed point semantics for stream reasoning. *Artif. Intell.*, *288*, 103370.

Apt, K. R., Blair, H. A., and Walker, A. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming* 1988, pp. 89–148. Morgan Kaufmann.

Bellusci, P., Mazzotta, G., and Ricca, F. Modelling the outlier detection problem in ASP(Q). In *PADL* 2022, volume 13165 of *LNCS*, pp. 15–23. Springer.

Bogaerts, B. Weighted abstract dialectical frameworks through the lens of approximation fixpoint theory. In *AAAI* 2019, pp. 2686–2693. AAAI Press.

Bogaerts, B. and Cruz-Filipe, L. 2018. Fixpoint semantics for active integrity constraints. *Artif. Intell.*, *255*, 43–70.

Bogaerts, B. and Jakubowski, M. Fixpoint semantics for recursive SHACL. In *ICLP Technical Communications* 2021, volume 345 of *EPTCS*, pp. 41–47.

Bogaerts, B., Janhunen, T., and Tasharrofi, S. 2016. Stable-unstable semantics: Beyond NP with normal logic programs. *Theory Pract. Log. Program.*, *16*, 5-6, 570–586.

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

Charalambidis, A., Handjopoulos, K., Rondogiannis, P., and Wadge, W. W. 2013. Extensional higher-order logic programming. *ACM Trans. Comput. Log.*, *14*, 3, 21:1–21:40.

Charalambidis, A., Nomikos, C., and Rondogiannis, P. 2019. The expressive power of higher-order datalog. *Theory Pract. Log. Program.*, *19*, 5-6, 925–940.

Charalambidis, A., Rondogiannis, P., and Symeonidou, I. 2018a. Approximation fixpoint theory and the well-founded semantics of higher-order logic programs. *Theory Pract. Log. Program.*, *18*a, 3-4, 421–437.

Charalambidis, A., Rondogiannis, P., and Troumpoukis, A. 2018b. Higher-order logic programming: An expressive language for representing qualitative preferences. *Sci. Comput. Program.*, *155*b, 173–197.

Dasseville, I., van der Hallen, M., Bogaerts, B., Janssens, G., and Denecker, M. A compositional typed higher-order logic with definitions. In *ICLP (Technical Communications)* 2016, volume 52 of *OASIcs*, pp. 14:1–14:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Dasseville, I., van der Hallen, M., Janssens, G., and Denecker, M. 2015. Semantics of templates in a compositional framework for building logics. *Theory Pract. Log. Program.*, *15*, 4-5, 681–695.

Denecker, M., Marek, V. W., and Truszczynski, M. 2003. Uniform semantic treatment of default and autoepistemic logics. *Artif. Intell.*, *143*, 1, 79–122.

Denecker, M., Marek, V. W., and Truszczynski, M. 2004. Ultimate approximation and its application in nonmonotonic knowledge representation systems. *Inf. Comput.*, *192*, 1, 84–121.

Dung, P. M. 1995. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, *77*, 2, 321–358.

Faber, W., Mazzotta, G., and Ricca, F. 2023. An efficient solver for ASP(Q). *Theory Pract. Log. Program.*, *23*, 4, 948–964.

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

Fandinno, J., Laferrière, F., Romero, J., Schaub, T., and Son, T. C. 2021. Planning with incomplete information in quantified answer set programming. *Theory Pract. Log. Program.*, *21*, 5, 663–679.

Fitting, M. 2002. Fixpoint semantics for logic programming a survey. *Theor. Comput. Sci.*, *278*, 1-2, 25–51.

Gelfond, M. and Lifschitz, V. The stable model semantics for logic programming. In *ICLP/SLP* 1988, pp. 1070–1080. MIT Press.

JANHUNEN, T. Implementing stable-unstable semantics with ASPTOOLS and clingo. In *PADL* 2022, volume 13165 of *LNCS*, pp. 135–153. Springer.

LICHTENSTEIN, D. AND SIPSER, M. 1980. GO is polynomial-space hard. *J. ACM*, *27*, 2, 393–401.

PELOV, N., DENECKER, M., AND BRUYNOOGHE, M. 2007. Well-founded and stable semantics of logic programs with aggregates. *Theory Pract. Log. Program.*, *7*, 3, 301–353.

PRZYMUSINSKI, T. C. 1990. The well-founded semantics coincides with the three-valued stable semantics. *Fundam. Inform.*, *13*, 4, 445–463.

RONDOGIANNIS, P. AND SYMEONIDOU, I. 2017. The intricacies of three-valued extensional semantics for higher-order logic programs. *Theory Pract. Log. Program.*, *17*, 5-6, 974–991.

STRASS, H. 2013. Approximating operators and semantics for abstract dialectical frameworks. *Artif. Intell.*, *205*, 39–70.

VAN EMDEN, M. H. AND KOWALSKI, R. A. 1976. The semantics of predicate logic as a programming language. *J. ACM*, *23*, 4, 733–742.

VAN GELDER, A., ROSS, K. A., AND SCHLIPF, J. S. Unfounded sets and well-founded semantics for general logic programs. In *PODS* 1988, pp. 221–230. ACM.

VANBESIEN, L., BRUYNOOGHE, M., AND DENECKER, M. 2022. Analyzing semantics of aggregate answer set programming using approximation fixpoint theory. *Theory Pract. Log. Program.*, *22*, 4, 523–537.