TPLP **26** (1-2): 80–94, 2026. © The Author(s), 2026. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution and reproduction, provided the original article is properly cited.

doi:10.1017/S147106842610057X First published online 3 July 2026

Equilibrium Semantics and Strong Equivalence for Higher-Order Logic Programs

ANGELOS CHARALAMBIDIS
Informatics and Telematics, Harokopio University of Athens, Greece
(*e-mail:* acharal@hua.gr)

GIANNOS CHATZIAGAPIS
Informatics and Telecommunications, National and Kapodistrian University of Athens, Greece
(*e-mail:* gchatziagap@di.uoa.gr)

BABIS KOSTOPOULOS
Informatics and Telematics, Harokopio University of Athens, Greece
(*e-mail:* kostbabis@hua.gr)

PANOS RONDOGIANNIS
Informatics and Telecommunications, National and Kapodistrian University of Athens, Greece
(*e-mail:* prondo@di.uoa.gr)

submitted 30 May 2026 UTC; revised 30 May 2026 UTC; accepted 1 June 2026 UTC

Abstract

One of the most significant achievements of equilibrium logic was the characterization of strong equivalence, a property crucial for program transformation and optimization in answer set programming (ASP). While ASP has recently been extended to a higher-order setting to enhance its expressive power, the lack of a comparable purely logical foundation has made verifying strong equivalence for higher-order programs or even proving the correctness of simple program transformations, a difficult challenge. This paper addresses this gap by developing a logical semantics for higher-order ASP by extending the equilibrium logic framework. Within this extended framework, we demonstrate that every stratified higher-order logic program possesses a unique equilibrium model. Moreover, we establish definability results demonstrating that the syntax of our higher-order language is sufficiently expressive to capture its semantic domains. Finally, and most importantly, we generalize the classical theorem of strong equivalence to the higher-order setting: we prove that two programs are strongly equivalent if and only if they share the same higher-order models.

Keywords: higher-order logic programming, answer set programming, strong equivalence

1 Introduction

A turning-point for answer set programming (ASP) was the purely logical characterization of its semantics by Pearce (1996, 1999) based on the logic of here-and-there (HT). Pearce developed a nonmonotonic system, called *equilibrium logic*, which can be

https://doi.org/10.1017/S147106842610057X Published online by Cambridge University Press

described as a form of minimal model reasoning in HT, and demonstrated that the equilibrium models of any program coincide with its answer sets. One of the most seminal successes of equilibrium logic was the logical characterization of the notion of *strong equivalence* of logic programs by Lifschitz *et al.* (2001), an important and useful concept which has received considerable research attention in the ASP community.

Recently, a higher-order extension of ASP was introduced (Bogaerts *et al.* 2024) using the abstract framework of approximation fixpoint theory (AFT) (Denecker *et al.* 2000, 2004). The introduction of higher-order ASP is driven by more than just theoretical curiosity: recently, many nontrivial extensions of ASP have been investigated in order to enhance its power (Bogaerts *et al.* 2016; Amendola *et al.* 2019; Fandinno *et al.* 2021), and thus it was a natural step to investigate the potential of developing a higher-order extension of the paradigm. Not surprisingly, higher-order ASP demonstrates impressive expressive power: as shown in the work of Bogaerts *et al.* (2024) several difficult computational problems were modeled in higher-order ASP, such as for example the Generalized Geography two-player game, which is a well-known (Lichtenstein and Sipser 1980) PSPACE-complete problem. The increased expressive power of higher-order ASP was theoretically confirmed very recently: as demonstrated by Charalambidis *et al.* (2025), $(k+1)$-order ASP captures co-(k-NEXP) using cautious reasoning and k-NEXP using brave reasoning. So, even the second-order fragment is extremely powerful and can express problems that are beyond the capabilities of standard ASP systems. Although expressive and quite promising, the idea of higher-order ASP lacks in one important respect: due to the fact that its current characterization by Bogaerts *et al.* (2024) is based on a fixpoint construction, proving the correctness of even simple transformations involves laborious and intricate double inductive arguments (see e.g., (Charalambidis *et al.* 2025, Appendix C)). In particular, defining the strong equivalence of higher-order ASP programs based on the fixpoint semantics from Bogaerts *et al.* (2024), appears to be a formidable challenge.

Based on the above discussion, a natural next step is to seek a logical characterization of higher-order ASP, which would allow notions such as strong equivalence to be defined in a simple way and reasoning about programs to be performed in a purely logical manner. Such a characterization requires extending HT to higher-order types and accordingly generalizing the framework of equilibrium logic. But what form should these higher-order HT relations have? Should they be arbitrary, or should they obey specific restrictions to capture the computational nature of logic programs? Drawing on well-known concepts from denotational semantics (Tennent 1991; Gunter 1993), we adopt the view that meaningful higher-order relations in a computational context should be monotonic over appropriate semantic domains. We introduce *justification monotonic* relations which prove to be a cornerstone of our work, essential for generalizing the characterization of strong equivalence to higher-order programs. The paper's main contributions are outlined below:

- We develop a purely logical semantics for the higher-order logic programming language $\mathcal{HOL}$. Our approach extends the 3-valued truth domain of HT to the full type hierarchy of $\mathcal{HOL}$, restricting the domain to *justification-monotonic* relations

https://doi.org/10.1017/S147106842610057X Published online by Cambridge University Press

at each level of the hierarchy. Within this framework, we define *equilibrium models* for $\mathcal{HOL}$ as the *total* interpretations that are minimal with respect to the justification ordering. We argue that this restriction is necessary: adhering to a standard model of types - where the arrow type constructor denotes the set of all 3-valued relations - yields counterintuitive results even for trivial programs. Finally, we demonstrate that under this proposed semantics, *stratified* $\mathcal{HOL}$ programs always possess a unique equilibrium model.

- We examine *definability* concepts regarding the proposed semantics. More specifically, we demonstrate that every *total* justification-monotonic relation can be represented by a stratified $\mathcal{HOL}$ program. Moreover, every *non-total* justification-monotonic relation can be "captured" (in a sense that will be precisely defined in Section 4) by a stratified $\mathcal{HOL}$ program. Such definability results have a foundational significance: they demonstrate that the syntactic elements of $\mathcal{HOL}$ are strong enough to express all the elements of its semantic domains. To use Robin Milner's words (Milner 1977), definability results suggest that the proposed semantics is not "over-generous".
- We generalize the classical proof of propositional strong equivalence to the world of higher-order logic programs. More specifically, we demonstrate that two $\mathcal{HOL}$ programs are strongly equivalent iff they have the same $\mathcal{HOL}$ models. Our proof relies significantly on the justification-monotonic nature of the proposed semantics and the derived definability results for $\mathcal{HOL}$. This result paves the road for further investigations on transformations and verification of higher-order logic programs, an area that at present is largely unexplored.

We believe that the present work can be the starting point for further investigations not only in the logical foundations of higher-order logic programs but also in their practical applications, extending in this way significantly the scope of the ASP paradigm.

2 $\mathcal{HOL}$: A higher-order logic programming language

In this section we define the syntax of the language $\mathcal{HOL}$ that we use throughout the paper. Actually, $\mathcal{HOL}$ is based on a simple type system with one *base* type, namely o, the Boolean domain.

Definition 2.1.
The types of $\mathcal{HOL}$ are denoted by π (and its subscripted versions), and are defined as:

$$\pi := o \mid (\pi_1 \to \pi_2)$$

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 $\pi_1 \to \cdots \to \pi_n \to o$, $n \geq 0$ (for $n = 0$ we assume that $\pi = o$). We proceed by defining the syntax of $\mathcal{HOL}$.

Definition 2.2.
The *alphabet* of $\mathcal{HOL}$ consists of the following: *predicate variables* of every type π (denoted by capital letters such as $\mathsf{P}, \mathsf{Q}, \mathsf{R}, \ldots$); *predicate constants* of every type π (denoted by lowercase letters such as $\mathsf{p}, \mathsf{q}, \mathsf{r}, \ldots$); the *conjunction* constant $\wedge$ of type $o \to o \to o$; and the *negation* constant $\sim$ of type $o \to o$.

https://doi.org/10.1017/S147106842610057X Published online by Cambridge University Press

Definition 2.3.
The expressions and *literals* of $\mathcal{HOL}$ are defined as follows. Every predicate variable or constant is an expression; if E_1 is an expression of type $\pi_1 \to \pi_2$ and E_2 an expression of type π_1 then $(\mathsf{E}_1\ \mathsf{E}_2)$ is an expression of type π_2. If E is an expression of type o then E is also a (positive) *literal* of type o; if E is an expression of type o then $(\sim\mathsf{E})$ is a (negative) *literal* of type o.

We will omit parentheses when no confusion arises and in some other cases we may use additional ones if we feel that this enhances readability; so, for example, when a predicate constant $\mathtt{p}$ is unary, we may write $\mathtt{p(R)}$ instead of $\mathtt{p\ R}$. Literals will be denoted by L and its subscripted versions. We write $\mathsf{E} : \pi$ to denote that an expression E has type π.

Definition 2.4.
A rule of $\mathcal{HOL}$ is a syntactic construct of the form $\mathtt{p}\ \mathsf{R}_1 \cdots \mathsf{R}_n \leftarrow \mathsf{L}_1 \wedge \cdots \wedge \mathsf{L}_m$, where $\mathtt{p}$ is a predicate constant of type $\pi_1 \to \cdots \to \pi_n \to o$, $\mathsf{R}_1, \ldots, \mathsf{R}_n$ are distinct variables of types $\pi_1, \ldots, \pi_n$ respectively and the L_i are literals. The term $\mathtt{p}\ \mathsf{R}_1 \cdots \mathsf{R}_n$ is the *head* of the rule and $\mathsf{L}_1 \wedge \cdots \wedge \mathsf{L}_m$ is its *body*. A *program* P of $\mathcal{HOL}$ is a finite set of rules.

We will often follow the common logic programming notation and write $\mathsf{L}_1, \ldots, \mathsf{L}_m$ instead of $\mathsf{L}_1 \wedge \cdots \wedge \mathsf{L}_m$ for the body of a rule. For brevity reasons, we will often denote a rule as $\mathtt{p}\ \overline{\mathsf{R}} \leftarrow \mathsf{B}$, 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 literals.

Example 2.1.
The following predicates illustrate the syntax of $\mathcal{HOL}$. Their precise semantics, particularly the distinction between `id` and `neg_neg`, will be clarified in the next section.

$$
\begin{aligned}
&\mathtt{id(R)} && \leftarrow \mathtt{R}\\
&\mathtt{neg(R)} && \leftarrow \sim\mathtt{R}\\
&\mathtt{neg_neg(R)} && \leftarrow \mathtt{neg(neg(R))}\\
&\mathtt{eq\,R\,Q} && \leftarrow \mathtt{neg(R)}, \mathtt{neg(Q)}\\
&\mathtt{eq\,R\,Q} && \leftarrow \mathtt{neg_neg(R)}, \mathtt{neg_neg(Q)}
\end{aligned}
$$

The predicate constants `id`, `neg`, and `neg_neg` are of type $o \to o$, while `eq` is of type $o \to (o \to o)$. The constant `id` denotes the identity relation, while `neg` lifts the negation operator $\sim$ to a predicate constant; this definition allows us to compose negations in rule bodies, as seen in the definition of `neg_neg`; this achieves the effect of double negation (i.e., $\sim\sim\mathtt{R}$), which is not directly supported as a syntactic form in $\mathcal{HOL}$. Finally, `eq` defines an equality relation on truth values, whose semantics will be explained in Section 3.

3 Equilibrium semantics for $\mathcal{HOL}$ programs

In this section we introduce the semantics of $\mathcal{HOL}$ programs, which is based on an extension of the equilibrium semantics of propositional programs introduced by Pearce (1996, 1999). Pearce's equilibrium semantics is built on top of Heyting's logic of HT; the semantics of HT is usually presented either through Kripke frames or through 3-valued

https://doi.org/10.1017/S147106842610057X Published online by Cambridge University Press

truth tables. We find it more convenient to use the latter approach, however we believe that our definitions and proofs can be adapted to the former. The underlying set of truth values of HT is $V = \{\mathbf{f}, \mathbf{t}^*, \mathbf{t}\}$. While $\mathbf{f}$ and $\mathbf{t}$ correspond to false and true, the value $\mathbf{t}^*$ is an intermediate one between $\mathbf{f}$ and $\mathbf{t}$, whose intuitive reading is "*weakly true because it lacks sufficient justification to be considered true*". The standard ordering in V is $\mathbf{f} < \mathbf{t}^* < \mathbf{t}$. Consider now the following ordering over V, which we represent by $\preceq$ and call the *justification ordering*:

Definition 3.1.
For all $v_1, v_2 \in \{\mathbf{f}, \mathbf{t}^*, \mathbf{t}\}$, $v_1 \preceq v_2$ iff $v_1 = v_2$ or $v_1 = \mathbf{t}^*$ and $v_2 = \mathbf{t}$.

The justification ordering is not new: it is the standard ordering used by D. Pearce over models in order to define the equilibrium ones. We use this ordering here to restrict our semantic domains to contain relations that preserve this ordering, namely the *justification monotonic* ones. The intuitive reason behind this requirement is that if a (higher-order) relation yields a $\mathbf{t}^*$ value, it must maintain or increase that value when the justification of its input increases. In other words, we don't want stronger evidence to invalidate a previous (even weakly true) conclusion. Later in this section, we provide a simple example which additionally motivates our adoption of justification monotonicity.

We proceed with the formal semantics of $\mathcal{HOL}$, starting with the semantics of its types. The semantics of the base Boolean domain is the set $V = \{\mathbf{f}, \mathbf{t}^*, \mathbf{t}\}$. The semantics of types of the form $\pi_1 \to \pi_2$ is the set of *justification-monotonic* functions from the domain of type π_1 to that of type π_2. We define, simultaneously with the meaning of every type π, two partial orders on the elements of type π: the relation $\leq_\pi$ which represents the *truth* ordering, and the relation $\preceq_\pi$ which represents the *justification* ordering.

Definition 3.2.
Let P be a $\mathcal{HOL}$ program. For every type π we define its meaning $[\![\pi]\!]$, as follows:

- $[\![o]\!] = \{\mathbf{f}, \mathbf{t}^*, \mathbf{t}\}$. The partial order $\leq_o$ is the usual one induced by the ordering $\mathbf{f} <_o \mathbf{t}^* <_o \mathbf{t}$; the partial order $\preceq_o$ is the one induced by the ordering $\mathbf{t}^* \prec_o \mathbf{t}$.
- $[\![\pi_1 \to \pi_2]\!] = [[\![\pi_1]\!] \to [\![\pi_2]\!]]$, namely the $\preceq$-monotonic functions[1] from $[\![\pi_1]\!]$ to $[\![\pi_2]\!]$. The partial order $\leq_{\pi_1 \to \pi_2}$ is defined as follows: for all $f, g \in [\![\pi_1 \to \pi_2]\!]$, $f \leq_{\pi_1 \to \pi_2} g$ iff $f(d) \leq_{\pi_2} g(d)$ for all $d \in [\![\pi_1]\!]$. The partial order $\preceq_{\pi_1 \to \pi_2}$ is defined as follows: for all $f, g \in [\![\pi_1 \to \pi_2]\!]$, $f \preceq_{\pi_1 \to \pi_2} g$ iff $f(d) \preceq_{\pi_2} g(d)$ for all $d \in [\![\pi_1]\!]$.

We will omit the subscripts of the above orders when they are obvious from context.

Definition 3.3.
An *interpretation* $\mathcal{I}$ assigns to each predicate constant $\mathsf{p} : \pi$ of $\mathcal{HOL}$, an element $\mathcal{I}(\mathsf{p}) \in [\![\pi]\!]$.

We will denote the set of interpretations with $\mathcal{H}$. We define two partial orders on $\mathcal{H}$ as follows: for all $\mathcal{I}, \mathcal{J} \in \mathcal{H}$, $\mathcal{I} \leq \mathcal{J}$ (respectively, $\mathcal{I} \preceq \mathcal{J}$) iff for every predicate constant p, $\mathcal{I}(\mathsf{p}) \leq_\pi \mathcal{J}(\mathsf{p})$ (respectively, $\mathcal{I}(\mathsf{p}) \preceq_\pi \mathcal{J}(\mathsf{p})$).

Definition 3.4.
A *state* s is a function that assigns to each predicate variable $\mathsf{R} : \pi$, an element $s(\mathsf{R}) \in [\![\pi]\!]$.

[1] Function $f \in [\![\pi_1 \to \pi_2]\!]$ is $\preceq$-monotonic if for all $d_1, d_2 \in [\![\pi_1]\!]$, $d_1 \preceq_{\pi_1} d_2$ implies $f(d_1) \preceq_{\pi_2} f(d_2)$.

https://doi.org/10.1017/S147106842610057X Published online by Cambridge University Press

We denote the set of states with $\mathcal{S}$. The partial orders $\leq$ and $\preceq$ extend to $\mathcal{S}$ in the obvious way. We will often use $s[\mathsf{R}_1/d_1, \ldots, \mathsf{R}_n/d_n]$ 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 $s[\overline{\mathsf{R}}/\overline{d}]$.

By abuse of language, we will often talk about "an interpretation (respectively, state) of a given program P" instead of "an interpretation (respectively, state) of $\mathcal{HOL}$".

Definition 3.5.
Let $\mathcal{I}$ be an interpretation and s a state. The semantics of expressions and literals with respect to $\mathcal{I}$ and s is defined as follows:

1. $[\![\mathsf{R}]\!]_s(\mathcal{I}) = s(\mathsf{R})$
2. $[\![\mathsf{p}]\!]_s(\mathcal{I}) = \mathcal{I}(\mathsf{p})$
3. $[\![(\mathsf{E}_1\ \mathsf{E}_2)]\!]_s(\mathcal{I}) = [\![\mathsf{E}_1]\!]_s(\mathcal{I})([\![\mathsf{E}_2]\!]_s(\mathcal{I}))$
4. $[\![(\sim\mathsf{E})]\!]_s(\mathcal{I}) = \sim[\![\mathsf{E}]\!]_s(\mathcal{I})$, *where* $\sim\mathbf{t} = \mathbf{f}, \sim\mathbf{f} = \mathbf{t}$, *and* $\sim\mathbf{t}^* = \mathbf{f}$.

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

Definition 3.6.
Let P be a program and $\mathcal{M}$ be an interpretation of P. Then, $\mathcal{M}$ is a *model* of P iff for every rule $\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n \leftarrow \mathsf{L}_1, \ldots, \mathsf{L}_m$ in P and for every state $s \in \mathcal{S}$, it holds that $\min_{\leq}\{[\![\mathsf{L}_i]\!]_s(\mathcal{M}) \mid i \in \{1, \ldots, m\}\} \leq [\![\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n]\!]_s(\mathcal{M})$.

We will be particularly interested in semantic elements that are essentially 2-valued:

Definition 3.7.
Let π be a type. An element $e \in [\![\pi]\!]$ is called *total* iff:

1. $\pi = o$ *and* $e \in \{\mathbf{f}, \mathbf{t}\}$, *or*
2. $\pi = \pi_1 \to \pi_2$ *and* $e(e')$ *is total for all total* $e' \in [\![\pi_1]\!]$.

An interpretation $\mathcal{I}$ *will be called* total *iff* $\mathcal{I}(\mathsf{p})$ *is total for every predicate constant* p.

The semantics of a $\mathcal{HOL}$ program will be captured by its set of *equilibrium* models:

Definition 3.8.
Let P be a $\mathcal{HOL}$ program and $\mathcal{M}$ be an interpretation of P. Then, $\mathcal{M}$ is called an *equilibrium model* of P iff $\mathcal{M}$ is total and a $\preceq$-minimal model of P.

Therefore, the meaning of a $\mathcal{HOL}$ program is captured by its set of equilibrium models. This semantics generalizes the classical equilibrium model semantics of Pearce for propositional programs (Pearce 1996, 1999) (because it uses the same logical machinery at the propositional level). In the rest of this section we give some examples of the proposed semantics, illustrate the necessity of restricting attention to $\preceq$-monotonic relations, and state two important properties of the semantics.

Example 3.1.
Based on the above semantics, it is not hard to check that the program consisting of the definitions in Example 2.1, has a unique equilibrium model which assigns to `id` the relation $\{(\mathbf{f}, \mathbf{f}), (\mathbf{t}^*, \mathbf{t}^*), (\mathbf{t}, \mathbf{t})\}$, to `neg` the relation $\{(\mathbf{f}, \mathbf{t}), (\mathbf{t}^*, \mathbf{f}), (\mathbf{t}, \mathbf{f})\}$, and to `neg_neg` the relation $\{(\mathbf{f}, \mathbf{f}), (\mathbf{t}^*, \mathbf{t}), (\mathbf{t}, \mathbf{t})\}$. Notice the difference between `id` and `neg_neg` which

https://doi.org/10.1017/S147106842610057X Published online by Cambridge University Press

demonstrates the fact that in HT double negation is different from the identity relation. Finally, the meaning of `eq` is a relation that returns $\mathbf{t}$ either if its two arguments are the same or one of them is $\mathbf{t}^*$ and the other $\mathbf{t}$; otherwise, it returns $\mathbf{f}$. It is easy to verify that all the aforementioned relations are $\preceq$-monotonic. On the other hand, the standard equality relation (which returns $\mathbf{t}$ only when its two arguments are the same and $\mathbf{f}$ otherwise), is not $\preceq$-monotonic and therefore not definable as a $\mathcal{HOL}$ program. In other words, $\mathcal{HOL}$ programs have no way to distinguish between $\mathbf{t}^*$ and $\mathbf{t}$.

Example 3.2.
Consider the following program:

$$\begin{aligned} \texttt{neg(R)} &\leftarrow \sim\texttt{R} \\ \texttt{p(R)} &\leftarrow \texttt{neg(neg(p(R)))} \end{aligned}$$

It can be verified that the above program has 4 equilibrium models, namely all the $\preceq$-monotonic relations of type $o \to o$ that are "two-valued" (i.e., they always return a value in $\{\mathbf{f}, \mathbf{t}\}$). More specifically, the four equilibrium models assign to `p` the following relations: $\mathcal{M}_1(\texttt{p}) = \{(\mathbf{f},\mathbf{f}), (\mathbf{t}^*,\mathbf{f}), (\mathbf{t},\mathbf{f})\}$, $\mathcal{M}_2(\texttt{p}) = \{(\mathbf{f},\mathbf{t}), (\mathbf{t}^*,\mathbf{f}), (\mathbf{t},\mathbf{f})\}$, $\mathcal{M}_3(\texttt{p}) = \{(\mathbf{f},\mathbf{f}), (\mathbf{t}^*,\mathbf{t}), (\mathbf{t},\mathbf{t})\}$, and $\mathcal{M}_4(\texttt{p}) = \{(\mathbf{f},\mathbf{t}), (\mathbf{t}^*,\mathbf{t}), (\mathbf{t},\mathbf{t})\}$.

The previous example demonstrates the production of all "two-valued" $\preceq$-monotonic relations of type $o \to o$. Can we produce *all the total* $\preceq$-monotonic ones? This is a more difficult task, requiring a more complicated program (see the supplementary material).

We now demonstrate, from a different angle, the importance of justification monotonicity in our semantics. The following simple example illustrates the necessity of the concept.

Example 3.3.
Consider the following program P, defining the predicate `p` of type $o \to o$:

$$\texttt{p(R)} \leftarrow \texttt{fail}$$

Intuitively, we expect that the meaning of `p` is the relation $\mathcal{M}(\texttt{p}) = \{(\mathbf{f},\mathbf{f}), (\mathbf{t}^*,\mathbf{f}), (\mathbf{t},\mathbf{f})\}$ (ie., the relation that maps every truth value to $\mathbf{f}$). However, if our semantics allowed arbitrary relations to be the meanings of predicates, the above program would have another equilibrium model, namely $\mathcal{N}(\texttt{p}) = \{(\mathbf{f},\mathbf{f}), (\mathbf{t}^*,\mathbf{t}^*), (\mathbf{t},\mathbf{f})\}$. Obviously, $\mathcal{N}(\texttt{p})$ is not monotonic, since $\mathbf{t}^* \preceq \mathbf{t}$ but $\mathcal{N}(\texttt{p})(\mathbf{t}^*) = \mathbf{t}^* \not\preceq \mathbf{f} = \mathcal{N}(\texttt{p})(\mathbf{t})$. However, $\mathcal{N}$ is total and $\preceq$-minimal model of P, so we would have to accept it as an equilibrium model of P if we had not restricted our domains to contain only the justification monotonic relations.

One could argue that accepting both $\mathcal{M}$ and $\mathcal{N}$ as equilibrium models, would be strange but not catastrophic; after all, the meaning of `p` is identical in both models if we restrict attention to total inputs. However, assume we add two more rules to the program:

$$\begin{aligned} \texttt{result} &\leftarrow \texttt{neg(neg(p(Q)))} \\ \texttt{p(R)} &\leftarrow \texttt{fail} \\ \texttt{neg(R)} &\leftarrow \sim\texttt{R} \end{aligned}$$

The definition of `result` uses in its body the existential variable `Q` (so, intuitively, the body reads "there exists a variable `Q` such that ..."). It is not hard to see that the

https://doi.org/10.1017/S147106842610057X Published online by Cambridge University Press

above program has two equilibrium models $\mathcal{M}$ and $\mathcal{N}$ such that $\mathcal{M}(\texttt{result}) = \mathbf{f}$ and $\mathcal{N}(\texttt{result}) = \mathbf{t}$. Obviously, now $\mathcal{N}$ produces a rather counterintuitive result.

We now establish two key properties of the proposed equilibrium semantics. Beyond their technical utility in proving subsequent results, these properties demonstrate that our semantics adheres to fundamental principles of logic programming. The first is a splitting lemma, analogous to the one introduced in (Lifschitz and Turner 1994). The second is a guarantee that the proposed semantics assigns a unique equilibrium model to every *stratified* $\mathcal{HOL}$ program. The proofs of the results are given in the supplementary material.

Definition 3.9.
Let P be a program and U a set of predicate constants. We say that U is a splitting set of P if for every rule C of P, if a predicate constant of U appears in the head of C, then every predicate constant appearing in C is included in U. The set of rules $C \in \mathsf{P}$ such that all predicate constants appearing in C are included in U is called the bottom of P relative to U and denoted by $b_U(\mathsf{P})$. The set $\mathsf{P} \setminus b_U(\mathsf{P})$ is called the top of P relative to U.

Lemma 3.1.
Let P be a program and U be a splitting set of P. If $\mathcal{M}$ is an equilibrium model of P, then $\mathcal{M}$ restricted to U is an equilibrium model of $b_U(\mathsf{P})$.

Stratified higher-order logic programs (Bogaerts *et al.* 2024) is a broad class of programs for which, as we show, the equilibrium semantics always produces a unique model.

Definition 3.10.
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.

Theorem 3.1.
Let P be a stratified $\mathcal{HOL}$ program. Then, P has a unique equilibrium model.

Notice that Examples 2.1 and 3.3 are stratified programs and, obviously, have a unique equilibrium model.

4 Two definability theorems for $\mathcal{HOL}$

In this section we demonstrate that every total $\preceq$-monotonic relation is definable by a stratified $\mathcal{HOL}$ program. Additionally, we argue that every (even non-total) $\preceq$-monotonic relation can be "captured" by a stratified $\mathcal{HOL}$ program (in a sense that will become clear later in the section). These definability results are important for two main reasons:

https://doi.org/10.1017/S147106842610057X Published online by Cambridge University Press

- Both results play an important technical role in establishing the strong equivalence theorem (Theorem 5.1 in the next section). The significance of the results in the proof, will be intuitively explained in the next section.
- Such definability results have an additional foundational significance: they demonstrate that $\mathcal{HOL}$ is strong enough to express all the elements of its semantic domain. To use Robin Milner's words (Milner 1977), such results suggest that the proposed semantics is not "over-generous". According to Milner, a semantics is over-generous if there exist elements in the semantic domain of a language that are not definable by phrases (ie., syntactic elements) of the language. It is interesting to note that Milner's remarks about definability, were given in the context of discussing *full-abstraction*, a notion conceptually close to strong equivalence.

We now formally state the two definability results and illustrate them by corresponding examples. The proofs of the theorems are given in the supplementary material. The two examples have been obtained by simplifying the programs produced by the constructions in the proofs of the theorems.

Theorem 4.1.
For each type π and every total $d \in [\![\pi]\!]$, there exists a stratified program P_d with a constant $\mathsf{c}_d : \pi$ such that P_d has a unique equilibrium model $\mathcal{M}$ with $\mathcal{M}(\mathsf{c}_d) = d$.

Example 4.1.
Let $d \in [\![o \to o]\!]$ be a total relation such that $d = \{(\mathbf{f}, \mathbf{t}), (\mathbf{t}^*, \mathbf{t}^*), (\mathbf{t}, \mathbf{t})\}$. We can create a program P_d as follows:

$$\begin{aligned} \mathtt{neg}(\mathtt{R}) &\leftarrow \sim\mathtt{R} \\ \mathtt{p_d}(\mathtt{R}) &\leftarrow \mathtt{neg}(\mathtt{R}) \\ \mathtt{p_d}(\mathtt{R}) &\leftarrow \mathtt{R} \end{aligned}$$

P_d is stratified and its unique equilibrium model $\mathcal{M}$ assigns to $\mathtt{p}_d$ the relation d.

Consider now the case of *non-total* $\preceq$-monotonic relations. Since our semantics is based on total relations (ie., every equilibrium model assigns to predicate constants total relations), a non-total relation is not directly definable in $\mathcal{HOL}$. However, we can "capture" non-total relations in the following sense: for every non-total $\preceq$-monotonic relation d we can create a stratified $\mathcal{HOL}$ program P_d which defines a predicate constant c^*_d; the unique equilibrium model $\mathcal{M}$ of P_d satisfies $\mathcal{M}(\mathsf{c}^*_d)(\mathbf{t}^*) = d$. In order to give the full formal statement of this result, we need the following definition of a collapse function.

Definition 4.1.
Let π be a predicate type and let $d \in [\![\pi]\!]$. We define *collapse (d)* recursively as:

- If $\pi = o$, $\mathit{collapse}(d) = \mathbf{f}$ if $d = \mathbf{f}$, $\mathit{collapse}(d) = \mathbf{t}$ otherwise.
- If $\pi = \pi_1 \to \pi_2$, then $\mathit{collapse}(d)(d_1) = \mathit{collapse}(d(d_1))$ for any $d_1 \in [\![\pi_1]\!]$.

We can now state our second definability theorem:

Theorem 4.2.
For each type π and $d \in [\![\pi]\!]$, there exists a stratified program P_d with a constant $\mathsf{c}^*_d : o \to \pi$ such that P_d has a unique equilibrium model $\mathcal{M}$ with

https://doi.org/10.1017/S147106842610057X Published online by Cambridge University Press

$$\mathcal{M}(\mathsf{c}_d^*)(u) = \begin{cases} d, & \text{if } u = \mathbf{t}^* \\ \mathit{collapse}(d), & \text{if } u = \mathbf{t} \\ \lambda\overline{x}.\mathbf{f}, & \text{if } u = \mathbf{f} \end{cases}$$

Example 4.2.
Let $d \in [\![o \to o]\!]$ be a relation such that $d = \{(\mathbf{f}, \mathbf{t}), (\mathbf{t}^*, \mathbf{t}^*), (\mathbf{t}, \mathbf{t}^*)\}$. We can create a program P_d as follows:

$$\begin{aligned} &\mathtt{neg(R)} && \leftarrow \sim\mathtt{R} \\ &\mathtt{neg_neg(R)} && \leftarrow \mathtt{neg(neg(R))} \\ &\mathtt{p}_d^*\,\mathtt{U}\,\mathtt{R} && \leftarrow \mathtt{neg_neg(U)}, \mathtt{neg(R)} \\ &\mathtt{p}_d^*\,\mathtt{U}\,\mathtt{R} && \leftarrow \mathtt{U}, \mathtt{neg_neg(R)} \\ &\mathtt{p}_d^*\,\mathtt{U}\,\mathtt{R} && \leftarrow \mathtt{U}, \mathtt{R} \end{aligned}$$

It is easy to see that P_d is stratified and its unique equilibrium model $\mathcal{M}$ satisfies the requirements of Theorem 4.2.

5 The strong equivalence theorem for higher-order logic programs

In this section we present the main theorem of the paper, which establishes that two programs P_1, P_2 are strongly equivalent iff they have the same models. Our result is a natural generalization, in programs of arbitrary orders, of the classical strong equivalence theorem for propositional programs (Lifschitz *et al.* 2001). We start by defining the notion of *strong equivalence* for $\mathcal{HOL}$ programs, which is a direct extension of the corresponding notion for propositional programs.

Definition 5.1.
Let $\mathsf{P}_1, \mathsf{P}_2$ be $\mathcal{HOL}$ programs. Then, P_1 and P_2 will be termed *strongly equivalent* iff for all $\mathcal{HOL}$ programs P, the programs $\mathsf{P}_1 \cup \mathsf{P}$ and $\mathsf{P}_2 \cup \mathsf{P}$ have the same equilibrium models.

Before presenting the theorem, we give two examples of programs that are not strongly equivalent. The first example involves two propositional programs and therefore we can use directly the classical characterization by Lifschitz *et al.* (2001). The second example is a related one, involving however a higher-order predicate. Both examples rely on the fact that double negation elimination is invalid in HT.

Example 5.1.
The following example illustrates that the "program" $\mathtt{p} \leftarrow \sim\sim\mathtt{r}$ is not strongly equivalent to the program $\mathtt{p} \leftarrow \mathtt{r}$; since our language does not allow nested negations of the form $\sim\sim\mathtt{r}$, we use an intermediate variable $\mathtt{q}$ to achieve the same effect. More specifically, the program P_1:

$$\begin{aligned} &\mathtt{p} \leftarrow \sim\mathtt{q} \\ &\mathtt{q} \leftarrow \sim\mathtt{r} \end{aligned}$$

and the program P_2:

$$\begin{aligned} &\mathtt{p} \leftarrow \mathtt{r} \\ &\mathtt{q} \leftarrow \sim\mathtt{r} \end{aligned}$$

https://doi.org/10.1017/S147106842610057X Published online by Cambridge University Press

are not strongly equivalent because they don't have the same HT models: the interpretation $\mathcal{I}(\mathtt{p}) = \mathbf{t}^*$, $\mathcal{I}(\mathtt{q}) = \mathbf{f}$, and $\mathcal{I}(\mathtt{r}) = \mathbf{t}^*$, is a model of P_2 but not of P_1. Actually, one can easily verify that by following the steps of the proof of Theorem 1 of Lifschitz et al. (2001), we can construct the program $\mathsf{P} = \{\mathtt{p} \leftarrow \mathtt{r}, \mathtt{r} \leftarrow \mathtt{p}\}$ such that $\mathsf{P}_1 \cup \mathsf{P}$ and $\mathsf{P}_2 \cup \mathsf{P}$ have different equilibrium models. In particular, $\mathsf{P}_1 \cup \mathsf{P}$ has the equilibrium model $\mathcal{K}(\mathtt{p}) = \mathbf{t}$, $\mathcal{K}(\mathtt{q}) = \mathbf{f}$, and $\mathcal{K}(\mathtt{r}) = \mathbf{t}$, which is not an equilibrium model of $\mathsf{P}_2 \cup \mathsf{P}$.

Example 5.2.
The following example illustrates that a program involving a predicate that is defined through double negation, is not strongly equivalent to one that does not use double negation. More specifically, the program P_1:

$$\mathtt{p}(\mathtt{R}) \leftarrow \mathtt{neg}(\mathtt{neg}(\mathtt{R}))$$
$$\mathtt{neg}(\mathtt{R}) \leftarrow {\sim}\mathtt{R}$$

and the program P_2:

$$\mathtt{p}(\mathtt{R}) \leftarrow \mathtt{R}$$
$$\mathtt{neg}(\mathtt{R}) \leftarrow {\sim}\mathtt{R}$$

are not strongly equivalent because they don't have the same higher-order $\mathcal{HOL}$ models. For example, the interpretation $\mathcal{I}$ such that $\mathcal{I}(\mathtt{p})(\mathbf{f}) = \mathbf{f}$, $\mathcal{I}(\mathtt{p})(\mathbf{t}^*) = \mathbf{t}^*$, $\mathcal{I}(\mathtt{p})(\mathbf{t}) = \mathbf{t}$, and $\mathcal{I}(\mathtt{neg})(\mathbf{f}) = \mathbf{t}$, $\mathcal{I}(\mathtt{neg})(\mathbf{t}^*) = \mathbf{f}$, $\mathcal{I}(\mathtt{neg})(\mathbf{t}) = \mathbf{f}$, is a model of P_2 but not of P_1. Actually, one can verify that by taking P to be the empty set, the programs $\mathsf{P}_1 \cup \mathsf{P} = \mathsf{P}_1$ and $\mathsf{P}_2 \cup \mathsf{P} = \mathsf{P}_2$ have different equilibrium models. In particular, $\mathcal{I}$ is an equilibrium model of P_2 but not of P_1.

The main result of the present paper (Theorem 5.1), namely that two $\mathcal{HOL}$ programs are strongly equivalent iff they have the same $\mathcal{HOL}$ models, is a nontrivial generalization of the corresponding result of Lifschitz *et al.* (2001) for propositional programs. In the rest of this section, we describe, at an intuitive level, the main technical difficulties that arise in the proof of Theorem 5.1 and how they are handled in our context. The full proof of the theorem is given in the supplementary material.

What makes the proof of Theorem 5.1 more challenging than the classical one is that in the heart of the proof lies the need to use the definability results of the previous section. To understand the need for definability, we have to first give an intuitive description of the central argument in the proof of the strong equivalence theorem for propositional programs (Lifschitz *et al.* 2001): assume that two (zero-order) programs P_1 and P_2 do not have the same models. For example, assume that there exists an interpretation $\mathcal{I}$ that is a model of P_1 but not of P_2. How can we find a program P such that $\mathsf{P}_1 \cup \mathsf{P}$ and $\mathsf{P}_2 \cup \mathsf{P}$ do not have the same equilibrium models? A crucial step in the proof of Lifschitz *et al.* (2001) is to include as facts in P all atoms A such that $\mathcal{I}(\mathsf{A}) = \mathbf{t}$. Turning to the proof for $\mathcal{HOL}$ programs, assume that for some predicate constant p in P and some elements $d_1, \ldots, d_n$ it holds that $\mathcal{I}(\mathsf{p})\, d_1 \cdots d_n = \mathbf{t}$. How could we create a "higher-order fact" in $\mathcal{HOL}$ which reflects that $\mathcal{I}(\mathsf{p})\, d_1 \cdots d_n = \mathbf{t}$?

Consider first the simpler case where $\mathcal{I}(\mathsf{p})$ *is a total relation*. In this case, we define $\mathcal{I}(\mathsf{p})$ as a predicate constant $\mathsf{c}_{\mathcal{I}(\mathsf{p})}$ (see Theorem 4.1) and then add to P the rule:

$$\mathsf{p}\, \mathsf{R}_1 \cdots \mathsf{R}_n \leftarrow \mathsf{c}_{\mathcal{I}(\mathsf{p})}\, \mathsf{R}_1 \cdots \mathsf{R}_n.$$

https://doi.org/10.1017/S147106842610057X Published online by Cambridge University Press

For the purposes of the proof of Theorem 5.1, the above rule behaves as a "higher-order fact": intuitively, given $d_1, \ldots, d_n$ such that $\mathcal{I}(\mathsf{p})\, d_1 \cdots d_n = \mathbf{t}$, the atom in the body of the above rule forces any interpretation $\mathcal{J}$ that satisfies the rule to also satisfy $\mathcal{J}(\mathsf{p})\, d_1 \cdots d_n = \mathbf{t}$.

Consider now the more involved case where $\mathcal{I}(\mathsf{p})$ *is not a total relation*. For this case the proof uses Theorem 4.2. More specifically, since $\mathcal{I}(\mathsf{p})$ is not total, there exist total elements $e_1, \ldots, e_n$ such that $\mathcal{I}(\mathsf{p})\, e_1 \cdots e_n = \mathbf{t}^*$. We denote by E_U the expression $(\mathsf{p}\, \mathsf{c}_{e_1} \cdots \mathsf{c}_{e_n})$. Then, we create the rule:

$$\mathsf{p}\, \mathsf{R}_1 \cdots \mathsf{R}_n \leftarrow \mathsf{c}^*_{\mathcal{I}(\mathsf{p})}\, \mathsf{E}_U\, \mathsf{R}_1 \cdots \mathsf{R}_n.$$

In the context of the proof of Theorem 5.1, this rule "behaves" as a fact. The formal statement of the theorem is given below and its proof can be found in the supplementary material.

Theorem 5.1.
For any programs P_1 and P_2 the following two statements are equivalent:

1. For every program P, $\mathsf{P}_1 \cup \mathsf{P}$ and $\mathsf{P}_2 \cup \mathsf{P}$ have the same equilibrium models.
2. P_1 and P_2 have the same models.

6 What is the "Correct" answer-set semantics for higher-order programs?

The study of answer-set semantics for higher-order logic programs was initiated by Bogaerts *et al.* (2024) using the framework of AFT (Denecker *et al.* 2000, 2004). A key advantage of AFT is its ability to unify various semantics - such as the well-founded, Kripke-Kleene, and stable model semantics - under a single constructive fixpoint framework. However, AFT also has a limitation: it characterizes these models purely iteratively rather than through a specific logical system. In contrast, the primary strength of the equilibrium semantics is that it offers a *logical characterization*, providing a framework that enables direct logical reasoning about programs.

The divergence between AFT and Equilibrium Logic in the propositional case was identified several years ago by Denecker *et al.* (2012). While the two techniques yield identical answer sets for normal propositional logic programs, they differ when rule bodies contain arbitrary propositional formulas. The precise relationship between the two is characterized by the following theorem by Denecker *et al.* (2012).

Theorem 6.1.
If no rule body of a propositional logic program P *contains nested negation, then* M *is a stable model of* P *in the sense of AFT iff* M *is an equilibrium model of* P.

Intuitively, the AFT-based semantics reduces double negation to identity, whereas the equilibrium semantics does not.

Example 6.1.
The program $\mathsf{p} \leftarrow \sim\sim\mathsf{p}$ has only the model $\{(\mathsf{p}, \mathbf{f})\}$ in AFT while it has two equilibrium models namely $\{(\mathsf{p}, \mathbf{f})\}$ and $\{(\mathsf{p}, \mathbf{t})\}$.

https://doi.org/10.1017/S147106842610057X Published online by Cambridge University Press

Unfortunately, the simple syntactic characterization of the differences between AFT and the equilibrium semantics, as implied by Theorem 6.1, does not straightforwardly generalize to $\mathcal{HOL}$ programs. In a higher-order context, the negation operator can effectively "disguise" itself, as illustrated by the following example.

Example 6.2.
The following program does not, at first sight, contain nested negation (the negation operator is hidden behind a different name).

$$\mathtt{p} \leftarrow \mathtt{neg}(\mathtt{neg}(\mathtt{p}))$$
$$\mathtt{neg}(\mathtt{R}) \leftarrow {\sim}\mathtt{R}$$

Again, this program has one stable model in AFT and two equilibrium models.

Because negation can be disguised in $\mathcal{HOL}$ in numerous ways (for instance, through a predicate that uses irrelevant code), syntactically characterizing the class of $\mathcal{HOL}$ programs where nested negation is used, appears to be a nontrivial, if not impossible, task. Crucially, double negation is not the sole source of conflict. One can define alternative predicates in $\mathcal{HOL}$ which, when used in a recursive context, also produce differing outcomes under AFT and equilibrium semantics. The following is such an example.

Example 6.3.
Consider the program:

$$\mathtt{q}(\mathtt{Q}) \leftarrow \mathtt{Q}$$
$$\mathtt{q}(\mathtt{Q}) \leftarrow {\sim}\mathtt{Q}$$
$$\mathtt{p} \leftarrow \mathtt{q}(\mathtt{p})$$

It can be verified that in AFT the program has a unique stable model in which $\mathtt{p}$ is $\mathbf{t}$; on the other hand, it does not have any equilibrium models.

These examples naturally raise the question: what is the "correct" answer-set semantics for higher-order logic programs? If one seeks generality and a constructive (fixpoint) approach for obtaining the semantics of a program, AFT is more suitable. However, we argue below that the equilibrium semantics offers a distinct advantage when the goal is to reason about programs and justify program transformations.

Example 6.4.
Consider again the program of Example 6.3. Assume we want to unfold the definition of $\mathtt{q}$ in the body of the definition of $\mathtt{p}$. This would give the new program:

$$\mathtt{q}(\mathtt{Q}) \leftarrow \mathtt{Q}$$
$$\mathtt{q}(\mathtt{Q}) \leftarrow {\sim}\mathtt{Q}$$
$$\mathtt{p} \leftarrow \mathtt{p}$$
$$\mathtt{p} \leftarrow {\sim}\mathtt{p}$$

Although the program seems, from a programmer's point of view, to be equivalent to that of Example 6.3, AFT now does not produce any stable model. On the other hand, the equilibrium semantics retains its behavior: the new program does not have any equilibrium models (as it was also the case for the initial one).

https://doi.org/10.1017/S147106842610057X Published online by Cambridge University Press

The above examples illustrate that mixing recursion and predicate application in unrestricted ways, results in discrepancies between the AFT and the equilibrium semantics. However, if recursion and applications are used in a controlled way, the two techniques converge. The proof of the following theorem is given in the supplementary material.

Theorem 6.2.
Let P *be a stratified program. Let* $\mathcal{M}$ *be its unique equilibrium model and* $\mathcal{N}$ *be its unique 2-valued stable model under the AFT semantics. Then,* $\pi(\mathcal{M}) = \mathcal{N}$*, where* π *is a collapse function from* $\mathcal{H}$ *to the set of 2-valued interpretations.*

We believe there exist much broader classes of programs than the stratified ones where the AFT-based and the equilibrium semantics coincide. One such potential fragment is the language "*Stratified+Choices Higher-Order Datalog*$^{\neg}$" defined in (Charalambidis *et al.* 2025), in which programs, intuitively, consist of a stratified part residing on top of a choice program. We have the following conjecture:

Conjecture 6.1.
Let P *be a program of Stratified+Choices Higher-Order Datalog*$^{\neg}$*. Then,* $\mathcal{M}$ *is an equilibrium model of* P *iff* $\pi(\mathcal{M})$ *is a 2-valued stable model of* P *under AFT, where* π *is a collapse function from* $\mathcal{H}$ *to the set of 2-valued interpretations.*

Beyond the aforementioned conjecture - which is significant for reconciling two prominent semantic frameworks - this work suggests several promising directions for future research. We briefly outline a few of these. A natural progression would be to extend equilibrium semantics to accommodate nested implications in rules. From a practical standpoint, it is also crucial to explore an extension of $\mathcal{HOL}$ that supports an individual data type ι (in addition to the Boolean type o), potentially building upon ideas found in Lifschitz *et al.* (2007); Pearce and Valverde (2008). Another interesting direction is the extension of temporal ASP (Becker *et al.* 2024) and stream reasoning (Beck *et al.* 2016) to the higher-order context.

Finally, future work could investigate the logical foundations of the well-founded semantics for $\mathcal{HOL}$ programs, utilizing a higher-order extension of the two-dimensional HT logic introduced by Cabalar (2001).

Supplementary material

To view supplementary material for this article, please visit https://doi.org/10.1017/S147106842610057X

References

Amendola, G., Ricca, F. and Truszczynski, M. 2019. Beyond NP: Quantifying over answer sets. *Theory and Practice of Logic Programming* 19, 705–721.

Beck, H., Dao-Tran, M. and Eiter, T. 2016. Equivalent stream reasoning programs. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI* 2016, New York, NY, USA, 9-15 July 2016, S. Kambhampati, Ed. IJCAI/AAAI Press, 929–935.

https://doi.org/10.1017/S147106842610057X Published online by Cambridge University Press

Becker, A., Cabalar, P., Diéguez, M., Schaub, T. and Schuhmann, A. 2024. Metric temporal equilibrium logic over timed traces. *Theory and Practice of Logic Programming* 24, 3, 425–452.

Bogaerts, B., Charalambidis, A., Chatziagapis, G., Kostopoulos, B., Pollaci, S. and Rondogiannis, P. 2024. The stable model semantics for higher-order logic programming. *Theory and Practice of Logic Programming* 24, 4, 737–754.

Bogaerts, B., Janhunen, T. and Tasharrofi, S. 2016. Stable-unstable semantics: Beyond NP with normal logic programs. *Theory and Practice of Logic Programming* 16, 570–586.

Cabalar, P. 2001. Well founded semantics as two dimensional here and there. In *Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st Intl. ASP'01 Workshop*, A. Provetti and T. C. Son, Eds. AAAI Press, Stanford, CA, USA, March 26-28, 2001.

Charalambidis, A., Kostopoulos, B., Nomikos, C. and Rondogiannis, P. 2025. The power of negation in higher-order datalog. *Theory and Practice of Logic Programming* 25, 4, 595–611.

Denecker, M., Bruynooghe, M. and Vennekens, J. 2012. *Approximation Fixpoint Theory and the Semantics of Logic and Answers Set Programs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 178–194.

Denecker, M., Marek, V. and Truszczyński, M. 2000. Approximations, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning. In *Logic-Based Artificial Intelligence 2000*, J. Minker, Eds. vol. 597 of The Springer International Series in Engineering and Computer Science, Springer US, 127–144.

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

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

Gunter, C. A. 1993. *Semantics of Programming Languages - Structures and Techniques*. Foundations of computing. MIT Press.

Lichtenstein, D. and Sipser, M. 1980. GO is polynomial-space hard. *Journal of the ACM* 27, 2, 393–401.

Lifschitz, V., Pearce, D. and Valverde, A. 2001. Strongly equivalent logic programs. *ACM Transactions on Computational Logic* 2, 4, 526–541.

Lifschitz, V., Pearce, D. and Valverde, A. 2007. *A Characterization of Strong Equivalence for Logic Programs with Variables*. Springer, Berlin Heidelberg, 188–200.

Lifschitz, V. and Turner, H. 1994. Splitting a logic program. In *Logic Programming: The 11th International Conference 1994*, The MIT Press.

Milner, R. 1977. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science* 4, 1, 1–22.

Pearce, D. 1996. A new logical characterisation of stable models and answer sets. In Non-Monotonic Extensions of Logic Programming, NMELP '96, Bad Honnef, Germany, September 5-6, 1996, Selected Papers 1996, J. Dix, L. M. Pereira and T. C. Przymusinski, Eds. vol. 1216 of Lecture Notes in Computer Science, Springer, 57–70.

Pearce, D. 1999. *From Here to There: Stable Negation in Logic Programming*. Springer Netherlands, 161–181.

Pearce, D. and Valverde, A. 2008. *Quantified Equilibrium Logic and Foundations for Answer Set Programs*. Springer, Berlin Heidelberg, 546–560.

Tennent, R. D. 1991. *Semantics of Programming Languages*. Prentice Hall International Series in Computer Science. Prentice Hall.

https://doi.org/10.1017/S147106842610057X Published online by Cambridge University Press