TLP **18** (3–4): 421–437, 2018. © Cambridge University Press 2018
doi:10.1017/S1471068418000108

Approximation Fixpoint Theory and the Well-Founded Semantics of Higher-Order Logic Programs

ANGELOS CHARALAMBIDIS, PANOS RONDOGIANNIS
and IOANNA SYMEONIDOU

Department of Informatics and Telecommunications, University of Athens, Greece
(*e-mail:* {acharala,prondo,sioanna}@di.uoa.gr)

submitted 22 April 2018; accepted 11 May 2018

Abstract

We define a novel, extensional, three-valued semantics for higher-order logic programs with negation. The new semantics is based on interpreting the types of the source language as three-valued Fitting-monotonic functions at all levels of the type hierarchy. We prove that there exists a bijection between such Fitting-monotonic functions and pairs of two-valued-result functions where the first member of the pair is monotone-antimonotone and the second member is antimonotone-monotone. By deriving an extension of *consistent approximation fixpoint theory* (Denecker *et al.* 2004) and utilizing the above bijection, we define an iterative procedure that produces for any given higher-order logic program a distinguished extensional model. We demonstrate that this model is actually a *minimal* one. Moreover, we prove that our construction generalizes the familiar well-founded semantics for classical logic programs, making in this way our proposal an appealing formulation for capturing the *well-founded semantics for higher-order logic programs*.

KEYWORDS: Higher-Order Logic Programming, Negation in Logic Programming, Approximation Fixpoint Theory.

1 Introduction

An intriguing and difficult question regarding logic programming, is whether it can be extended to a higher-order setting without sacrificing its semantic simplicity and clarity. Research results in this direction (Wadge 1991; Bezem 1999; Charalambidis *et al.* 2013; Rondogiannis and Symeonidou 2016; Rondogiannis and Symeonidou 2017) strongly suggest that it is possible to design higher-order logic programming languages that have powerful expressive capabilities, and which, at the same time, retain all the desirable semantic properties of classical first-order logic programming. In particular, it has been shown that higher-order logic programming can be given an *extensional semantics*, namely one in which program predicates denote sets. Under such a semantics one can use standard set-theoretic concepts in order to understand the meaning of programs and reason about them. For a more detailed discussion of extensionality and its importance for higher-order logic programming, the interested reader can consult the discussion in Section 2 of (Rondogiannis and Symeonidou 2017).

https://doi.org/10.1017/S1471068418000108 Published online by Cambridge University Press

The above line of research started many years ago by W. W. Wadge (Wadge 1991) who considered *positive* higher-order logic programs (i.e., programs without negation in clause bodies). Wadge argued that if such a program obeys some simple and natural syntactic rules, then it has a unique *minimum* Herbrand model. It is well-known that the *minimum model property* is a cornerstone of the theory of first-order logic programming (van Emden and Kowalski 1976). In this respect, Wadge's result suggested that it might be possible to extend all the elegant theory of classical logic programming to the higher-order case. The results in (Wadge 1991) were obtained using standard techniques from denotational semantics involving *continuous* interpretations and Kleene's least fixpoint theorem. A few years after Wadge's initial result, M. Bezem came to similar conclusions (Bezem 1999) but from a different direction. In particular, Bezem demonstrated that by using a fixpoint construction on the ground instantiation of the source higher-order program, one can obtain a model of the original program that satisfies an extensionality condition defined in (Bezem 1999). Despite their different philosophies, Wadge's and Bezem's approaches have recently been shown (Charalambidis *et al.* 2017) to have close connections. Apart from the above results, recent work (Charalambidis *et al.* 2013) has also shown that we can define a sound and complete proof procedure for positive higher-order logic programs, which generalizes classical SLD-resolution. In other words, the central results for positive first-order logic programs, generalize to the higher-order case.

A natural question that arises is whether one can still obtain an extensional semantics if negation is added to programs. Surprisingly, this question proved harder to resolve. The first result in this direction was reported in (Charalambidis *et al.* 2014), where it was demonstrated that every higher-order logic program with negation has a minimum extensional Herbrand model constructed over a logic with an infinite number of truth values. This result was obtained using domain-theoretic techniques as-well-as an extension of Kleene's fixpoint theorem that applies to a class of functions that are potentially non-monotonic (Ésik and Rondogiannis 2015). More recently, it was shown in (Rondogiannis and Symeonidou 2016) that Bezem's technique for positive programs can also be extended to apply to higher-order logic programs with negation, provided that it is interpreted under the same infinite-valued logic used in (Charalambidis *et al.* 2014). The above results, although satisfactory from a mathematical point of view, left open an annoying natural question: "Is it possible to define a *three-valued* extensional semantics for higher-order logic programs with negation that generalizes the standard well-founded semantics for classical logic programs?".

The above question was recently undertaken in (Rondogiannis and Symeonidou 2017). The surprising result was obtained that if Bezem's approach is interpreted under a three-valued logic, then the resulting semantics *can not be extensional* in the general case. One can see that similar arguments hold for the technique of (Charalambidis *et al.* 2014). Therefore, if we seek an extensional three-valued semantics for higher-order logic programs with negation, we need to follow an approach that is radically different from both (Charalambidis *et al.* 2014) and (Rondogiannis and Symeonidou 2017).

In this paper we undertake exactly the above problem. We demonstrate that we can indeed define a three-valued extensional semantics for higher-order logic programs with negation, which generalizes the familiar well-founded semantics of first-order logic programs (Gelder *et al.* 1991). Our results heavily utilize the technique of *approximation fixpoint theory* (Denecker *et al.* 2000; Denecker *et al.* 2004), which proved to be an

https://doi.org/10.1017/S1471068418000108 Published online by Cambridge University Press

indispensable tool in our investigation. The main contributions of the present paper can be outlined as follows:

- We define the first (to our knowledge) extensional three-valued semantics for higher-order logic programs with negation. Our semantics is based on interpreting the predicate types of our language as three-valued Fitting-monotonic functions (at all levels of the type hierarchy). We prove that there exists a bijection between such Fitting-monotonic functions and pairs of two-valued-result functions of the form (f_1, f_2), where f_1 is monotone-antimonotone, f_2 is antimonotone-monotone, and $f_1 \leq f_2$ (these notions will be explained in detail in Section 5).
- By deriving an extension of *consistent approximation fixpoint theory* (Denecker *et al.* 2004) and utilizing the above bijection, we define an iterative procedure that produces for any given higher-order logic program a distinguished extensional model. We prove that this model is actually a *minimal* one and we demonstrate that our construction generalizes the familiar well-founded semantics for classical logic programs. Therefore, we argue that our proposal is an appealing formulation for capturing the *well-founded semantics for higher-order logic programs*, paving in this way the road for a further study of negation in higher-order logic programming.

The rest of the paper is organized as follows. Section 2 presents in an intuitive way the main ideas developed in the paper. Section 3 introduces the syntax and Section 4 the semantics of our source language. Section 5 demonstrates the bijection between Fitting-monotonic functions and pairs of monotone-antimonotone and antimonotone-monotone functions. Section 6 develops the well-founded semantics of higher-order logic programs with negation, based on an extension of consistent approximation fixpoint theory. Section 7 compares the present work with that of (Charalambidis *et al.* 2014; Rondogiannis and Symeonidou 2017), and concludes by identifying some promising research directions. The proofs of most results of the paper are given in the supplementary material corresponding to this paper at the TPLP archives.

2 An Intuitive Overview of the Proposed Approach

In this section we describe in an intuitive way the main ideas and results obtained in the paper. As we have already mentioned, our goal is to derive a generalization of the well-founded semantics for higher-order logic programs with negation.

We start with our source language $\mathcal{HOL}$ which, intuitively speaking, allows *distinct* predicate variables (but not predicate constants) to appear in the heads of clauses. This is a syntactic restriction initially introduced in (Wadge 1991), which has been preserved and used by all subsequent articles in the area. As an example, consider the following program (for the moment we use ad-hoc Prolog-like syntax):

Example 1
The program below defines the `subset` relation over two unary predicates `P` and `Q`:

```
subset(P,Q) ← ∼ nonsubset(P,Q).
nonsubset(P,Q) ← P(X), ∼ Q(X).
```

Intuitively, `P` is a subset of `Q` if it is not the case that `P` is a non-subset of `Q`; and `P` is a non-subset of `Q` if there exists some `X` for which `P` is true while `Q` is false.

https://doi.org/10.1017/S1471068418000108 Published online by Cambridge University Press

The syntax we will introduce in Section 3 will allow a more compact notation using λ-expressions as the bodies of clauses (see Example 2 later in the paper).

We would like, for programs such as the above that are higher-order and use negation, to devise a three-valued extensional semantics. The key idea when assigning extensional semantics to *positive* higher-order logic programs (Wadge 1991; Charalambidis *et al.* 2013) is to interpret the predicate types of the language as monotonic and continuous functions. This is a well-known idea in the area of denotational semantics (Tennent 1991) and is a key assumption for obtaining the least fixpoint semantics for functional programs. This same idea was used in (Wadge 1991; Charalambidis *et al.* 2013) for obtaining the minimum Herbrand model semantics for positive higher-order logic programs. Unfortunately, this idea breaks down when we consider programs with negation: predicates defined using negation in clause bodies are not-necessarily monotonic. Non-monotonicity means that a higher-order predicate may be true of an input relation, but it may be false for a superset of this relation. For example, consider the predicate `p` below:

$$\texttt{p(Q)} \leftarrow \sim \texttt{Q(a).}$$

Obviously, `p` is true of the empty relation { } but it is false of the relation {`a`}. Notice that the notion of monotonicity we just discussed is usually called *monotonicity with respect to the (standard) truth ordering*.

Fortunately, there is another notion of monotonicity which is obeyed by higher-order logic programs with negation, namely *Fitting-monotonicity* (or *monotonicity with respect to the information ordering*) (Fitting 2002). Consider the program:

$$\begin{aligned}&\texttt{p(Q)} \leftarrow \sim \texttt{Q(a).}\\&\texttt{r(a)} \leftarrow \sim \texttt{r(a).}\\&\texttt{s(a).}\end{aligned}$$

Under the standard well-founded semantics for classical (first-order) logic programs, the truth value assigned to `r(a)` is *undefined*; on the other hand, `s(a)` is *true* in the same semantics. In other words, `r` corresponds to the 3-valued relation {(`a`, *undef*)} while `s` to the relation {(`a`, *true*)}. Fitting-monotonicity intuitively states that if a relation takes as argument a *more defined* relation, then it returns a more defined result. In our case this means that we expect the answer to the query `p(r)` to be less defined (alternatively, to *have less information*) than the answer to the query `p(s)` (more specifically, we expect `p(r)` to be *undefined* and `p(s)` to be *false*).

Based on the above discussion, we interpret the predicate types of our language as Fitting-monotonic functions. Then, an *interpretation* of a program is a function that assigns Fitting-monotonic functions to the predicates of the program. Given a program `P`, it is straightforward to define its *immediate consequence operator* $\Psi_{\texttt{P}}$, which, as usual, takes as input a Herbrand interpretation of the program and returns a new one. It is easy to prove that $\Psi_{\texttt{P}}$ is Fitting-monotonic. It is now tempting to assume that the least fixpoint of $\Psi_{\texttt{P}}$ with respect to the Fitting ordering, is the well-founded model that we are looking for. However, this is not the case: the least fixpoint of $\Psi_{\texttt{P}}$ is minimal with respect to the Fitting (i.e., information) ordering, while the well-founded model should be minimal with respect to the standard truth ordering. In order to get the correct model, we need a few more steps.

https://doi.org/10.1017/S1471068418000108 Published online by Cambridge University Press

We prove that there exists a bijection between Fitting-monotonic functions and pairs of functions of the form (f_1, f_2), where f_1 is monotone-antimonotone, f_2 is antimonotone-monotone, and $f_1 \leq f_2$ (where $\leq$ corresponds to the standard truth ordering). A similar bijection is established between three-valued interpretations and pairs of two-valued-result ones. This bijection allows us to use the powerful tool of approximation fixpoint theory (Denecker *et al.* 2000; Denecker *et al.* 2004). In particular, starting from a pair consisting of an underdefined interpretation and an overdefined one, and by iterating an appropriate operator, we demonstrate that we get to a pair of interpretations that is the limit of this sequence. Using our bijection, we show that this limit pair can be converted to a three-valued interpretation $\mathcal{M}_{\mathsf{P}}$ which is a three-valued model of our program P and actually a minimal one with respect to the standard truth ordering. We argue that this is the well-founded semantics of P, because its construction is a generalization of the construction in (Denecker *et al.* 2004) for the well-founded semantics of classical logic programs.

3 The Syntax of the Higher-Order Language $\mathcal{HOL}$

In this section we introduce $\mathcal{HOL}$, a higher-order language based on a simple type system that supports two base types: o, the boolean domain, and ι, the domain of individuals (data objects). The composite types are partitioned into three classes: *functional* (assigned to individual constants, individual variables and function symbols), *predicate* (assigned to predicate constants and variables) and *argument* (assigned to parameters of predicates).

Definition 1
A type can either be *functional*, *predicate*, or *argument*, denoted by σ, π and ρ respectively and defined as:

$$\sigma := \iota \mid \iota \to \sigma$$
$$\pi := o \mid \rho \to \pi$$
$$\rho := \iota \mid \pi$$

We will use τ to denote an arbitrary type (either functional, predicate or argument).

The binary operator $\to$ is right-associative. A functional type that is different from ι will often be written in the form $\iota^n \to \iota$, $n \geq 1$ (which stands for $\iota \to \iota \to \cdots \to \iota$ $(n+1)$-times). It can be easily seen that every predicate type π can be written uniquely in the form $\rho_1 \to \cdots \to \rho_n \to o$, $n \geq 0$ (for $n = 0$ we assume that $\pi = o$). We now define the alphabet, the expressions, and the program clauses of $\mathcal{HOL}$:

Definition 2
The *alphabet* of the higher-order language $\mathcal{HOL}$ consists of the following:

1. *Predicate variables* of every predicate type π (denoted by capital letters such as P and Q).
2. *Predicate constants* of every predicate type π (denoted by lowercase letters such as p and q).
3. *Individual variables* of type ι (denoted by capital letters such as X and Y).

https://doi.org/10.1017/S1471068418000108 Published online by Cambridge University Press

4. *Individual constants* of type ι (denoted by lowercase letters such as a and b).
5. *Function symbols* of every functional type $\sigma \neq \iota$ (denoted by lowercase letters such as f and g).
6. The following *logical constant symbols*: the constants false and true of type o; the equality constant $\approx$ of type $\iota \to \iota \to o$; the generalized disjunction and conjunction constants $\bigvee_\pi$ and $\bigwedge_\pi$ of type $\pi \to \pi \to \pi$, for every predicate type π; the generalized inverse implication constants $\leftarrow_\pi$ of type $\pi \to \pi \to o$, for every predicate type π; the existential quantifier $\exists_\rho$ of type $(\rho \to o) \to o$, for every argument type ρ; the negation constant $\sim$ of type $o \to o$.
7. The *abstractor* λ and the parentheses "(" and ")".

The set consisting of the predicate variables and the individual variables of $\mathcal{HOL}$ will be called the set of *argument variables* of $\mathcal{HOL}$. Argument variables will be denoted by R.

Definition 3
The set of *expressions* of the higher-order language $\mathcal{HOL}$ is defined as follows:

1. Every predicate variable (respectively, predicate constant) of type π is an expression of type π; every individual variable (respectively, individual constant) of type ι is an expression of type ι; the propositional constants false and true are expressions of type o.
2. If f is an n-ary function symbol and $\mathsf{E}_1, \ldots, \mathsf{E}_n$ are expressions of type ι, then $(\mathsf{f}\ \mathsf{E}_1 \cdots \mathsf{E}_n)$ is an expression of type ι.
3. If E_1 is an expression of type $\rho \to \pi$ and E_2 is an expression of type ρ, then $(\mathsf{E}_1\ \mathsf{E}_2)$ is an expression of type π.
4. If R is an argument variable of type ρ and E is an expression of type π, then $(\lambda \mathsf{R}.\mathsf{E})$ is an expression of type $\rho \to \pi$.
5. If $\mathsf{E}_1, \mathsf{E}_2$ are expressions of type π, then $(\mathsf{E}_1 \bigwedge_\pi \mathsf{E}_2)$ and $(\mathsf{E}_1 \bigvee_\pi \mathsf{E}_2)$ are expressions of type π.
6. If E is an expression of type o, then $(\sim\mathsf{E})$ is an expression of type o.
7. If $\mathsf{E}_1, \mathsf{E}_2$ are expressions of type ι, then $(\mathsf{E}_1 \approx \mathsf{E}_2)$ is an expression of type o.
8. If E is an expression of type o and R is a variable of type ρ then $(\exists_\rho \mathsf{R}\, \mathsf{E})$ is an expression of type o.

To denote that an expression E has type τ we will write $\mathsf{E} : \tau$. The notions of *free* and *bound* variables of an expression are defined as usual. An expression is called *closed* if it does not contain any free variables. An expression of type ι will be called a *term*; if it does not contain any individual variables, it will be called a *ground term*.

Definition 4
A *program clause* of $\mathcal{HOL}$ is of the form $\mathsf{p} \leftarrow_\pi \mathsf{E}$ where p is a predicate constant of type π and E is a closed expression of type π. A *program* is a finite set of program clauses.

Example 2
We rewrite the program of Example 1 using the syntax of $\mathcal{HOL}$. For every argument type ρ, the `subset` predicate of type $(\rho \to o) \to (\rho \to o) \to o$ takes as arguments two relations of type $\rho \to o$ and returns *true* if the first relation is a subset of the second:

$$\texttt{subset} \leftarrow_{(\rho\to o)\to(\rho\to o)\to o} \lambda\texttt{P}.\lambda\texttt{Q}.\sim\exists_\rho\texttt{X((P X)}\wedge\sim\texttt{(Q X))}$$

The use of λ-expressions obviates the need to have the formal parameters of the predicate in the left-hand side of the definition.

https://doi.org/10.1017/S1471068418000108 Published online by Cambridge University Press

4 The Semantics of the Higher-Order Language $\mathcal{HOL}$

In this section we begin the development of the semantics of the language $\mathcal{HOL}$. We start with the semantics of types, proceed with the semantics of expressions, and then with that of programs. We assume a familiarity with the basic notions of partially ordered sets (see Appendix A in the supplementary material corresponding to this paper at the TPLP archives for the main definitions).

The semantics of the base boolean domain is three-valued. The semantics of types of the form $\pi_1 \to \pi_2$ is the set of *Fitting-monotonic* functions from the domain of type π_1 to that of type π_2. We define, simultaneously with the meaning of every type τ, two partial orders on the elements of type τ: the relation $\leq_\tau$ which represents the *truth* ordering, and the relation $\preceq_\tau$ which represents the *information* or *Fitting* ordering.

Definition 5

Let D be a nonempty set. For every type τ we define recursively the set of possible meanings of elements of $\mathcal{HOL}$ of type τ, denoted by $[\![\tau]\!]_D$, as follows:

- $[\![o]\!]_D = \{\mathit{false}, \mathit{true}, \mathit{undef}\}$. The partial order $\leq_o$ is the usual 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 $\mathit{undef} \prec_o \mathit{true}$.
- $[\![\iota]\!]_D = D$. The partial order $\leq_\iota$ is defined as $d \leq_\iota d$ for all $d \in D$. The partial order $\preceq_\iota$ is also defined as $d \preceq_\iota d$ for all $d \in D$.
- $[\![\iota^n \to \iota]\!]_D = D^n \to D$. No ordering relations are defined for these types.
- $[\![\iota \to \pi]\!]_D = D \to [\![\pi]\!]_D$. The partial order $\leq_{\iota\to\pi}$ is defined as follows: for all $f, g \in [\![\iota \to \pi]\!]_D$, $f \leq_{\iota\to\pi} g$ iff $f(d) \leq_\pi g(d)$ for all $d \in D$. The partial order $\preceq_{\iota\to\pi}$ is defined as follows: for all $f, g \in [\![\iota \to \pi]\!]_D$, $f \preceq_{\iota\to\pi} g$ iff $f(d) \preceq_\pi g(d)$ for all $d \in D$.
- $[\![\pi_1 \to \pi_2]\!]_D = [[\![\pi_1]\!]_D \to [\![\pi_2]\!]_D]$, namely the $\preceq$-monotonic functions[1] from $[\![\pi_1]\!]_D$ to $[\![\pi_2]\!]_D$. The partial order $\leq_{\pi_1\to\pi_2}$ is defined as follows: for all $f, g \in [\![\pi_1 \to \pi_2]\!]_D$, $f \leq_{\pi_1\to\pi_2} g$ iff $f(d) \leq_{\pi_2} g(d)$ for all $d \in [\![\pi_1]\!]_D$. The partial order $\preceq_{\pi_1\to\pi_2}$ is defined as follows: for all $f, g \in [\![\pi_1 \to \pi_2]\!]_D$, $f \preceq_{\pi_1\to\pi_2} g$ iff $f(d) \preceq_{\pi_2} g(d)$ for all $d \in [\![\pi_1]\!]_D$.

The subscripts in the above partial orders will often be omitted when they are obvious from context. For every type π, the set $[\![\pi]\!]_D$ has a least element $\bot_{\leq_\pi}$ and a greatest element $\top_{\leq_\pi}$, called the *bottom* and the *top* elements of $[\![\pi]\!]_D$ with respect to $\leq_\pi$, respectively. In particular, $\bot_{\leq_o} = \mathit{false}$ and $\top_{\leq_o} = \mathit{true}$; $\bot_{\leq_{\iota\to\pi}}(d) = \bot_{\leq_\pi}$ and $\top_{\leq_{\iota\to\pi}}(d) = \top_{\leq_\pi}$, for all $d \in D$; $\bot_{\leq_{\pi_1\to\pi_2}}(d) = \bot_{\leq_{\pi_2}}$ and $\top_{\leq_{\pi_1\to\pi_2}}(d) = \top_{\leq_{\pi_2}}$, for all $d \in [\![\pi_1]\!]_D$. Moreover, for every type π, the set $[\![\pi]\!]_D$ has a least element with respect to $\preceq_\pi$, denoted by $\bot_{\preceq_\pi}$ and called the *bottom* element of $[\![\pi]\!]_D$ with respect to $\preceq_\pi$. In particular, $\bot_{\preceq_o} = \mathit{undef}$. The element $\bot_{\preceq_\pi}$ for $\pi \neq o$ can be defined in the obvious way as above. We will simply write $\bot$ to denote the bottom element of any of the above partially ordered sets, when the ordering relation and the specific domain are obvious from context.

We have the following proposition, whose proof is given in Appendix A in the supplementary material:

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

https://doi.org/10.1017/S1471068418000108 Published online by Cambridge University Press

Proposition 1
Let D be a nonempty set. For every predicate type π, $(\llbracket\pi\rrbracket_D, \leq_\pi)$ is a complete lattice and $(\llbracket\pi\rrbracket_D, \preceq_\pi)$ is a chain complete poset.

We can now proceed to define the semantics of $\mathcal{HOL}$:

Definition 6
A (three-valued) interpretation $\mathcal{I}$ of $\mathcal{HOL}$ consists of:

1. a nonempty set D called the *domain* of $\mathcal{I}$;
2. an assignment to each individual constant symbol c, of an element $\mathcal{I}(\mathsf{c}) \in D$;
3. an assignment to each predicate constant $\mathsf{p} : \pi$, of an element $\mathcal{I}(\mathsf{p}) \in \llbracket\pi\rrbracket_D$;
4. an assignment to each function symbol $\mathsf{f} : \iota^n \to \iota$, of a function $\mathcal{I}(\mathsf{f}) \in D^n \to D$.

Definition 7
Let D be a nonempty set. A *state* s of $\mathcal{HOL}$ over D is a function that assigns to each argument variable R of type ρ of $\mathcal{HOL}$, an element $s(\mathsf{R}) \in \llbracket\rho\rrbracket_D$.

We define: $\mathit{true}^{-1} = \mathit{false}$, $\mathit{false}^{-1} = \mathit{true}$ and $\mathit{undef}^{-1} = \mathit{undef}$.

Definition 8
Let D be a nonempty set, let $\mathcal{I}$ be an interpretation over D, and let s be a state over D. The semantics of expressions of $\mathcal{HOL}$ with respect to $\mathcal{I}$ and s, is defined as follows:

1. $\llbracket\mathsf{false}\rrbracket_s(\mathcal{I}) = \mathit{false}$, and $\llbracket\mathsf{true}\rrbracket_s(\mathcal{I}) = \mathit{true}$
2. $\llbracket\mathsf{c}\rrbracket_s(\mathcal{I}) = \mathcal{I}(\mathsf{c})$, for every individual constant c
3. $\llbracket\mathsf{p}\rrbracket_s(\mathcal{I}) = \mathcal{I}(\mathsf{p})$, for every predicate constant p
4. $\llbracket\mathsf{R}\rrbracket_s(\mathcal{I}) = s(\mathsf{R})$, for every argument variable R
5. $\llbracket(\mathsf{f}\ \mathsf{E}_1 \cdots \mathsf{E}_n)\rrbracket_s(\mathcal{I}) = \mathcal{I}(\mathsf{f})\ \llbracket\mathsf{E}_1\rrbracket_s(\mathcal{I}) \cdots \llbracket\mathsf{E}_n\rrbracket_s(\mathcal{I})$, for every n-ary function symbol f
6. $\llbracket(\mathsf{E}_1\mathsf{E}_2)\rrbracket_s(\mathcal{I}) = \llbracket\mathsf{E}_1\rrbracket_s(\mathcal{I})(\llbracket\mathsf{E}_2\rrbracket_s(\mathcal{I}))$
7. $\llbracket(\lambda\mathsf{R}.\mathsf{E})\rrbracket_s(\mathcal{I}) = \lambda d.\llbracket\mathsf{E}\rrbracket_{s[\mathsf{R}/d]}(\mathcal{I})$, where if $\mathsf{R} : \rho$ then d ranges over $\llbracket\rho\rrbracket_D$
8. $\llbracket(\mathsf{E}_1 \bigvee_\pi \mathsf{E}_2)\rrbracket_s(\mathcal{I}) = \bigvee_{\leq_\pi}\{\llbracket\mathsf{E}_1\rrbracket_s(\mathcal{I}), \llbracket\mathsf{E}_2\rrbracket_s(\mathcal{I})\}$
9. $\llbracket(\mathsf{E}_1 \bigwedge_\pi \mathsf{E}_2)\rrbracket_s(\mathcal{I}) = \bigwedge_{\leq_\pi}\{\llbracket\mathsf{E}_1\rrbracket_s(\mathcal{I}), \llbracket\mathsf{E}_2\rrbracket_s(\mathcal{I})\}$
10. $\llbracket(\sim\mathsf{E})\rrbracket_s(\mathcal{I}) = (\llbracket\mathsf{E}\rrbracket_s(\mathcal{I}))^{-1}$
11. $\llbracket(\mathsf{E}_1 \approx \mathsf{E}_2)\rrbracket_s(\mathcal{I}) = \begin{cases} \mathit{true}, & \text{if } \llbracket\mathsf{E}_1\rrbracket_s(\mathcal{I}) = \llbracket\mathsf{E}_2\rrbracket_s(\mathcal{I}) \\ \mathit{false}, & \text{otherwise} \end{cases}$
12. $\llbracket(\exists_\rho\mathsf{R}\,\mathsf{E})\rrbracket_s(\mathcal{I}) = \bigvee_{\leq_o}\{\llbracket\mathsf{E}\rrbracket_{s[\mathsf{R}/d]}(\mathcal{I}) \mid d \in \llbracket\rho\rrbracket_D\}$

For closed expressions E we will often write $\llbracket\mathsf{E}\rrbracket(\mathcal{I})$ instead of $\llbracket\mathsf{E}\rrbracket_s(\mathcal{I})$ (since, in this case, the meaning of E is independent of s). The following lemma demonstrates that our semantic valuation function returns elements that belong to the appropriate domain (the proof of the lemma by structural induction on E, is easy and omitted).

Lemma 1
Let $\mathsf{E} : \rho$ be an expression and let D be a nonempty set. Moreover, let s be a state over D and let $\mathcal{I}$ be an interpretation over D. Then, $\llbracket\mathsf{E}\rrbracket_s(\mathcal{I}) \in \llbracket\rho\rrbracket_D$.

Finally, we define the notion of *model* for $\mathcal{HOL}$ programs:

Definition 9
Let P be a $\mathcal{HOL}$ program and let M be an interpretation of P. Then M will be called a *model* of P iff for all clauses $\mathsf{p} \leftarrow_\pi \mathsf{E}$ of P, it holds $\llbracket\mathsf{E}\rrbracket(M) \leq_\pi M(\mathsf{p})$.

https://doi.org/10.1017/S1471068418000108 Published online by Cambridge University Press

5 An Alternative View of Fitting-Monotonic Functions

In this section we demonstrate that every Fitting-monotonic function f can be equivalently represented as a pair of functions (f_1, f_2), where f_1 is monotone-antimonotone, f_2 is antimonotone-monotone and $f_1 \leq f_2$. Consider for example a function f of type $o \to o$, i.e., $f : \{true, false, undef\} \to \{true, false, undef\}$. One can view the truth values as pairs where *true* corresponds to $(true, true)$, *false* corresponds to $(false, false)$, and *undef* corresponds to $(false, true)$. Therefore, f can also equivalently be seen as a function f' that takes pairs and returns pairs. We can then "break" f' into two components f_1 and f_2 where f_1 returns the first element of the pair that f' returns while f_2 returns the second. The monotone-antimonotone and antimonotone-monotone requirements ensure that the pair (f_1, f_2) retains the property of Fitting-monotonicity of the original function f. These ideas can be generalized to arbitrary types. The formal details of this equivalence are described below. The following definitions will be used:

Definition 10
Let L_1, L_2 be sets and let $\leq$ be a partial order on $L_1 \cup L_2$. We define: $L_1 \otimes_{\leq} L_2 = \{(x, y) \in L_1 \times L_2 : x \leq y\}$.

We will omit the $\leq$ from $\otimes_{\leq}$ when it is obvious from context.

Definition 11
Let L_1, L_2 be sets and let $\leq$ be a partial order on $L_1 \cup L_2$. Also, let $(A, \leq_A)$ be a partially ordered set. A function $f : (L_1 \otimes L_2) \to A$ will be called *monotone-antimonotone* (respectively *antimonotone-monotone*) if for all $(x, y), (x', y') \in L_1 \otimes L_2$ with $x \leq x'$ and $y' \leq y$, it holds that $f(x, y) \leq_A f(x', y')$ (respectively $f(x', y') \leq_A f(x, y)$). We denote by $[(L_1 \otimes L_2) \overset{\mathsf{ma}}{\to} A]$ the set of functions that are monotone-antimonotone and by $[(L_1 \otimes L_2) \overset{\mathsf{am}}{\to} A]$ those that are antimonotone-monotone.

In order to establish the bijection between Fitting-monotonic functions and pairs of monotone-antimonotone and antimonotone-monotone functions, we reinterpret the predicate types of $\mathcal{HOL}$ in an alternative way.

Definition 12
Let D be a nonempty set. For every type τ we define the monotone-antimonotone and the antimonotone-monotone meanings of the elements of type τ with respect to D, denoted respectively by $[\![\tau]\!]^{\mathsf{ma}}_D$ and $[\![\tau]\!]^{\mathsf{am}}_D$. At the same time we define a partial order $\leq_\tau$ between the elements of $[\![\tau]\!]^{\mathsf{ma}}_D \cup [\![\tau]\!]^{\mathsf{am}}_D$.

- $[\![o]\!]^{\mathsf{ma}}_D = [\![o]\!]^{\mathsf{am}}_D = \{false, true\}$. The partial order $\leq_o$ is the usual one induced by the ordering $false \leq_o true$.
- $[\![\iota]\!]^{\mathsf{ma}} = [\![\iota]\!]^{\mathsf{am}} = D$. The partial order $\leq_\iota$ is defined as $d \leq_\iota d$, for all $d \in D$.
- $[\![\iota^n \to \iota]\!]^{\mathsf{ma}} = [\![\iota^n \to \iota]\!]^{\mathsf{am}} = D^n \to D$. There is no partial order for elements of type $\iota^n \to \iota$.
- $[\![\iota \to \pi]\!]^{\mathsf{ma}}_D = D \to [\![\pi]\!]^{\mathsf{ma}}_D$ and $[\![\iota \to \pi]\!]^{\mathsf{am}}_D = D \to [\![\pi]\!]^{\mathsf{am}}_D$. The partial order $\leq_{\iota \to \pi}$ is defined as follows: for all $f, g \in [\![\iota \to \pi]\!]^{\mathsf{ma}}_D \cup [\![\iota \to \pi]\!]^{\mathsf{am}}_D$, $f \leq_{\iota \to \pi} g$ iff $f(d) \leq_\pi g(d)$ for all $d \in D$.
- $[\![\pi_1 \to \pi_2]\!]^{\mathsf{ma}}_D = [([\![\pi_1]\!]^{\mathsf{ma}}_D \otimes [\![\pi_1]\!]^{\mathsf{am}}_D) \overset{\mathsf{ma}}{\to} [\![\pi_2]\!]^{\mathsf{ma}}_D]$, and $[\![\pi_1 \to \pi_2]\!]^{\mathsf{am}}_D = [([\![\pi_1]\!]^{\mathsf{ma}}_D \otimes [\![\pi_1]\!]^{\mathsf{am}}_D) \overset{\mathsf{am}}{\to} [\![\pi_2]\!]^{\mathsf{am}}_D]$. The relation $\leq_{\pi_1 \to \pi_2}$ is the partial order defined as follows: for all $f, g \in [\![\pi_1 \to \pi_2]\!]^{\mathsf{ma}}_D \cup [\![\pi_1 \to \pi_2]\!]^{\mathsf{am}}_D$, $f \leq_{\pi_1 \to \pi_2} g$ iff $f(d_1, d_2) \leq_{\pi_2} g(d_1, d_2)$ for all $(d_1, d_2) \in [\![\pi_1]\!]^{\mathsf{ma}}_D \otimes [\![\pi_1]\!]^{\mathsf{am}}_D$.

https://doi.org/10.1017/S1471068418000108 Published online by Cambridge University Press

For every π, the bottom and top elements of $[\![\pi]\!]_D^{\mathsf{ma}}$ and $[\![\pi]\!]_D^{\mathsf{am}}$ can be defined in the obvious way. We have the following proposition (see Appendix B in the supplementary material corresponding to this paper at the TPLP archives for the proof):

Proposition 2
Let D be a nonempty set. For every predicate type π, $([\![\pi]\!]_D^{\mathsf{ma}}, \leq_\pi)$ and $([\![\pi]\!]_D^{\mathsf{am}}, \leq_\pi)$ are complete lattices.

We extend, in a pointwise way, our orderings to apply to pairs. For simplicity, we overload our notation and use the same symbols $\leq$ and $\preceq$ for the new orderings.

Definition 13
Let D be a nonempty set and let π be a predicate type. We define the relations $\leq_\pi$ and $\preceq_\pi$, so that for all $(x, y), (x', y') \in [\![\pi]\!]_D^{\mathsf{ma}} \otimes [\![\pi]\!]_D^{\mathsf{am}}$:

- $(x, y) \leq_\pi (x', y')$ iff $x \leq_\pi x'$ and $y \leq_\pi y'$.
- $(x, y) \preceq_\pi (x', y')$ iff $x \leq_\pi x'$ and $y' \leq_\pi y$.

The following proposition is demonstrated in Appendix B in the supplementary material:

Proposition 3
Let D be a nonempty set. For each predicate type π, $[\![\pi]\!]_D^{\mathsf{ma}} \otimes [\![\pi]\!]_D^{\mathsf{am}}$ is a complete lattice with respect to $\leq_\pi$ and a chain-complete poset with respect to $\preceq_\pi$.

In the rest of the paper we will denote the *first* and *second* selection functions on pairs with the more compact notation $[\cdot]_1$ and $[\cdot]_2$: given any pair (x, y), it is $[(x, y)]_1 = x$ and $[(x, y)]_2 = y$. We can now establish the bijection between $[\![\pi]\!]_D$ and $[\![\pi]\!]_D^{\mathsf{ma}} \otimes [\![\pi]\!]_D^{\mathsf{am}}$. The following definition and two propositions (whose proofs are given in Appendix B in the supplementary material), explain how.

Definition 14
Let D be a nonempty set. For every predicate type π, we define recursively the functions $\tau_\pi : [\![\pi]\!]_D \to ([\![\pi]\!]_D^{\mathsf{ma}} \otimes [\![\pi]\!]_D^{\mathsf{am}})$ and $\tau_\pi^{-1} : ([\![\pi]\!]_D^{\mathsf{ma}} \otimes [\![\pi]\!]_D^{\mathsf{am}}) \to [\![\pi]\!]_D$, as follows.

- $\tau_o(\mathit{false}) = (\mathit{false}, \mathit{false})$, $\tau_o(\mathit{true}) = (\mathit{true}, \mathit{true})$, $\tau_o(\mathit{undef}) = (\mathit{false}, \mathit{true})$
- $\tau_{\iota\to\pi}(f) = (\lambda d.[\tau_\pi(f(d))]_1, \lambda d.[\tau_\pi(f(d))]_2)$
- $\tau_{\pi_1\to\pi_2}(f) = (\lambda(d_1, d_2).[\tau_{\pi_2}(f(\tau_{\pi_1}^{-1}(d_1, d_2)))]_1, \lambda(d_1, d_2).[\tau_{\pi_2}(f(\tau_{\pi_1}^{-1}(d_1, d_2)))]_2)$

and

- $\tau_o^{-1}(\mathit{false}, \mathit{false}) = \mathit{false}$, $\tau_o^{-1}(\mathit{true}, \mathit{true}) = \mathit{true}$, $\tau_o^{-1}(\mathit{false}, \mathit{true}) = \mathit{undef}$
- $\tau_{\iota\to\pi}^{-1}(f_1, f_2) = \lambda d.\tau_\pi^{-1}(f_1(d), f_2(d))$
- $\tau_{\pi_1\to\pi_2}^{-1}(f_1, f_2) = \lambda d.\tau_{\pi_2}^{-1}(f_1(\tau_{\pi_1}(d)), f_2(\tau_{\pi_1}(d)))$.

Proposition 4
Let D be a nonempty set and let π be a predicate type. Then, for every $f, g \in [\![\pi]\!]_D$ and for every $(f_1, f_2), (g_1, g_2) \in [\![\pi]\!]_D^{\mathsf{ma}} \otimes [\![\pi]\!]_D^{\mathsf{am}}$, the following statements hold:

1. $\tau_\pi(f) \in ([\![\pi]\!]_D^{\mathsf{ma}} \otimes [\![\pi]\!]_D^{\mathsf{am}})$ and $\tau_\pi^{-1}(f_1, f_2) \in [\![\pi]\!]_D$.
2. If $f \preceq_\pi g$ then $\tau_\pi(f) \preceq_\pi \tau_\pi(g)$.
3. If $f \leq_\pi g$ then $\tau_\pi(f) \leq_\pi \tau_\pi(g)$.
4. If $(f_1, f_2) \preceq_\pi (g_1, g_2)$ then $\tau_\pi^{-1}(f_1, f_2) \preceq_\pi \tau_\pi^{-1}(g_1, g_2)$.
5. If $(f_1, f_2) \leq_\pi (g_1, g_2)$ then $\tau_\pi^{-1}(f_1, f_2) \leq_\pi \tau_\pi^{-1}(g_1, g_2)$.

Proposition 5
Let D be a nonempty set and let π be a predicate type. Then, for every $f \in [\![\pi]\!]_D$, $\tau_\pi^{-1}(\tau_\pi(f)) = f$, and for every $(f_1, f_2) \in [\![\pi]\!]_D^{\mathsf{ma}} \otimes [\![\pi]\!]_D^{\mathsf{am}}$, $\tau_\pi(\tau_\pi^{-1}(f_1, f_2)) = (f_1, f_2)$.

https://doi.org/10.1017/S1471068418000108 Published online by Cambridge University Press

6 The Well-Founded Semantics for $\mathcal{HOL}$ Programs

In this section we demonstrate that every program of $\mathcal{HOL}$ has a distinguished *minimal Herbrand model* which can be obtained by an iterative procedure. This construction generalizes the familiar well-founded semantics. Our main results are based on a mild generalization of the consistent approximation fixpoint theory of (Denecker *et al.* 2004). We start with the relevant definitions.

Definition 15
Let P be a program. The Herbrand universe U_P of P is the set of all ground terms that can be formed out of the individual constants[2] and the function symbols of P.

Definition 16
A (three-valued) *Herbrand interpretation* $\mathcal{I}$ of a program P is an interpretation such that:

1. the domain of $\mathcal{I}$ is the Herbrand universe U_P of P;
2. for every individual constant c of P, $\mathcal{I}(\mathsf{c}) = \mathsf{c}$;
3. for every predicate constant $\mathsf{p} : \pi$ of P, $\mathcal{I}(\mathsf{p}) \in [\![\pi]\!]_{U_\mathsf{P}}$;
4. for every n-ary function symbol f of P and for all $\mathsf{t}_1, \ldots, \mathsf{t}_n \in U_\mathsf{P}$, $\mathcal{I}(\mathsf{f})\,\mathsf{t}_1 \cdots \mathsf{t}_n = \mathsf{f}\,\mathsf{t}_1 \cdots \mathsf{t}_n$.

We denote the set of all three-valued Herbrand interpretations of a program P by $\mathcal{H}_\mathsf{P}$. A *Herbrand state* of P is a state whose underlying domain is U_P. A *Herbrand model* of P is a Herbrand interpretation that is a model of P. The truth and the information orderings easily extend to Herbrand interpretations:

Definition 17
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, $\mathcal{I}(\mathsf{p}) \preceq_\pi \mathcal{J}(\mathsf{p})$).

The proof of the following proposition is analogous to that of Proposition 1 and omitted:

Proposition 6
Let P be a program. Then, $(\mathcal{H}_\mathsf{P}, \leq)$ is a complete lattice and $(\mathcal{H}_\mathsf{P}, \preceq)$ is a chain complete poset.

The following lemma is also easy to establish, and its proof is omitted:

Lemma 2
Let P be a program, let $\mathcal{I}, \mathcal{J} \in \mathcal{H}_\mathsf{P}$, and let s be a Herbrand state of P. For every expression E, if $\mathcal{I} \preceq \mathcal{J}$ then $[\![\mathsf{E}]\!]_s(\mathcal{I}) \preceq [\![\mathsf{E}]\!]_s(\mathcal{J})$.

The bijection established in Section 5 extends also to interpretations. More specifically, every three-valued Herbrand interpretation $\mathcal{I}$ of a program P can be mapped by (an extension of) τ to a pair of functions (I, J) such that:

- for every individual constant c of P, $I(\mathsf{c}) = J(\mathsf{c}) = \mathsf{c}$;
- for every predicate constant $\mathsf{p} : \pi$ of P, $I(\mathsf{p}) \in [\![\pi]\!]^{\mathsf{ma}}_{U_\mathsf{P}}$ and $J(\mathsf{p}) \in [\![\pi]\!]^{\mathsf{am}}_{U_\mathsf{P}}$;

[2] As usual, if P has no constants, we assume the existence of an arbitrary one.

https://doi.org/10.1017/S1471068418000108 Published online by Cambridge University Press

- for every n-ary function symbol f of P and for all $\mathsf{t}_1, \ldots, \mathsf{t}_n \in U_\mathsf{P}$, $I(\mathsf{f})\,\mathsf{t}_1 \cdots \mathsf{t}_n = J(\mathsf{f})\,\mathsf{t}_1 \cdots \mathsf{t}_n = \mathsf{f}\,\mathsf{t}_1 \cdots \mathsf{t}_n$.

Functions of the form I above will be called "*monotone-antimonotone Herbrand interpretations*" and functions of the form J will be called "*antimonotone-monotone Herbrand interpretations*". We will denote by $\mathcal{H}_\mathsf{P}^{\mathsf{ma}}$ the set of functions of the former type and by $\mathcal{H}_\mathsf{P}^{\mathsf{am}}$ those of the latter type. As in Definition 17, we can define a partial order $\leq$ on $\mathcal{H}_\mathsf{P}^{\mathsf{ma}} \cup \mathcal{H}_\mathsf{P}^{\mathsf{am}}$. Similarly, as in Definition 13, we can define partial orders $\leq$ and $\preceq$ on $\mathcal{H}_\mathsf{P}^{\mathsf{ma}} \otimes \mathcal{H}_\mathsf{P}^{\mathsf{am}}$. The proof of the following proposition is a direct consequence of the proofs of Propositions 3 and 2, and therefore omitted.

Proposition 7
Let P be a program. Then, $(\mathcal{H}_\mathsf{P}^{\mathsf{ma}}, \leq)$ and $(\mathcal{H}_\mathsf{P}^{\mathsf{am}}, \leq)$ are complete lattices having the same $\bot$ and $\top$ elements. Moreover, $(\mathcal{H}_\mathsf{P}^{\mathsf{ma}} \otimes \mathcal{H}_\mathsf{P}^{\mathsf{am}}, \leq)$ is a complete lattice and $(\mathcal{H}_\mathsf{P}^{\mathsf{ma}} \otimes \mathcal{H}_\mathsf{P}^{\mathsf{am}}, \preceq)$ is a chain-complete poset.

The bijection between $\mathcal{H}_\mathsf{P}$ and $\mathcal{H}_\mathsf{P}^{\mathsf{ma}} \otimes \mathcal{H}_\mathsf{P}^{\mathsf{am}}$ can be explained more formally as follows. Given $\mathcal{I} \in \mathcal{H}_\mathsf{P}$, we define $\tau(\mathcal{I}) = (I, J)$, where for every predicate constant $\mathsf{p} : \pi$ it holds $I(\mathsf{p}) = [\tau_\pi(\mathcal{I}(\mathsf{p}))]_1$ and $J(\mathsf{p}) = [\tau_\pi(\mathcal{I}(\mathsf{p}))]_2$. Conversely, given a pair $(I, J) \in \mathcal{H}_\mathsf{P}^{\mathsf{ma}} \otimes \mathcal{H}_\mathsf{P}^{\mathsf{am}}$, we define the three-valued Herbrand interpretation $\mathcal{I}$ as follows: $\mathcal{I}(\mathsf{p}) = \tau_\pi^{-1}(I(\mathsf{p}), J(\mathsf{p}))$. We now define the three-valued and two-valued immediate consequence operators:

Definition 18
Let P be a program. The *three-valued immediate consequence operator* $\Psi_\mathsf{P} : \mathcal{H}_\mathsf{P} \to \mathcal{H}_\mathsf{P}$ of P is defined for every $\mathsf{p} : \pi$ as: $\Psi_\mathsf{P}(\mathcal{I})(\mathsf{p}) = \bigvee_{\leq_\pi} \{[\![\mathsf{E}]\!](\mathcal{I}) \mid (\mathsf{p} \leftarrow_\pi \mathsf{E}) \in \mathsf{P}\}$.

Definition 19
Let P be a program. The *two-valued immediate consequence operator* $T_\mathsf{P} : (\mathcal{H}_\mathsf{P}^{\mathsf{ma}} \otimes \mathcal{H}_\mathsf{P}^{\mathsf{am}}) \to (\mathcal{H}_\mathsf{P}^{\mathsf{ma}} \otimes \mathcal{H}_\mathsf{P}^{\mathsf{am}})$ of P is defined as: $T_\mathsf{P}(I, J) = \tau(\Psi_\mathsf{P}(\tau^{-1}(I, J)))$.

From Proposition 14 in Appendix D (found in the supplementary material corresponding to this paper at the TPLP archives) it follows that T_P is well-defined. Moreover, it is Fitting-monotonic as the following lemma demonstrates (see Appendix D for the proof):

Lemma 3
Let P be a program and let $(I_1, J_1), (I_2, J_2) \in \mathcal{H}_\mathsf{P}^{\mathsf{ma}} \otimes \mathcal{H}_\mathsf{P}^{\mathsf{am}}$. If $(I_1, J_1) \preceq (I_2, J_2)$ then $T_\mathsf{P}(I_1, J_1) \preceq T_\mathsf{P}(I_2, J_2)$.

We will use T_P to construct the well-founded model of program P. Our construction is based on a mild extension of consistent approximation fixpoint theory (Denecker *et al.* 2004). Therefore, in order for the following two definitions and subsequent theorem to be fully comprehended, it would be helpful if the reader had some familiarity with the material in (Denecker *et al.* 2004).

Definition 20
Let P be a program and let $(I, J) \in \mathcal{H}_\mathsf{P}^{\mathsf{ma}} \otimes \mathcal{H}_\mathsf{P}^{\mathsf{am}}$. Assume that $(I, J) \preceq T_\mathsf{P}(I, J)$. We define $I^\uparrow = \mathit{lfp}([T_\mathsf{P}(I, \cdot)]_2)$ and $J^\downarrow = \mathit{lfp}([T_\mathsf{P}(\cdot, J)]_1)$, where by $T_\mathsf{P}(\cdot, J)$ we denote the function $f(x) = T_\mathsf{P}(x, J)$ and by $T_\mathsf{P}(I, \cdot)$ the function $g(x) = T_\mathsf{P}(I, x)$.

It can be shown (see Appendix C in the supplementary material) that $I^\uparrow$ and $J^\downarrow$ are well-defined, and this is due to the crucial assumption $(I, J) \preceq T_\mathsf{P}(I, J)$. This property

https://doi.org/10.1017/S1471068418000108 Published online by Cambridge University Press

was introduced in (Denecker *et al.* 2004) where it is named A-reliability (in our case A is the T_{P} operator). Before proceeding to the definition of the well-founded semantics, we need to define one more operator, namely the *stable revision operator* (see (Denecker *et al.* 2004)[page 91] for the intuition and motivation behind this operator).

Definition 21
Let P be a program. We define the function $\mathcal{C}_{T_{\mathsf{P}}}$ which for every pair $(I, J) \in \mathcal{H}_{\mathsf{P}}^{\mathsf{ma}} \otimes \mathcal{H}_{\mathsf{P}}^{\mathsf{am}}$ with $(I, J) \preceq T_{\mathsf{P}}(I, J)$, returns the pair $(J^{\downarrow}, I^{\uparrow})$:

$$\mathcal{C}_{T_{\mathsf{P}}}(I, J) = (J^{\downarrow}, I^{\uparrow}) = (lfp([T_{\mathsf{P}}(\cdot, J)]_1), lfp([T_{\mathsf{P}}(I, \cdot)]_2))$$

The function $\mathcal{C}_{T_{\mathsf{P}}}$ will be called the *stable revision operator* for T_{P}.

The following theorem is a direct consequence of Theorem 5 given in Appendix C in the supplementary material (which extends Theorem 3.11 in (Denecker *et al.* 2004) to our case):

Theorem 1
Let P be a program. We define the following sequence of pairs of interpretations:

$$\begin{array}{ll} (I_0, J_0) & = (\bot, \top) \\ (I_{\lambda+1}, J_{\lambda+1}) & = \mathcal{C}_{T_{\mathsf{P}}}(I_\lambda, J_\lambda) \\ (I_\lambda, J_\lambda) & = \bigvee_{\preceq}\{(I_\kappa, J_\kappa) \mid \kappa < \lambda\} \text{ for limit ordinals } \lambda \end{array}$$

Then, the above sequence of pairs of interpretations is well-defined. Moreover, there exists a least ordinal δ such that $(I_\delta, J_\delta) = \mathcal{C}_{T_{\mathsf{P}}}(I_\delta, J_\delta)$ and $(I_\delta, J_\delta) \in \mathcal{H}_{\mathsf{P}}^{\mathsf{ma}} \otimes \mathcal{H}_{\mathsf{P}}^{\mathsf{am}}$.

In the following, we will denote with $\mathcal{M}_{\mathsf{P}}$ the interpretation $\tau^{-1}(I_\delta, J_\delta)$. The following two lemmas demonstrate that the pre-fixpoints of T_{P} correspond exactly to the three-valued models of P (see Appendix D in the supplementary material for the corresponding proofs).

Lemma 4
Let P be a program. If $(I, J) \in \mathcal{H}_{\mathsf{P}}^{\mathsf{ma}} \otimes \mathcal{H}_{\mathsf{P}}^{\mathsf{am}}$ is a pre-fixpoint of T_{P} then $\tau^{-1}(I, J)$ is a model of P.

Lemma 5
Let $\mathcal{M} \in \mathcal{H}_{\mathsf{P}}$ be a model of P. Then, $\tau(\mathcal{M})$ is a pre-fixpoint of T_{P}.

Finally, the following two lemmas (see Appendix D in the supplementary material for the proofs), provide evidence that $\mathcal{M}_{\mathsf{P}}$ is an extension of the classical well-founded semantics to the higher-order case:

Theorem 2
Let P be a program. Then, $\mathcal{M}_{\mathsf{P}}$ is a $\leq$-minimal model of P.

Theorem 3
For every propositional program P, $\mathcal{M}_{\mathsf{P}}$ coincides with the well-founded model of P.

In Appendix E in the supplementary material we give an example construction of $\mathcal{M}_{\mathsf{P}}$ for a given program P.

https://doi.org/10.1017/S1471068418000108 Published online by Cambridge University Press

7 Related and Future Work

In this section we compare our technique with the existing proposals for assigning semantics to higher-order logic programs with negation and we discuss possibly fruitful directions for future research.

The proposed extensional three-valued approach has important differences from the existing alternative ones, namely (Charalambidis *et al.* 2014), (Rondogiannis and Symeonidou 2016) and (Rondogiannis and Symeonidou 2017). As already mentioned in the introduction section, the technique in (Rondogiannis and Symeonidou 2017) is not extensional in the general case (it is however extensional if the source higher-order programs are *stratified* - see (Rondogiannis and Symeonidou 2017) for the formal definition of this notion). In this respect, the present approach is more general since it assigns an extensional semantics to *all* the programs of $\mathcal{HOL}$.

On the other hand, both of the techniques (Charalambidis *et al.* 2014) and (Rondogiannis and Symeonidou 2016) rely on an infinite-valued logic, and give a very fine-grained semantics to programs. This fine-grained nature of the infinite-valued approach makes it very appealing from a mathematical point of view. As it was recently demonstrated in (Ésik 2015; Carayol and Ésik 2016), in the case of first-order logic programs the infinite-valued approach satisfies all identities of *iteration theories* (Bloom and Ésik 1993), while the well-founded semantics does not. Since iteration theories provide an abstract framework for the evaluation of the merits of various semantic approaches for languages that involve recursion, these results appear to suggest that the infinite-valued approach has advantages from a mathematical point of view. On the other hand, the well-founded semantics is based on a much simpler three-valued logic, it is widely known to the logic programming community, and it has been studied and used for almost three decades. It is important however to emphasize that the differences between the infinite-valued and the well-founded approaches are not only a matter of mathematical elegance. In many programs, the two techniques behave differently. For example, given the program:

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

the approaches in (Charalambidis *et al.* 2014) and (Rondogiannis and Symeonidou 2016) will produce the model $\{(\mathtt{p}, \mathit{undef})\}$, while our present approach will produce the model $\{(\mathtt{p}, \mathit{false})\}$. In essence, our present approach *cancels such nested negations* (see also the discussion in (Denecker *et al.* 2012)[page 185, Example 1] on this issue), while the approaches in (Charalambidis *et al.* 2014) and (Rondogiannis and Symeonidou 2016) assign the value *undef* due to the circular dependence of `p` on itself through negation.

Similarly, for the following program (taken from (Rondogiannis and Symeonidou 2017)):

$$\begin{aligned}
&\mathtt{s} \leftarrow \lambda \mathtt{Q}.\mathtt{Q}\ (\mathtt{s}\ \mathtt{Q})\\
&\mathtt{p} \leftarrow \lambda \mathtt{R}.\mathtt{R}\\
&\mathtt{q} \leftarrow \lambda \mathtt{R}.\sim (\mathtt{w}\ \mathtt{R})\\
&\mathtt{w} \leftarrow \lambda \mathtt{R}.(\sim \mathtt{R})
\end{aligned}$$

the infinite-valued approaches will return the value *false* for the query `(s p)` and *undef* for `(s q)`, while our present approach will return the value *false* for both queries.

It is an interesting topic for future research to identify large classes of programs where the infinite-valued approach and the present one coincide. Possibly a good candidate for such a comparison would be the class of stratified higher-order logic programs

https://doi.org/10.1017/S1471068418000108 Published online by Cambridge University Press

(Rondogiannis and Symeonidou 2016). More generally, we believe that an investigation of the connections between the well-founded semantics and the infinite-valued one, will be quite rewarding.

Another interesting direction for future research would be to consider other possible semantics that can be revealed using approximation fixpoint theory. It is well-known that for first-order logic programs, approximation fixpoint theory can be used in order to define other useful fixpoints such as *stable*, *Kripke-Kleene*, and *supported* ones. We argue that using the approach proposed in this paper, this can also be done for higher-order logic programs. In particular, as in the first-order case, the fixpoints of T_{P} correspond to 3-*valued supported models of* P (recall that by Lemma 4 every fixpoint of T_{P} is a model of P). Moreover, since T_{P} is Fitting-monotonic over $\mathcal{H}_{\mathsf{P}}^{\mathsf{ma}} \otimes \mathcal{H}_{\mathsf{P}}^{\mathsf{am}}$ (which by Proposition 7 is a chain-complete poset), it has a least fixpoint which we can take as the *Kripke-Kleene fixpoint of* T_{P}. Finally, as in the case of first-order logic programs, the set of all fixpoints of $\mathcal{C}_{T_{\mathsf{P}}}$ is the set of *stable fixpoints of* T_{P}, and can be taken as the 3-valued stable models of P (by Theorem 6 in Appendix C in the supplementary material corresponding to this paper at the TPLP archives, every fixpoint of $\mathcal{C}_{T_{\mathsf{P}}}$ is also a fixpoint of T_{P} and therefore a model of P).

In contrast to the above 3-valued semantics, the definition of 2-*valued stable models* for higher-order logic programs seems less direct to obtain. In the case of first-order logic programs, the 2-valued stable models are those fixpoints of $\mathcal{C}_{T_{\mathsf{P}}}$ that are *exact* (Denecker *et al.* 2000; Denecker *et al.* 2004), i.e., that are of the form (I, I). In the higher-order case however, things are not that simple. Consider for example the positive higher-order logic program consisting only of the rule $\mathtt{p(R)} \leftarrow \mathtt{R}$, where $\mathtt{p}$ is of type $o \rightarrow o$. Since this is a positive program, it is reasonable to assume that it has a unique 2-valued stable model which assigns to $\mathtt{p}$ the identity relation over the set of classical two truth values. The meaning of this program under the semantics proposed in the present paper is captured by the pair of interpretations (I, J) where: $I(\mathtt{p})(\mathit{false}, \mathit{false}) = \mathit{false}$, $I(\mathtt{p})(\mathit{true}, \mathit{true}) = \mathit{true}$, $I(\mathtt{p})(\mathit{false}, \mathit{true}) = \mathit{false}$, and $J(\mathtt{p})(\mathit{false}, \mathit{false}) = \mathit{false}$, $J(\mathtt{p})(\mathit{true}, \mathit{true}) = \mathit{true}$, $J(\mathtt{p})(\mathit{false}, \mathit{true}) = \mathit{true}$. Notice that $I \neq J$ and this is due to the fact that I and J are 3-valued interpretations and not 2-valued ones as in the first-order case. In other words, under our semantics there does not exist an exact pair of interpretations that is a fixpoint of $\mathcal{C}_{T_{\mathsf{P}}}$ which we could take as the 2-valued stable semantics of the program. What needs to be done here is to generalize the notion of "*exact pair of interpretations*". Informally speaking, a pair (I, J) of Herbrand interpretations of P will be called exact if for every predicate constant $\mathtt{p}$ of the program, $I(\mathtt{p})$ coincides with $J(\mathtt{p})$ when they are applied to arguments that are *essentially* 2-*valued* (we need to define inductively for all types what it means for a relation to be essentially 2-valued). Notice that $I(\mathtt{p})$ agrees with $J(\mathtt{p})$ when applied to 2-valued arguments, i.e., when applied to $(\mathit{true}, \mathit{true})$ and $(\mathit{false}, \mathit{false})$. We believe that the approach sketched above leads to a characterization of the 2-valued stable models, but the details need to be carefully examined and specified.

In this paper we have claimed that the proposed approach is an appealing formulation for capturing the well-founded semantics for higher-order logic programs with negation. We have substantiated our claim by demonstrating that the proposed semantics generalizes the well-founded one for propositional programs. As suggested by one of the reviewers, this claim would be stronger if one could define alternative semantics that lead to the same model. One such approach would be to extend the original definition of the

https://doi.org/10.1017/S1471068418000108 Published online by Cambridge University Press

well-founded semantics (Gelder *et al.* 1991) which was based on the notion of *unfounded sets*. Another promising direction would be to derive an extension of Przymusinski's *iterated least fixpoint construction* (Przymusinski 1989) to the higher-order case. Both of these directions seem quite fruitful and non-trivial, and certainly require further investigation.

Supplementary Material

To view supplementary material for this article, please visit http://dx.doi.org/10.1017/S1471068418000108.

References

Bezem, M. 1999. Extensionality of simply typed logic programs. In *Logic Programming: The 1999 International Conference, Las Cruces, New Mexico, USA, November 29 - December 4, 1999*, D. D. Schreye, Ed. MIT Press, 395–410.

Bloom, S. L. and Ésik, Z. 1993. *Iteration Theories - The Equational Logic of Iterative Processes*. EATCS Monographs on Theoretical Computer Science. Springer.

Carayol, A. and Ésik, Z. 2016. An analysis of the equational properties of the well-founded fixed point. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town, South Africa, April 25-29, 2016.*, C. Baral, J. P. Delgrande, and F. Wolter, Eds. AAAI Press, 533–536.

Charalambidis, A., Ésik, Z., and Rondogiannis, P. 2014. Minimum model semantics for extensional higher-order logic programming with negation. *TPLP 14,* 4-5, 725–737.

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

Charalambidis, A., Rondogiannis, P., and Symeonidou, I. 2017. Equivalence of two fixed-point semantics for definitional higher-order logic programs. *Theor. Comput. Sci. 668*, 27–42.

Davey, B. A. and Priestley, H. A. 2002. *Introduction to Lattices and Order.* Cambridge University Press.

Denecker, M., Bruynooghe, M., and Vennekens, J. 2012. Approximation fixpoint theory and the semantics of logic and answers set programs. In *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, E. Erdem, J. Lee, Y. Lierler, and D. Pearce, Eds. Lecture Notes in Computer Science, vol. 7265. Springer, 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. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Boston, MA, 127–144.

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.

Ésik, Z. 2015. Equational properties of stratified least fixed points (extended abstract). In *Logic, Language, Information, and Computation - 22nd International Workshop, WoLLIC 2015, Bloomington, IN, USA, July 20-23, 2015, Proceedings*, V. de Paiva, R. J. G. B. de Queiroz, L. S. Moss, D. Leivant, and A. G. de Oliveira, Eds. Lecture Notes in Computer Science, vol. 9160. Springer, 174–188.

Ésik, Z. and Rondogiannis, P. 2015. A fixed point theorem for non-monotonic functions. *Theoretical Computer Science 574*, 18–38.

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

https://doi.org/10.1017/S1471068418000108 Published online by Cambridge University Press

Gelder, A. V., Ross, K. A., and Schlipf, J. S. 1991. The well-founded semantics for general logic programs. *J. ACM 38,* 3, 620–650.

Przymusinski, T. C. 1989. Every logic program has a natural stratification and an iterated least fixed point model. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania, USA*. 11–21.

Rondogiannis, P. and Symeonidou, I. 2016. Extensional semantics for higher-order logic programs with negation. In *Logics in Artificial Intelligence - 15th European Conference, JELIA 2016, Larnaca, Cyprus, November 9-11, 2016, Proceedings*, L. Michael and A. C. Kakas, Eds. Lecture Notes in Computer Science, vol. 10021. 447–462.

Rondogiannis, P. and Symeonidou, I. 2017. The intricacies of three-valued extensional semantics for higher-order logic programs. *TPLP 17,* 5-6, 974–991.

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

van Emden, M. H. and Kowalski, R. A. 1976. The semantics of predicate logic as a programming language. *J. ACM 23,* 4, 733–742.

Wadge, W. W. 1991. Higher-order horn logic programming. In *Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, Oct. 28 - Nov 1, 1991*, V. A. Saraswat and K. Ueda, Eds. MIT Press, 289–303.

https://doi.org/10.1017/S1471068418000108 Published online by Cambridge University Press