TLP **17** (5–6): 974–991, 2017. © Cambridge University Press 2017
doi:10.1017/S1471068417000357 First published online 23 August 2017

The intricacies of three-valued extensional semantics for higher-order logic programs

PANOS RONDOGIANNIS and IOANNA SYMEONIDOU
National and Kapodistrian University of Athens, Athens, Greece
(*e-mail*:prondo@di.uoa.gr,i.symeonidou@di.uoa.gr)

submitted 10 March 2017; revised 17 July 2017; accepted 27 July 2017

Abstract

M. Bezem defined an extensional semantics for positive higher-order logic programs. Recently, it was demonstrated by Rondogiannis and Symeonidou that Bezem's technique can be extended to higher-order logic programs with negation, retaining its extensional properties, provided that it is interpreted under a logic with an infinite number of truth values. Rondogiannis and Symeonidou also demonstrated that Bezem's technique, when extended under the stable model semantics, does not in general lead to extensional stable models. In this paper, we consider the problem of extending Bezem's technique under the well-founded semantics. We demonstrate that the well-founded extension *fails* to retain extensionality in the general case. On the positive side, we demonstrate that for stratified higher-order logic programs, extensionality is indeed achieved. We analyze the reasons of the failure of extensionality in the general case, arguing that a three-valued setting cannot distinguish between certain predicates that appear to have a different behaviour inside a program context, but which happen to be identical as three-valued relations.

KEYWORDS: Extensional higher-order logic programming, Negation in logic programming.

1 Introduction

Recent research (Wadge 1991; Bezem 1999, 2001; Kountouriotis *et al.* 2005; Charalambidis *et al.* 2013, 2017; Rondogiannis and Symeonidou 2016) has investigated the possibility of providing *extensional* semantics to higher-order logic programming. Under an extensional semantics, predicates denote sets, and therefore one can use standard set theory in order to understand programs and reason about them. Of course, extensionality comes with a price: to obtain an extensional semantics, one usually has to consider higher-order logic programs with a relatively restricted syntax. Actually, this is a main difference between the extensional and the more traditional *intensional* approaches to higher-order logic programming such as those described by Miller and Nadathur (2012) and Chen *et al.* (1993): the latter languages have a richer syntax and expressive capabilities but they are not usually amenable to a standard set-theoretic semantics.

There exist two main research directions for providing extensional semantics to higher-order logic programs. The first one (Wadge 1991; Kountouriotis *et al.* 2005; Charalambidis *et al.* 2013, 2014) has been developed using domain-theoretic tools, and resembles the techniques for assigning denotational semantics to functional languages. The second approach (Bezem 1999, 2001; Rondogiannis and Symeonidou 2016) relies on the syntactic entities that exist in a program, and is based on processing the ground instantiation of the program. The two research directions are not unrelated: it has recently been shown by Charalambidis *et al.* (2017) that for a broad class of positive programs, the two approaches coincide with respect to ground atoms.

In this paper, we focus exclusively on the second extensional approach. This approach was initially proposed by Bezem (1999; 2001) for *positive* (i.e., negationless) higher-order logic programs. Recently, it was demonstrated by Rondogiannis and Symeonidou (2016) that by combining the technique of Bezem (1999; 2001) with the infinite-valued semantics of Rondogiannis and Wadge (2005), we obtain an extensional semantics for higher-order logic programs with negation. In the extended version (2017) of the paper by Rondogiannis and Symeonidou (2016), a negative and unexpected result is established: by combining the technique of Bezem (1999; 2001) with the stable model semantics (Gelfond and Lifschitz 1988), we get a semantics that is not necessarily extensional! It remained as an open problem of Rondogiannis and Symeonidou (2016; 2017) whether the combination of the technique of Bezem (1999; 2001) with the well-founded approach (Gelder *et al.* 1991) leads to an extensional semantics. It is exactly this problem that we undertake to solve in the present paper.

We demonstrate that the well-founded extension of Bezem's technique *fails* to retain extensionality in the general case. On the positive side, we prove that for stratified higher-order logic programs, extensionality is indeed achieved. We analyze the reasons of the failure of extensionality in the general case, and claim that this is not an inherent shortcoming of Bezem's approach but a more general phenomenon. In particular, we argue that restricting attention to three-valued logic appears to "throw away too much information" and makes predicates that are expected to have different behaviours, appear as identical three-valued relations. The main contributions of the present paper can be summarized as follows:

- We demonstrate that the well-founded adaptation of Bezem's technique does not, in general, lead to an extensional model. In particular, we exhibit a program with a non-extensional well-founded model. This result, despite its negative flavour, indicates that the addition of negation to higher-order logic programming is not such a straightforward task as it was possibly initially anticipated. Notice that, as it was recently demonstrated by Rondogiannis and Symeonidou (2017), the stable model adaptation of Bezem's technique is also non-extensional in general.
- Despite the above negative result, we prove that the well-founded adaptation of Bezem's technique gives an extensional two-valued model in the case of *stratified* programs. This result affirms the importance and the well-behaved

https://doi.org/10.1017/S1471068417000357 Published online by Cambridge University Press

nature of stratified programs, which was, until now, only known for the first-order case.

- We study the more general question of the possible existence of an *alternative* extensional three-valued semantics for higher-order logic programs with negation. We indicate that in order to achieve such a semantics, one has to make some (arguably) non-standard assumptions regarding the behaviour of negation.

The rest of the paper is organized as follows. Section 2 introduces the basic notions and the advantages of the extensional approach to the semantics of higher-order logic programming. Section 3 presents in an intuitive way the main concepts and results of the paper. Section 4 introduces the syntax, and Section 5 the semantics of our source language. Section 6 demonstrates that Bezem's approach is not extensional under the well-founded semantics. In Section 7, it is established that stratified programs have an extensional, two-valued, well-founded semantics. Section 8 concludes by discussing the restrictions that any reasonable three-valued semantics would have with respect to extensionality. The proofs of all results are given in corresponding appendices, included in the supplementary material accompanying the paper at the TPLP archive.

2 Extensional higher-order logic programming

Wadge (1991) suggested that if we appropriately restrict the syntax of higher-order logic programming, then we can obtain languages that can be assigned a standard denotational semantics in which predicates denote sets. In other words, for such syntactically restricted languages one can apply traditional domain-theoretic notions and tools that have been used extensively in higher-order functional programming. The most crucial syntactic restriction imposed by Wadge [and also later independently by Bezem (1999)] is the following:

The extensionality syntactic restriction: *In the head of every rule in a program, each argument of predicate type must be a variable, and all such variables must be distinct.*

Example 1

The following is a legitimate program that defines the union of two relations `P`, `Q` (for the moment we use ad-hoc Prolog-like syntax):

```
union(P,Q)(X):-P(X).
union(P,Q)(X):-Q(X).
```

However, the following program does not satisfy Wadge's restriction:

```
q(a).
r(q).
```

because the predicate constant `q` appears as an argument in the head of a rule. Similarly, the program

```
p(Q,Q):-Q(a).
```

https://doi.org/10.1017/S1471068417000357 Published online by Cambridge University Press

is problematic because the predicate variable `Q` is used twice in the head of the rule.

The advantages of extensionality were identified by Wadge and Bezem in their respective papers. First of all, under the extensional approach, program predicates can be understood declaratively in terms of extensional notions. For example, the program

```
map(R,[],[]).
map(R,[H1|T1],[H2|T2]):-R(H1,H2), map(R,T1,T2).
```

can be understood in a similar way as the well-known `map` function of Haskell. Moreover, since under the extensional approach predicates denote sets, two predicates that are true of the same arguments, are considered indistinguishable. So, for example, if we define two sorting predicates `merge_sort` and `quick_sort` that have the same type, say τ, and that perform the same task (possibly with different efficiency), it is *guaranteed* that any predicate which operates on relations of type τ will have the same behaviour whether it is given `merge_sort` or `quick_sort` as an argument. As mentioned by Wadge (1991), "extensionality means exactly that predicates are used as *black boxes* - and the "black box" concept is central to all kinds of engineering". It is this property that makes extensional languages so appealing (and is actually one of the greatest assets of traditional functional programming).

Another important advantage of this declarative approach to higher-order logic programming is that many techniques and ideas that have been successfully developed in the functional programming world (such as program transformations, optimizations, techniques for proving program correctness, and so on), could be transferred to the higher-order logic programming domain, opening in this way promising new research directions for logic programming as a whole.

3 An intuitive overview of the proposed approach

In this paper, we consider the semantic technique for positive higher-order logic programs proposed by Bezem (1999; 2001) and we investigate whether it can be applied in order to provide an extensional well-founded semantics for higher-order logic programs with negation in clause bodies. In this section, we give an intuitive description of Bezem's idea and we outline how we use it when negation is added to programs.

Given a positive higher-order logic program, the starting idea behind Bezem's approach is to take its "ground instantiation", in which we replace variables with well-typed terms that can be created using syntactic entities that appear in the program. For example, consider the higher-order program below:

```
q(a).
q(b).
p(Q):-Q(a).
id(R)(X):-R(X).
```

https://doi.org/10.1017/S1471068417000357 Published online by Cambridge University Press

In order to obtain the ground instantiation of this program, we consider each clause and replace each variable of the clause with a ground term that has the same type as the variable under consideration (the formal definition of this procedure will be given in Definition 8). In this way, we obtain the following infinite program:

```
q(a).
q(b).
p(q):-q(a).
id(q)(a):-q(a).
id(q)(b):-q(b).
p(id(q)):-id(q)(a).
        ...
```

One can now treat the new program as an infinite propositional one (i.e., each ground atom can be seen as a propositional variable). This implies that we can use the standard least fixed-point construction of classical logic programming [see for example Lloyd (1987)] in order to compute the set of atoms that should be taken as "true". In our example, the least fixed-point will contain atoms such as `q(a)`, `q(b)`, `p(q)`, `id(q)(a)`, `id(q)(b)`, `p(id(q))`, and so on.

Bezem demonstrated that the least fixed-point semantics of the ground instantiation of every positive higher-order logic program of the language considered by Bezem (1999; 2001) is *extensional* in a sense that can be explained as follows. In our example, `q` and `id(q)` are equal since they are both true of exactly the constants `a` and `b`. Therefore, we expect that (for example) if `p(q)` is true, then `p(id(q))` is also true, because `q` and `id(q)` should be considered as indistinguishable. This property of "indistinguishability" is formally defined by Bezem (1999; 2001) and it is demonstrated that it holds in the least fixed-point of the immediate consequence operator of the ground instantiation of every program that abides to the simple extensionality syntactic restriction given in the previous section (and formally described by Definition 5 later in the paper).

The key idea behind extending Bezem's semantics in order to apply to higher-order logic programs with negation is straightforward to state: given such a program, we first take its ground instantiation. The resulting program is a (possibly infinite) propositional program with negation, and therefore we can compute its semantics in any standard way that exists for obtaining the meaning of such programs. For example, one could use the well-founded semantics (Gelder *et al.* 1991), the stable model semantics (Gelfond and Lifschitz 1988) or the infinite-valued semantics (Rondogiannis and Wadge 2005), and then proceed to examine whether the well-founded model (respectively, each stable model, or the minimum infinite-valued model) is extensional in the sense of Bezem (1999; 2001) (informally described above).

An open problem posed by Rondogiannis and Symeonidou (2017) was whether Bezem's technique, under the well-founded semantics, always leads to an extensional well-founded model. As we are going to see in the subsequent sections, this is not the case. In particular, we exhibit a program containing three predicates `s`, `p` and `q`, such that `p` and `q` are extensionally equal under the well-founded semantics, but

https://doi.org/10.1017/S1471068417000357 Published online by Cambridge University Press

`s(p)` and `s(q)` have a different truth value. On the positive side, we prove that every stratified higher-order logic program with negation has an extensional well-founded model. In this sense, we identify a broad class of programs that are well-behaved in terms of extensionality.

4 The syntax of $\mathscr{H}$

In this section, we define the syntax of the language $\mathscr{H}$ that we use throughout the paper. $\mathscr{H}$ is based on a simple type system with two base types: o, the boolean domain, and ι, the domain of data objects. The composite types are partitioned into three classes: functional (assigned to function symbols), predicate (assigned to predicate symbols) 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). As usual, the binary operator $\to$ is right-associative. A functional type that is different than ι will often be written in the form $\iota^n \to \iota$, $n \geqslant 1$. Moreover, it can be easily seen that every predicate type π can be written in the form $\rho_1 \to \cdots \to \rho_n \to o$, $n \geqslant 0$ (for $n = 0$ we assume that $\pi = o$). We proceed by defining the syntax of $\mathscr{H}$:

Definition 2
The *alphabet* of $\mathscr{H}$ consists of the following: *predicate variables* of every predicate type π (denoted by capital letters such as $\mathsf{Q}, \mathsf{R}, \mathsf{S}, \ldots$); *individual variables* of type ι (denoted by capital letters such as $\mathsf{X}, \mathsf{Y}, \mathsf{Z}, \ldots$); *predicate constants* of every predicate type π (denoted by lowercase letters such as $\mathsf{p}, \mathsf{q}, \mathsf{r}, \ldots$); *individual constants* of type ι (denoted by lowercase letters such as $\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots$); *function symbols* of every functional type $\sigma \neq \iota$ (denoted by lowercase letters such as $\mathsf{f}, \mathsf{g}, \mathsf{h}, \ldots$); the *inverse implication* constant $\leftarrow$; the *negation* constant $\sim$; the comma; the left and right parentheses; and the *equality* constant $\approx$ for comparing terms of type ι.

Arbitrary variables will usually be denoted by V and its subscripted versions.

Definition 3
The set of *terms* of $\mathscr{H}$ is defined as follows: every predicate variable (respectively, predicate constant) of type π is a term of type π; every individual variable (respectively, individual constant) of type ι is a term of type ι; if f is an n-ary function symbol and $\mathsf{E}_1, \ldots, \mathsf{E}_n$ are terms of type ι, then $(\mathsf{f}\ \mathsf{E}_1 \cdots \mathsf{E}_n)$ is a term of type ι; if E_1 is a term of type $\rho \to \pi$ and E_2 a term of type ρ, then $(\mathsf{E}_1\ \mathsf{E}_2)$ is a term of type π.

https://doi.org/10.1017/S1471068417000357 Published online by Cambridge University Press

Definition 4
The set of *expressions* of $\mathscr{H}$ is defined as follows: a term of type ρ is an expression of type ρ; if E is a term of type o, then $(\sim\mathsf{E})$ is an expression of type o; if E_1 and E_2 are terms of type ι, then $(\mathsf{E}_1 \approx \mathsf{E}_2)$ is an expression of type o.

We will omit parentheses when no confusion arises. To denote that an expression E has type ρ, we will often write $\mathsf{E} : \rho$. We will write $vars(\mathsf{E})$ to denote the set of all the variables in E. Expressions (respectively, terms) that have no variables will be referred to as *ground expressions* (respectively, *ground terms*). Terms of type o will be referred to as *atoms*. Expressions of type o that do not contain negation, i.e., expressions of the form $(\mathsf{E}_1 \approx \mathsf{E}_2)$ or atoms, will be called *positive literals*, while expressions of the form $(\sim\mathsf{E})$ will be called *negative literals*. A *literal* is either a positive literal or a negative literal.

Definition 5
A *clause* of $\mathscr{H}$ is a formula $\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n \leftarrow \mathsf{L}_1, \ldots, \mathsf{L}_m$, where p is a predicate constant of type $\rho_1 \rightarrow \cdots \rightarrow \rho_n \rightarrow o$, $\mathsf{V}_1, \ldots, \mathsf{V}_n$ are distinct variables of types $\rho_1, \ldots, \rho_n$, respectively, and $\mathsf{L}_1, \ldots, \mathsf{L}_m$ are literals. The term $\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n$ is called the *head* of the clause, the variables $\mathsf{V}_1, \ldots, \mathsf{V}_n$ are the *formal parameters* of the clause and the conjunction $\mathsf{L}_1, \ldots, \mathsf{L}_m$ is its *body*. A *program* P of $\mathscr{H}$ is a finite set of clauses.

Example 2
The program below defines the `subset` relation over unary predicates:

```
subset S1 S2 ← ∼(nonsubset S1 S2)
nonsubset S1 S2 ← (S1 X), ∼(S2 X)
```

Given unary predicates `p` and `q`, `subset p q` is true iff `p` is a subset of `q`.

Example 3
For a more "real-life" higher-order logic program with negation, assume that we have a unary predicate `movie M` and a binary predicate `ranking M R` which returns the ranking `R` of a given movie `M`. Consider also the following first-order predicate that defines a preference over movies based on their ranking:

```
prefer M1 M2 ← movie M1, movie M2, ranking M1 R1, ranking M2 R2, R1>R2.
```

The following higher-order predicate `winnow` [see for example Chomicki (2003)] can be used to select all the "best" tuples `T` out of a given relation `R` based on a preference relation `P`:

```
winnow P R T ← R T, ∼(bypassed P R T).
bypassed P R T ← R T1, P T1 T.
```

Intuitively, `winnow` returns all the tuples `T` of the relation `R` such that there does not exist any tuple `T1` in the relation `R` that is better from `T` with respect to the preference relation `P`. For example, if we ask the query `?- winnow prefer movie T.`, we expect as answers all those movies that have the highest possible ranking. Notice that since `winnow` is a higher-order predicate, it can be invoked with different

https://doi.org/10.1017/S1471068417000357 Published online by Cambridge University Press

arguments; for example, it can be used to select out of a `book` relation, all those books that have the lowest possible price, or out of a `flight` relation all those flights that go to London, and so on.

The ground instantiation of a program is described by the following definitions:

Definition 6
A *substitution* θ is a finite set of the form $\{\mathsf{V}_1/\mathsf{E}_1,\ldots,\mathsf{V}_n/\mathsf{E}_n\}$ where the V_i's are different variables and each E_i is a term having the same type as V_i. We write $dom(\theta)$ to denote the domain $\{\mathsf{V}_1,\ldots,\mathsf{V}_n\}$ of θ. If all the terms $\mathsf{E}_1,\ldots,\mathsf{E}_n$ are ground, θ is called a *ground substitution.*

Definition 7
Let θ be a substitution and E be an expression. Then, $\mathsf{E}\theta$ is an expression obtained from E as follows:

- $\mathsf{E}\theta = \mathsf{E}$ if E is a predicate constant or individual constant;
- $\mathsf{V}\theta = \theta(\mathsf{V})$ if $\mathsf{V} \in dom(\theta)$; otherwise, $\mathsf{V}\theta = \mathsf{V}$;
- $(\mathsf{f}\ \mathsf{E}_1 \cdots \mathsf{E}_n)\theta = (\mathsf{f}\ \mathsf{E}_1\theta \cdots \mathsf{E}_n\theta)$;
- $(\mathsf{E}_1\ \mathsf{E}_2)\theta = (\mathsf{E}_1\theta\ \mathsf{E}_2\theta)$;
- $(\sim\mathsf{E})\theta = (\sim\mathsf{E}\theta)$;
- $(\mathsf{E}_1 \approx \mathsf{E}_2)\theta = (\mathsf{E}_1\theta \approx \mathsf{E}_2\theta)$.

If θ is a ground substitution such that $vars(\mathsf{E}) \subseteq dom(\theta)$, then the ground expression $\mathsf{E}\theta$ is called a *ground instance* of E.

Definition 8
Let P be a program. A *ground instance of a clause* $\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n \leftarrow \mathsf{L}_1,\ldots,\mathsf{L}_m$ of P is a formula $(\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n)\theta \leftarrow \mathsf{L}_1\theta,\ldots,\mathsf{L}_m\theta$, where θ is a ground substitution whose domain is the set of all variables that appear in the clause, such that for every $\mathsf{V} \in dom(\theta)$ with $\mathsf{V} : \rho$, $\theta(\mathsf{V})$ is a ground term of type ρ that has been formed with predicate constants, function symbols and individual constants that appear in P. The *ground instantiation of a program* P, denoted by $\mathsf{Gr}(\mathsf{P})$, is the (possibly infinite) set that contains all the ground instances of the clauses of P.

5 The semantics of $\mathscr{H}$

Bezem (1999; 2001) developed a semantics for higher-order logic programs which generalizes the familiar Herbrand-model semantics of classical (first-order) logic programs. In this section, we extend Bezem's semantics to the case of higher-order logic programs with negation.

In order to interpret the programs of $\mathscr{H}$, we need to specify the semantic domains in which the expressions of each type τ are assigned their meanings. The following definition is a slightly modified version of the corresponding definition of Bezem (1999; 2001), and it implies that the expressions of predicate types should be understood as representing functions. We use $[S_1 \rightarrow S_2]$ to denote the set of (possibly partial) functions from a set S_1 to a set S_2. The possibility to have a partial function arises due to a technicality which is explained in the remark just above Definition 11.

https://doi.org/10.1017/S1471068417000357 Published online by Cambridge University Press

Definition 9
A *functional type structure* $\mathscr{S}$ for $\mathscr{H}$ consists of two non-empty sets D and A together with an assignment $[\![\tau]\!]$ to each type τ of $\mathscr{H}$, so that the following are satisfied:

- $[\![\iota]\!] = D$;
- $[\![\iota^n \to \iota]\!] = D^n \to D$;
- $[\![o]\!] = A$;
- $[\![\rho \to \pi]\!] \subseteq [[\![\rho]\!] \to [\![\pi]\!]]$.

Given a functional type structure $\mathscr{S}$, any function $v : [\![o]\!] \to \{false, 0, true\}$ will be called a *three-valued valuation function* (or simply *valuation function*) for $\mathscr{S}$. We will use the term *two-valued valuation functions* to distinguish the subset of valuation functions which do not assign the value 0 to any element of $[\![o]\!]$, i.e., the functions $v : [\![o]\!] \to \{false, true\}$.

It is customary in the study of the semantics of logic programming languages to restrict attention to *Herbrand interpretations*. Given a program P, a Hebrand interpretation is one that has as its underlying universe the so-called *Herbrand universe* of P:

Definition 10
For a program P, we define the *Herbrand universe* for every argument type ρ, denoted by $U_{\mathsf{P},\rho}$ to be the set of all ground terms of type ρ that can be formed out of the individual constants, function symbols, and predicate constants in the program. Moreover, we define $U^+_{\mathsf{P},o}$ to be the set of all ground expressions of type o, that can be formed out of the above symbols, i.e., the set $U^+_{\mathsf{P},o} = U_{\mathsf{P},o} \cup \{(\mathsf{E}_1 \approx \mathsf{E}_2) \mid \mathsf{E}_1, \mathsf{E}_2 \in U_{\mathsf{P},\iota}\} \cup \{(\sim\mathsf{E}) \mid \mathsf{E} \in U_{\mathsf{P},o}\}$.

Following Bezem (1999; 2001), we take D and A in Definition 9 to be equal to $U_{\mathsf{P},\iota}$ and $U^+_{\mathsf{P},o}$ respectively. Then, for each predicate type $\rho \to \pi$, each element of $U_{\mathsf{P},\rho\to\pi}$ can be perceived as a function mapping elements of $[\![\rho]\!]$ to elements of $[\![\pi]\!]$, through syntactic application mapping. That is, $\mathsf{E} \in U_{\mathsf{P},\rho\to\pi}$ can be viewed as the function mapping each term $\mathsf{E}' \in U_{\mathsf{P},\rho}$ to the term $(\mathsf{E}\,\mathsf{E}') \in U_{\mathsf{P},\pi}$. Similarly, every n-ary function symbol f appearing in P can be viewed as the function mapping each element $(\mathsf{E}_1, \ldots, \mathsf{E}_n) \in U^n_{\mathsf{P},\iota}$ to the term $(\mathsf{f}\,\mathsf{E}_1 \cdots \mathsf{E}_n) \in U_{\mathsf{P},\iota}$.

Remark: There is a small technicality here which we need to clarify. In the case where $\rho = o$, $\mathsf{E} \in U_{\mathsf{P},o\to\pi}$ is a partial function because it maps elements of $U_{\mathsf{P},o}$ (and not of $U^+_{\mathsf{P},o}$) to elements of $U_{\mathsf{P},\pi}$; this is due to the fact that our syntax does not allow an expression of type $o \to \pi$ to take as argument an expression of the form $(\mathsf{E}_1 \approx \mathsf{E}_2)$ nor of the form $(\sim\mathsf{E})$. In all other cases (i.e., when $\rho \neq o$), E represents a total function.

Definition 11
A *(three-valued) Herbrand interpretation* I of a program P consists of the following:

(1) the functional type structure $\mathscr{S}_{\mathsf{P}}$, such that $D = U_{\mathsf{P},\iota}$, $A = U^+_{\mathsf{P},o}$ and $[\![\rho \to \pi]\!] = U_{\mathsf{P},\rho\to\pi}$ for every predicate type $\rho \to \pi$, called the Herbrand-type structure of P;

https://doi.org/10.1017/S1471068417000357 Published online by Cambridge University Press

(2) the assignment to each individual constant c in P, of the element $I(\mathsf{c}) = \mathsf{c}$; to each predicate constant p in P, of the element $I(\mathsf{p}) = \mathsf{p}$; to each function symbol f in P, of the element $I(\mathsf{f}) = \mathsf{f}$;

(3) a valuation function $v_I(\cdot)$ for $\mathscr{S}_\mathsf{P}$, assigning to each element of $U^+_{\mathsf{P},o}$ an element in $\{false, 0, true\}$, while satisfying the following:

- for all $\mathsf{E}_1, \mathsf{E}_2 \in U_{\mathsf{P},\iota}$, $v_I((\mathsf{E}_1 \approx \mathsf{E}_2)) = \begin{cases} false, & \text{if } \mathsf{E}_1 \neq \mathsf{E}_2 \\ true, & \text{if } \mathsf{E}_1 = \mathsf{E}_2 \end{cases}$;
- for all $\mathsf{E} \in U_{\mathsf{P},o}$, $v_I((\sim\mathsf{E})) = \begin{cases} false, & \text{if } v_I(\mathsf{E}) = true \\ 0, & \text{if } v_I(\mathsf{E}) = 0 \\ true, & \text{if } v_I(\mathsf{E}) = false \end{cases}$.

We call $v_I(\cdot)$ the *valuation function of* I and omit the reference to $\mathscr{S}_\mathsf{P}$, since the latter is common to all Herbrand interpretations of a program. In fact, individual Herbrand interpretations are only set apart by their valuation functions. If the valuation function $v_I(\cdot)$ is two-valued, then I will also be called a *two-valued Herbrand interpretation.*

Definition 12
A *Herbrand state* (or simply *state*) s of a program P is a function that assigns to each variable V of type ρ an element of $U_{\mathsf{P},\rho}$.

Given a Herbrand interpretation I and state s, we can define the semantics of expressions with respect to I and s.

Definition 13
Let P be a program. Also, let I be a Herbrand interpretation and s a Herbrand state of P. Then, the semantics of expressions with respect to I and s is defined as follows:

- $[\![\mathsf{c}]\!]_{I,s} = I(\mathsf{c}) = \mathsf{c}$, for every individual constant c;
- $[\![\mathsf{p}]\!]_{I,s} = I(\mathsf{p}) = \mathsf{p}$, for every predicate constant p;
- $[\![\mathsf{V}]\!]_{I,s} = s(\mathsf{V})$, for every variable V;
- $[\![(\mathsf{f}\ \mathsf{E}_1 \cdots \mathsf{E}_n)]\!]_{I,s} = (I(\mathsf{f})\ [\![\mathsf{E}_1]\!]_{I,s} \cdots [\![\mathsf{E}_n]\!]_{I,s}) = (\mathsf{f}\ [\![\mathsf{E}_1]\!]_{I,s} \cdots [\![\mathsf{E}_n]\!]_{I,s})$, for every n-ary function symbol f;
- $[\![(\mathsf{E}_1\ \mathsf{E}_2)]\!]_{I,s} = ([\![\mathsf{E}_1]\!]_{I,s}\ [\![\mathsf{E}_2]\!]_{I,s})$;
- $[\![(\mathsf{E}_1 \approx \mathsf{E}_2)]\!]_{I,s} = ([\![\mathsf{E}_1]\!]_{I,s} \approx [\![\mathsf{E}_2]\!]_{I,s})$;
- $[\![(\sim\mathsf{E})]\!]_{I,s} = (\sim[\![\mathsf{E}]\!]_{I,s})$.

It is easy to see that the semantic function $[\![\cdot]\!]$ is well defined, in the sense that, for every Herbrand state s and every expression E of every argument type ρ, we have $[\![\mathsf{E}]\!]_{I,s} \in [\![\rho]\!]$. Note that this makes $[\![\mathsf{E}]\!]_{I,s}$ a ground expression of the language. Also, note that if E is a ground expression, then $[\![\mathsf{E}]\!]_{I,s} = \mathsf{E}$; therefore, if E is a ground literal, we can write $v_I(\mathsf{E})$ instead of $v_I([\![\mathsf{E}]\!]_{I,s})$. Stretching this abuse of notation a little further, we can extend a valuation function to assign truth values to ground conjunctions of literals; this allows us to define the concept of Herbrand models for our higher-order programs in the same way as in classical logic programming.

https://doi.org/10.1017/S1471068417000357 Published online by Cambridge University Press

Definition 14
Let P be a program and I be a Herbrand interpretation of P. We define $v_I(\mathsf{L}_1, \ldots, \mathsf{L}_n) = min\{v_I(\mathsf{L}_1), \ldots, v_I(\mathsf{L}_n)\}$ for all $\mathsf{L}_1, \ldots, \mathsf{L}_n \in U^+_{\mathsf{P},o}$. Moreover, we say I is a *model* of P if $v_I([\![\mathsf{A}]\!]_{I,s}) \geqslant v_I([\![\mathsf{L}_1]\!]_{I,s}, \ldots, [\![\mathsf{L}_m]\!]_{I,s})$ holds for every clause $\mathsf{A} \leftarrow \mathsf{L}_1, \ldots, \mathsf{L}_m$ and every Herbrand state s of P.

Bezem's semantics is based on the observation that, given a positive higher-order program, we can use the minimum model of its ground instantiation as a (two-valued) valuation function defining a Herbrand interpretation for the initial program itself. We follow the same idea but now for programs with negation: we can use as the valuation function of a given $\mathscr{H}$ program, the Herbrand model defined by any semantic approach that applies to its ground instantiation. Actually, we demonstrate (see Theorem 1 below) that any interpretation I of the higher-order program P will be a minimal model of P, if its chosen valuation function is a minimal model of $\mathsf{Gr(P)}$.

We consider two different notions of minimality, based on the *truth* ordering $\leqslant$ and the *Fitting* ordering $\preceq$ of truth values, respectively. Recall that $\leqslant$ is the partial order defined by $\mathit{false} \leqslant 0 \leqslant \mathit{true}$, while $\preceq$ is the partial order defined by $0 \preceq \mathit{false}$ and $0 \preceq \mathit{true}$.

Definition 15
If I and J are two Herbrand interpretations of a higher-order program P, we say $I \leqslant J$ (respectively, $I \preceq J$) if, for all atoms A in $U_{\mathsf{P},o}$ we have $v_I(\mathsf{A}) \leqslant v_J(\mathsf{A})$ (resp., $v_I(\mathsf{A}) \preceq v_J(\mathsf{A})$). If M is a model of P, then we say it is $\leqslant$-minimal (resp., $\preceq$-minimal) if there does not exist a different model N of P, such that $N \leqslant M$ (resp., $N \preceq M$).

Theorem 1
Let P be a program and let $\mathsf{Gr(P)}$ be its ground instantiation. Also, let M be a partial interpretation[1] of $\mathsf{Gr(P)}$ and let $\mathscr{M}$ be the Herbrand interpretation of P, such that $v_{\mathscr{M}}(\mathsf{A}) = M(\mathsf{A})$ for every $\mathsf{A} \in U_{\mathsf{P},o}$. Then, $\mathscr{M}$ is a Herbrand model of P if and only if M is a model of $\mathsf{Gr(P)}$. Moreover, $\mathscr{M}$ is $\leqslant$-minimal (respectively, $\preceq$-minimal) if and only if M is $\leqslant$-minimal (respectively, $\preceq$-minimal).

As an application of the above developments, we define two special Herbrand interpretations of higher-order programs, employing the well-known perfect model (Apt *et al.* 1988; Gelder 1989) and well-founded model (Gelder *et al.* 1991) of the ground instantiation of a program, as valuation functions.

Definition 16
Let P be a program and let $\mathsf{Gr(P)}$ be the ground instantiation of P. Also, let $N_{\mathsf{Gr(P)}}$ be the perfect model[1] (if this exists) and $M_{\mathsf{Gr(P)}}$ be the well-founded model[1] of $\mathsf{Gr(P)}$. We define $\mathscr{N}_{\mathsf{P}}$ to be the two-valued Herbrand interpretation of P such that $v_{\mathscr{N}_{\mathsf{P}}}(\mathsf{A}) = N_{\mathsf{Gr(P)}}(\mathsf{A})$ for every $\mathsf{A} \in U_{\mathsf{P},o}$. Similarly, we define $\mathscr{M}_{\mathsf{P}}$ to be the three-valued Herbrand interpretation of P such that $v_{\mathscr{M}_{\mathsf{P}}}(\mathsf{A}) = M_{\mathsf{Gr(P)}}(\mathsf{A})$ for every $\mathsf{A} \in U_{\mathsf{P},o}$.

[1] See the supplementary material accompanying the paper at the TPLP archive for the relevant definitions.

https://doi.org/10.1017/S1471068417000357 Published online by Cambridge University Press

Clearly, by Theorem 1, $\mathscr{N}_{\mathsf{P}}$ (if it exists) is a two-valued minimal model and $\mathscr{M}_{\mathsf{P}}$ is a three-valued minimal model of P. In the following sections, we investigate their suitability for providing extensional semantics for $\mathscr{H}$ programs. In particular, we examine if each of them enjoys the extensionality property, formally defined by Bezem (1999; 2001) through relations $\cong_{v,\tau}$ over the set of expressions of a given type τ and under a given valuation function v. These relations intuitively express extensional equality of type τ, in the sense discussed in Section 3. For the purposes of this paper, only extensional equality of argument types will be needed, for which the formal definition is as follows:

Definition 17
Let $\mathscr{S}$ be a functional type structure and v be a valuation function for $\mathscr{S}$. For every argument type ρ, we define the relations $\cong_{v,\rho}$ on $[\![\rho]\!]$ as follows: Let $d, d' \in [\![\rho]\!]$; then $d \cong_{v,\rho} d'$ if and only if

(1) $\rho = \iota$ and $d = d'$, or
(2) $\rho = o$ and $v(d) = v(d')$, or
(3) $\rho = \rho' \to \pi$ and $d\,e \cong_{v,\pi} d'\,e'$ for all $e, e' \in [\![\rho']\!]$, such that $e \cong_{v,\rho'} e'$ and $d\,e, d'\,e'$ are both defined.

One can easily verify that, for all $d, d' \in [\![\rho_1 \to \cdots \to \rho_n \to o]\!]$, $e_1, e'_1 \in [\![\rho_1]\!]$, $\ldots$, $e_n, e'_n \in [\![\rho_n]\!]$, if $d \cong_{v,\rho_1 \to \cdots \to \rho_n \to o} d'$, $e_1 \cong_{v,\rho_1} e'_1, \ldots, e_n \cong_{v,\rho_n} e'_n$ and $d\,e_1 \cdots e_n, d'\,e'_1 \cdots e'_n$ are both defined, then $v(d\ e_1 \cdots e_n) = v(d'\ e'_1 \cdots e'_n)$.

Generally, it is not guaranteed that such relations will be equivalence relations; rather they are partial equivalences [they are shown by Bezem (1999) to be symmetric and transitive]. Whether they are moreover reflexive, depends on the specific valuation function.

The above discussion leads to the notion of *extensional interpretation*:

Definition 18
Let P be a program and let I be a Herbrand interpretation of P with valuation function v_I. We say I is *extensional* if for all argument types ρ the relations $\cong_{v_I,\rho}$ are reflexive, i.e., for all $\mathsf{E} \in [\![\rho]\!]$, it holds that $\mathsf{E} \cong_{v_I,\rho} \mathsf{E}$.

The above notion will be extensively used in the following two sections.

6 Non-extensionality of the well-founded model

In this section, we demonstrate that the adaptation of Bezem's technique under the well-founded semantics does not in general preserve extensionality. In particular, we exhibit below a program that has a non-extensional well-founded model.

Example 4
Consider the higher-order program P:

```
s Q ← Q (s Q)
p R ← R
q R ← ∼(w R)
w R ← ∼R
```

https://doi.org/10.1017/S1471068417000357 Published online by Cambridge University Press

where the predicate variable Q is of type $o \rightarrow o$ and the predicate variable R is of type o. Before stating formally the non-extensionality result, certain explanations at an intuitive level are in order. Consider first the predicate p of type $o \rightarrow o$. One can view p as representing the identity relation on truth values, i.e., as the relation $\{(v,v) \mid v \in \{false, 0, true\}\}$. It is not hard to see that the predicate q of type $o \rightarrow o$, represents exactly the same relation. However, the definition of q involves two applications of negation, while p is defined directly (without the use of negation).

Consider now the predicate s of type $(o \rightarrow o) \rightarrow o$ which can take as a parameter either p or q. When s takes p as a parameter, we get the following two clauses (by substituting p for Q and (s p) for R in the above program):

```
s p ← p (s p)
p (s p) ← (s p)
```

A recursive definition of this form assigns to (s p), under the well-founded semantics, the value *false*. Consider on the other hand the case where s takes q as a parameter. Then, by doing analogous substitutions, we get the following three clauses:

```
s q ← q (s q)
q (s q) ← ∼(w (s q))
w (s q) ← ∼(s q)
```

Under the well-founded semantics, (s q) is assigned the value 0. In other words, despite the fact that p and q are extensionally equal (see also below), (s p) and (s q) have different truth values. In conclusion, the adaptation of the well-founded semantics under Bezem's technique does not lead to an extensional model in all cases.

Of course, the above discussion is based on intuitive arguments, but it is not hard to formalize it. The main difficulty lies in establishing that p and q are extensionally equal because the above program has an infinite ground instantiation Gr(P) (see Appendix B in the supplementary material). The following lemma, whose detailed proof is given in Appendix B, suggests that $\mathscr{M}_{\mathtt{P}}$, i.e., the Herbrand interpretation of our example program P defined by using the well-founded model $M_{\mathsf{Gr(P)}}$ of Gr(P) as the valuation function, is not extensional.

Lemma 1

The Herbrand interpretation $\mathscr{M}_{\mathtt{P}}$ of the program of Example 4 is not extensional.

The consequences that the above lemma has for the investigation of alternative extensional three-valued semantics for higher-order logic programs with negation will be discussed in Section 8.

A natural question that arises is whether there exists a broad and useful class of programs that are extensional under the well-founded semantics. The next section answers exactly this question.

https://doi.org/10.1017/S1471068417000357 Published online by Cambridge University Press

7 Extensionality of stratified programs

In this section, we present the notion of *stratified* higher-order programs, originally introduced by Rondogiannis and Symeonidou (2016), and argue that the well-founded model of such a program enjoys the extensionality property defined in Section 5. In the following definition, a predicate type π is understood to be *greater than* a second predicate type π', if π is of the form $\rho_1 \rightarrow \cdots \rightarrow \rho_n \rightarrow \pi'$, where $n \geqslant 1$.

Definition 19
A program P is called *stratified* if and only if it is possible to decompose the set of all predicate constants that appear in P into a finite number r of disjoint sets (called *strata*) $S_1, S_2, \ldots, S_r$, such that for every clause $\mathsf{H} \leftarrow \mathsf{A}_1, \ldots, \mathsf{A}_m, \sim\mathsf{B}_1, \ldots, \sim\mathsf{B}_n$ in P, where the predicate constant of H is p, we have the following:

(1) for every $i \leqslant m$, if A_i is a term that starts with a predicate constant q, then $\mathit{stratum}(\mathsf{q}) \leqslant \mathit{stratum}(\mathsf{p})$;
(2) for every $i \leqslant m$, if A_i is a term that starts with a predicate variable Q, then for all predicate constants q that appear in P such that the type of q is greater than or equal to the type of Q, it holds $\mathit{stratum}(\mathsf{q}) \leqslant \mathit{stratum}(\mathsf{p})$;
(3) for every $i \leqslant n$, if B_i starts with a predicate constant q, then $\mathit{stratum}(\mathsf{q}) < \mathit{stratum}(\mathsf{p})$;
(4) for every $i \leqslant n$, if B_i starts with a predicate variable Q, then for all predicate constants q that appear in P such that the type of q is greater than or equal to the type of Q, it holds $\mathit{stratum}(\mathsf{q}) < \mathit{stratum}(\mathsf{p})$;

where $\mathit{stratum}(\mathsf{r}) = i$ if the predicate constant r belongs to S_i.

One may easily see that the stratification for classical logic programs (Apt *et al.* 1988; Gelder 1989) is a special case of the above definition.

Example 5
It is straightforward to see that the program

```
p Q ← ∼(Q a)
q X ← (X≈a)
```

is stratified. However, it can easily be checked that the program

```
p Q ← ∼(Q a)
q X Y ← (X≈a), (Y≈a), p (q a)
```

is not stratified because if the term `(q a)` is substituted for `Q`, we get a circularity through negation. Notice that the type of `q` is $\iota \rightarrow \iota \rightarrow o$ and it is greater than the type of `Q` which is $\iota \rightarrow o$.

As it turns out, stratified higher-order logic programs have an extensional well-founded model. The proof of the following theorem can be found in Appendix C in the supplementary material accompanying the paper at the TPLP archive.

Theorem 2
The well-founded model $\mathscr{M}_{\mathsf{P}}$ of a stratified program P is extensional.

https://doi.org/10.1017/S1471068417000357 Published online by Cambridge University Press

Despite the fact that stratified programs lead to an extensional well-founded model, we have not been able to verify that the same property holds for *locally stratified higher-order logic programs* [for a formal definition of the notion of local stratification for $\mathscr{H}$ programs, please see Rondogiannis and Symeonidou (2016)]. On the other hand, our attempts to find a locally stratified program with a non-extensional well-founded model have also been unsuccessful, and therefore it is not at present clear to us whether this class of programs is well-behaved with respect to extensionality, or not.

8 The restrictions of three-valued approaches

In this section, we re-examine the counterexample of Section 6 but now from a broader perspective. In particular, we indicate that in order to achieve an extensional three-valued semantics for higher-order logic programs with negation, one has to make some (arguably) non-standard assumptions regarding the behaviour of negation in such programs. On the other hand, a logic with an infinite number of truth values appears to be a more appropriate vehicle for achieving extensionality. In the following discussion, we assume some basic familiarity with the main intuition behind the approaches described by Rondogiannis and Symeonidou (2016) and by Charalambidis *et al.* (2014).

Consider again the program of Section 6. Under the infinite-valued adaptation of Bezem's approach given by Rondogiannis and Symeonidou (2016) and also under the domain-theoretic infinite-valued approach by Charalambidis *et al.* (2014), the semantics of that program *is* extensional. The reason is that both of these approaches differentiate the meaning of `p` from the meaning of `q`. The truth domain in both approaches is the set

$$V = \{F_\alpha \mid \alpha < \Omega\} \cup \{0\} \cup \{T_\alpha \mid \alpha < \Omega\}$$

where F_α and T_α represent different degrees of truth and falsity, and Ω is the first uncountable ordinal. Under this truth domain, predicate `p` (intuitively) corresponds to the infinite-valued relation:

$$p = \{(v, v) \mid v \in V\}$$

while predicate `q` corresponds to the relation

$$q = \{(F_\alpha, F_{\alpha+2}) \mid \alpha < \Omega\} \cup \{(0, 0)\} \cup \{(T_\alpha, T_{\alpha+2}) \mid \alpha < \Omega\}$$

Obviously, the relations p and q are different as sets, and therefore it is not a surprise that under both the semantics of Rondogiannis and Symeonidou (2016) and Charalambidis *et al.* (2014), the atoms `(s p)` and `(s q)` have different truth values. Notice, however, that if we collapse p and q in three-valued logic (i.e., if we map each F_α to *false*, each T_α to *true*, and 0 to 0), the collapsed relations become identical.

Assume now that want to devise an (alternative to the one presented in this paper) extensional three-valued semantics for $\mathscr{H}$ programs. Under such a semantics,

https://doi.org/10.1017/S1471068417000357 Published online by Cambridge University Press

it seems reasonable to assume that `p` and `q` would correspond to the same three-valued relation, namely $\{(v, v) \mid v \in \{false, 0, true\}\}$. Notice, however, that from a logic programming perspective, `p` and `q` are expected to have a different operational behaviour when they appear inside a program. In particular, given the program

```
s Q ← Q (s Q)
p R ← R
```

we expect the atom `(s p)` to have the value *false* (due to the circularity that occurs if we try to evaluate it), while given the program

```
s Q ← Q (s Q)
q R ← ∼(w R)
w R ← ∼R
```

we expect the atom `(s q)` to have the value 0 due to the circularity through negation. At first sight, the above discussion seems to suggest that there is no way we can have a three-valued extensional semantics for all higher-order logic programs with negation.

However, the above discussion is based mainly on our experience regarding the behaviour of first-order logic programs with negation. One could argue that we could devise a semantics under which `(s q)` will also return the value *false*. One possible justification for such a semantics would be that the definition of `q` uses two negations which cancel each other, and therefore we should actually expect `q` to behave exactly like `p` when it appears inside a program. Despite the fact that such a proposal seems somewhat unintuitive to us, we cannot exclude it as a possibility. It is worth noting that such cancellations of double negations appear in certain semantic approaches to negation. For example, for certain extended propositional programs, the semantics based on approximation fixpoint theory has the effect of cancelling double negations [see, for example, Denecker *et al.* (2012, page 185, Example 1)]. It is possible that higher-order logic programs with negation behave similarly to extended propositional programs, and it is conceivable that one could construct an extensional three-valued semantics for all higher-order logic programs with negation, using an approach based on approximation fixpoint theory. This research direction certainly deserves further investigation.

It is our belief, however, that the most rigorous approach to extensionality for higher-order logic programs with negation is through the use of the infinite-valued approach. It is worth noting that recently some advantages of the infinite-valued approach versus the well-founded one were identified in a different context. More specifically, as it was recently demonstrated by Ésik (2015) and Carayol and Ésik (2016), the infinite-valued approach satisfies all identities of *iteration theories* (Bloom and Ésik 1993), while the well-founded semantics does not. Since iteration theories (intuitively) provide an abstract framework for the evaluation of the merits of various semantic approaches for languages that involve recursion, the results just mentioned give an extra incentive for the further study and use of the infinite-valued approach.

https://doi.org/10.1017/S1471068417000357 Published online by Cambridge University Press

Supplementary materials

To view supplementary material for this article, please visit https://doi.org/10.1017/S1471068417000357

References

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

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

Bezem, M. 2001. An improved extensionality criterion for higher-order logic programs. In *Proc. of Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL,* Paris, France, September 10–13, 2001, L. Fribourg, Ed. Lecture Notes in Computer Science, vol. 2142. Springer, 203–216.

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 *Proc. of Principles of Knowledge Representation and Reasoning: Proceedings of the 15th 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. *Theory and Practice of Logic Programming 14,* 4–5, 725–737.

Charalambidis, A., Handjopoulos, K., Rondogiannis, P. and Wadge, W. W. 2013. Extensional higher-order logic programming. *ACM Transaction on Computational Logic 14,* 3, 21.

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

Chen, W., Kifer, M. and Warren, D. S. 1993. HILOG: A foundation for higher-order logic programming. *Journal of Logic Programming 15,* 3, 187–230.

Chomicki, J. 2003. Preference formulas in relational queries. *ACM Transactions on Database Systems 28,* 4, 427–466.

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.

Ésik, Z. 2015. Equational properties of stratified least fixed points (extended abstract). In *Proc. of Logic, Language, Information, and Computation - 22nd International Workshop, WoLLIC 2015,* Bloomington, IN, USA, July 20–23, 2015, 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.

Gelder, A. V. 1989. Negation as failure using tight derivations for general logic programs. *Journal of Logic Programming 6,* 1&2, 109–133.

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

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proc. of the 5th International Conference and Symposium Logic Programming,* Seattle,

Washington, August 15–19, 1988 (2 Volumes), R. A. Kowalski and K. A. Bowen, Eds. MIT Press, 1070–1080.

Kountouriotis, V., Rondogiannis, P. and Wadge, W. W. 2005. Extensional higher-order datalog. In *Short Paper Proc. of the 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, 1–5.

Lloyd, J. W. 1987. *Foundations of Logic Programming*. Springer Verlag.

Miller, D. and Nadathur, G. 2012. *Programming with Higher-Order Logic*, 1st ed. Cambridge University Press, New York, NY, USA.

Przymusinska, H. and Przymusinski, T. 1990. Semantic issues in deductive databases and logic programs. In *Formal Techniques in Artificial Intelligence*. North-Holland, 321–367.

Przymusinski, T. C. 1988. On the declarative semantics of deductive databases and logic programs. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 193–216.

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

Rondogiannis, P. and Symeonidou, I. 2017. Extensional semantics for higher-order logic programs with negation. *CoRR abs/1701.08622.*

Rondogiannis, P. and Wadge, W. W. 2005. Minimum model semantics for logic programs with negation-as-failure. *ACM Transactions on Computational Logic 6,* 2, 441–467.

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