Semantics of Negation in Extensional Higher-Order Logic Programming[⋆]

Ioanna Symeonidou

Department of Informatics and Telecommunications
National and Kapodistrian University of Athens
isymeonidou@di.uoa.gr

Abstract. There exist two different extensional approaches to the semantics of positive higher-order logic programming, introduced by W. W. Wadge and M. Bezem respectively. We show that the two approaches coincide for a broad class of programs, but differ in general. Moreover, we adapt Bezems technique under the well-founded, stable model and infinite-valued semantics and show that only the latter succeeds in retaining extensionality in the general case. We analyse the reasons for the failure of the well-founded adaptation of Bezem's technique, 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. Finally, we define for the first time the notions of stratification and local stratification for higher-order logic programs with negation and prove that every stratified program has a distinguished extensional model, which can be equivalently obtained through the well-founded, stable model or infinite-valued semantics.

1 Intoduction

Out of the many extensions of traditional logic programming that have been proposed over the years, the transition to a higher-order setting has been a particularly intriguing and at the same time controversial potential course. The key characteristic of higher-order logic programming is that (roughly speaking) it allows predicates to be passed as parameters of other predicates.

Recent research [19, 2, 3, 5, 4] has investigated the possibility of providing *extensional* semantics to higher-order logic programming. Extensionality facilitates the use of standard set theory in order to reason about programs, at the price of 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 the ones of [13, 9]: the latter languages have a richer syntax but they are not usually amenable to a standard set-theoretic semantics. For a more detailed discussion of extensionality and its importance for higher-order logic programming, the reader can consult the discussion in Section 2 of [15].

⋆ Dissertation Advisor: Panos Rondogiannis, Professor

Despite the fact that only very few articles have been written regarding extensionality in higher-order logic programming, two main semantic approaches can be identified. The first one (called "Wadge's semantics" in the following) [19, 5] was originally proposed by W. W. Wadge [19] for positive programs and later refined and extended by Charalambidis et al. [5]. It has been developed using domain-theoretic tools and resembles the techniques for assigning denotational semantics to functional languages. The second approach (called "Bezem's semantics" in the following) [2, 3], was proposed by M. Bezem also for positive programs. This approach relies on the syntactic entities that exist in a program and is based on processing the ground instantiation of the program.

A natural question that arises is whether one can still obtain an extensional semantics if negation is added to programs. Wadge's semantics has been extended to apply to programs with negation in two ways. The extension proposed in [4] was built upon the infinite-valued semantics [18], a relatively recent proposal to the semantics of negation in logic programming, defined over a logic with an infinite number of truth values. Very recently, a second extension was proposed in [8], generalizing the well-founded semantics [11]. On the other hand, prior to the work reported in this dissertation, no similar extension of Bezem's semantics had been proposed.

In this dissertation we focus on Bezem's approach and attempt to evaluate it, first in comparison to Wadge's semantics as the sole existing alternative and second with respect to its potential to generalise to programs with negation. Our contributions can be summarized as follows:

- We show that for a very broad class of *positive programs* the approaches of [2, 3] and [19, 5] coincide with respect to ground atoms that involve symbols of the program. On the other hand, we argue that if we consider an extended language, which allows existentially quantified predicate variables in the bodies of program clauses, then the two approaches give different results in general. This result, published in [6, 7], will not be discussed in the present summary of the dissertation.
- We demonstrate that neither the well-founded [11] nor the stable model [12] adaptation of Bezem's technique leads to an extensional model in the general case. The result concerning the stable model semantics was published in [16], while the result concerning the well-founded semantics was stated in [15, 17].
- We study the reasons behind the failure of the well-founded adaptation of Bezem's technique and 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, for example as in the case of [8]. The argument was first presented in [15].
- We demonstrate that by combining the technique of [2, 3] with the infinite-valued semantics of [18], we obtain an extensional semantics for higher-order logic programs with negation. This result was published in [14, 16]. Note that

the infinite-valued semantics was the first approach to negation to enable the extension of the semantics of [19, 5] (see [4]).
- We define the notions of *stratification* and *local stratification* for higher-order logic programs with negation. These notions were first defined in [14]; note that similar notions have not yet been studied under the semantics of [19, 5, 4, 8]. We prove that the stable model, the well-founded and the infinite-valued adaptations of Bezem's technique give equivalent extensional models in the case of *stratified* programs. The extensionality of the well-founded model for stratified programs was shown in [15] and affirmed the importance and the well-behaved nature of stratified programs, which was, until now, only known for the first-order case.

The next two sections motivate in an intuitive way the main ideas behind extending Bezem's semantics in order to apply to higher-order programs with negation. The remaining sections develop the material in a more formal way.

2 An Intuitive Overview of the Proposed Approach

In this section we give an intuitive description of the semantic technique for positive higher-order logic programs proposed by Bezem [2, 3] and we outline how we use it when negation is added to programs. Given a positive program, the starting idea behind Bezem's approach is to take its "ground instantiation", in which we replace variables with well-typed terms constructed from syntactic entities that appear in the program. For example, consider the higher-order program (for the moment, we use ad-hoc Prolog-like syntax):

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

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 9):

```
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 in order to compute the set of atoms that should be taken as "true".

Bezem demonstrated that the least fixed-point semantics of the ground instantiation of every positive higher-order logic program of the language considered in [2,3], 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 if `p(q)` is true then `p(id(q))` is also true, because `q` and `id(q)` should be considered as indistinguishable.

We use the same idea with programs that include negation: the ground instantiation of such a program can be seen as a (possibly infinite) propositional program with negation. Therefore, we can compute its semantics in any standard way that exists for obtaining the meaning of such programs and then proceed to examine whether the chosen model is extensional in the sense of Bezem [2,3]. As we are going to see in the subsequent sections, when the infinite valued model of the ground instantiation of the program is chosen for this purpose, the semantics we obtain is indeed extensional, but the same does not hold for the well-founded model or the stable model(s).

3 Infinite-valued Semantics

In this section we discuss the motivation behind the infinite-valued semantics [18]. The key idea of this approach is that in order to give a logical semantics to negation-as-failure and to distinguish it from ordinary negation, one needs to extend the domain of truth values. For example, consider the program:

```
p ←
r ← ∼p
s ← ∼q
t ← ∼t
```

According to negation-as-failure, both `p` and `s` receive the value *true*. However, `p` seems "truer" than `s` because there is a clause which says so, whereas `s` is true only because we are never obliged to make `q` true. In a sense, `s` is true only by default. For this reason, it was proposed in [18] to introduce a "default" truth value T_1 just below the "real" true T_0, and (by symmetry) a weaker false value F_1 just above ("not as false as") the real false F_0. Then, negation-as-failure is a combination of ordinary negation with a weakening. Thus $\sim F_0 = T_1$ and $\sim T_0 = F_1$. Since negations can be iterated, the new truth domain has a sequence $\ldots, T_3, T_2, T_1$ of weaker and weaker truth values below T_0 but above a neutral value 0; and a mirror image sequence $F_1, F_2, F_3, \ldots$ above F_0 and below 0. Since our propositional programs are possibly countably infinite, we need a T_α and a F_α for every countable ordinal α. The intermediate truth value 0 is needed for certain atoms that have a "pathological" negative dependence on themselves (such as `t` in the above program). In conclusion, our truth domain $\mathbb{V}^\infty$ is shaped as follows:

$$F_0 < F_1 < \cdots < F_\omega < \cdots < F_\alpha < \cdots < 0 < \cdots < T_\alpha < \cdots < T_\omega < \cdots < T_1 < T_0$$

It is shown in [18] that every first-order logic program has a unique minimum infinite-valued model, under an ordering relation $\sqsubseteq$. For example, the minimum infinite-valued model of the program presented above may be described as the set $\{(\mathtt{p}, T_0), (\mathtt{q}, F_0), (\mathtt{r}, F_1), (\mathtt{s}, T_1), (\mathtt{t}, 0)\}$.

4 The Syntax of $\mathcal{H}$

In this section we define the syntax of our language $\mathcal{H}$. $\mathcal{H}$ uses 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:*

$$\begin{aligned}\sigma &:= \iota \mid (\iota \to \sigma)\\ \pi &:= o \mid (\rho \to \pi)\\ \rho &:= \iota \mid \pi\end{aligned}$$

We will use τ to denote an arbitrary type. The binary operator $\to$ is right-associative. It can be easily seen that every predicate type π can be written in the form $\rho_1 \to \cdots \to \rho_n \to o$, $n \geq 0$ (for $n = 0$ we assume that $\pi = o$).

Definition 2. *The* alphabet *of $\mathcal{H}$ 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; and the left and right parentheses.*

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

Definition 3. *The set of* terms *of $\mathcal{H}$ is defined as follows: every predicate variable (resp., predicate constant) of type π is a term of type π; every individual variable (resp., 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.*

Definition 4. *The set of* expressions *of $\mathcal{H}$ 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.*

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* and expressions of type o will be referred to as *literals*.

Definition 5. *A* clause *of $\mathcal{H}$ is a formula* $\mathsf{p}\ \mathsf{E}_1 \cdots \mathsf{E}_n \leftarrow \mathsf{L}_1, \ldots, \mathsf{L}_m$*, where* p *is a predicate constant of type* $\rho_1 \rightarrow \cdots \rightarrow \rho_n \rightarrow o$*,* $\mathsf{E}_1, \ldots, \mathsf{E}_n$ *are terms of types* $\rho_1, \ldots, \rho_n$ *respectively, so that all* E_i *with* $\rho_i \neq \iota$ *are distinct variables, and* $\mathsf{L}_1, \ldots, \mathsf{L}_m$ *are literals. The term* $\mathsf{p}\ \mathsf{E}_1 \cdots \mathsf{E}_n$ *is called the* head *of the clause and the conjunction* $\mathsf{L}_1, \ldots, \mathsf{L}_m$ *is its* body*. A* program P *of $\mathcal{H}$ is a finite set of clauses.*

Example 1. The program below defines the `subset` relation over unary predicates:

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

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*. The domain* $\{\mathsf{V}_1, \ldots, \mathsf{V}_n\}$ *of* θ *is denoted by* $dom(\theta)$*. 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* $\mathsf{V}\theta = \mathsf{V}$*;* $(\mathsf{f}\ \mathsf{E}_1 \cdots \mathsf{E}_n)\theta = (\mathsf{f}\ \mathsf{E}_1\theta \cdots \mathsf{E}_n\theta)$*;* $(\mathsf{E}_1\ \mathsf{E}_2)\theta = (\mathsf{E}_1\theta\ \mathsf{E}_2\theta)$*;* $(\sim\mathsf{E})\theta = (\sim\mathsf{E}\theta)$*. If* θ *is a ground substitution with* $vars(\mathsf{E}) \subseteq dom(\theta)$*, then the ground expression* $\mathsf{E}\theta$ *is called a* ground instance *of* E*.*

Definition 8. *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.*

Definition 9. *Let* P *be a program. A* ground instance of a clause $\mathsf{p}\ \mathsf{E}_1 \cdots \mathsf{E}_n \leftarrow \mathsf{L}_1, \ldots, \mathsf{L}_m$ *of* P *is a formula* $(\mathsf{p}\ \mathsf{E}_1 \cdots \mathsf{E}_n)\theta \leftarrow \mathsf{L}_1\theta, \ldots, \mathsf{L}_m\theta$*, 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* $\mathsf{V} : \rho$*,* $\theta(\mathsf{V}) \in U_{\mathsf{P},\rho}$*. 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 $\mathcal{H}$

In [2, 3] M. Bezem developed a semantics for higher-order logic programs that is a generalization of the familiar Herbrand-model semantics of classical (first-order) logic programs. As such, the approach proposes that essentially predicates are understood as mapping tuples of syntactic objects to truth values. Because of this, the following simplified definition of a higher-order interpretation is possible:

Definition 10. *A* (higher-order) Herbrand interpretation *I of a program P is a function which assigns to each ground atom of $U_{\mathsf{P},o}$, and to the negation thereof, an element in a specified domain of truth values.*

The truth domain used in [2, 3] is the traditional two-valued one, as only positive programs are studied. In this dissertation we also consider Herbrand interpretations with a three-valued truth domain, i.e. $\{\mathit{false}, 0, \mathit{true}\}$, as well as interpretations with an infinite-valued truth domain, i.e. $\mathbb{V}^\infty$.

The concept of "Herbrand model" of a higher-order program can be defined as in classical logic programming.

Definition 11. *Let P be a program and I be a Herbrand interpretation of P. We say I is a* model *of P if $I(\mathsf{A}) \geq \mathit{min}\{I(\mathsf{L}_1), \ldots, I(\mathsf{L}_n)\}$ holds for every ground instance $\mathsf{A} \leftarrow \mathsf{L}_1, \ldots, \mathsf{L}_m$ of a clause of P.*

Bezem's semantics is based on the observation that, given a positive higher-order program, the minimum model of its ground instantiation serves as a Herbrand interpretation for the program itself. We follow the same idea for programs with negation: we can use as an interpretation of a given higher-order program P, the model defined by any semantics that applies to its ground instantiation. It is trivial to see that any such model is also a Herbrand model of P.

In the following sections we focus on the well-founded [11], stable [12] and infinite-valued [18] models. We investigate if these models enjoy the extensionality property, formally defined by Bezem [2, 3] through relations $\cong_{I,\rho}$ over the set of ground expressions of a given type ρ and under a given interpretation I. These relations intuitively express extensional equality of type ρ, in the sense discussed in Section 2. The formal definition is as follows:

Definition 12. *Let I be a Herbrand interpretation for a given program P. For every argument type ρ we define the relations $\cong_{I,\rho}$ on $U_{\mathsf{P},\rho}$ as follows. Let $\mathsf{E}, \mathsf{E}' \in U_{\mathsf{P},\rho}$; then $\mathsf{E} \cong_{I,\rho} \mathsf{E}'$ if and only if: $\rho = \iota$ and $\mathsf{E} = \mathsf{E}'$; or $\rho = o$ and $I(\mathsf{E}) = I(\mathsf{E}')$; or $\rho = \rho' \to \pi$ and $\mathsf{E}\,\mathsf{D} \cong_{I,\pi} \mathsf{E}'\,\mathsf{D}'$ for all $\mathsf{D}, \mathsf{D}' \in U_{\mathsf{P},\rho'}$, such that $\mathsf{D} \cong_{I,\rho'} \mathsf{D}'$.*

Generally, such relations are symmetric and transitive [2, 3] (partial equivalences). Whether they are moreover reflexive (full equivalences), depends on the specific interpretation, which leads to the notion of *extensional interpretation*:

Definition 13. *Let P be a program and let I be a Herbrand interpretation of P. We say I is* extensional *if for all argument types ρ, $\cong_{I,\rho}$ is reflexive, i.e. for all $\mathsf{E} \in U_{\mathsf{P},\rho}$, $\mathsf{E} \cong_{I,\rho} \mathsf{E}$.*

6 Extensionality Study of Selected Models

In this section we demonstrate that the adaptation of Bezem's technique under the the two main approaches to negation known from first-order logic programming, i.e. the stable model semantics and the well-founded semantics, do not in general preserve extensionality. On the other hand, we indicate that the infinite-valued semantics can lead to an extensional semantics for the programs of our higher-order language $\mathcal{H}$.

We begin by showing that the ground instantiation of a $\mathcal{H}$ program may not always have extensional stable models.

Example 2. Consider the program:

```
r Q ← ∼(s Q), ∼(r p)
s Q ← ∼(r Q), ∼(s q)
q a ←
p a ←
```

where, in the first two clauses, `Q` is of type $\iota \rightarrow o$. By examining the ground instantiation of the above program, one can see that it has the non-extensional stable model {`(p a)`, `(q a)`, `(s p)`, `(r q)`}. However, it has *no* extensional stable models: there are four extensional interpretations that are potential candidates, namely M_1 = {`(p a)`,`(q a)`}, M_2 = {`(p a)`,`(q a)`,`(r p)`,`(r q)`}, M_3 = {`(p a)`, `(q a)`, `(s p)`, `(s q)`}, and M_4 = {`(p a)`,`(q a)`,`(s p)`,`(s q)`, `(r p)`, `(r q)`}; but none of these models is a stable model of the program. $\square$

The above examples seem to suggest that the extensional approach of [2, 3] is incompatible with the stable model semantics. Unfortunately, the same holds for the well-founded semantics.

Example 3. Consider the higher-order program `P`:

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

where the predicate variable `Q` is of type $o \rightarrow o$ and the predicate variable `R` is of type o. It is not hard to see that the predicates `p` $: o \rightarrow o$ and `q` $: o \rightarrow o$ represent the same relation, namely $\{(v, v) \mid v \in \{\mathit{false}, 0, \mathit{true}\}\}$.

Consider the predicate `s` $: (o \rightarrow o) \rightarrow o$. By taking the ground instances of the clauses involved, it is easy to see that the atom `(s p)`, under the well-founded semantics, will be assigned the value *false*. On the other hand, `(s q)` is assigned the value 0, under the well-founded semantics, since the ground instances of the relevant clauses form a circular definition involving negation. In other words, `p` and `q` are extensionally equal, but `(s p)` and `(s q)` have different truth values.

The above discussion is based on intuitive arguments, but it is not hard to formalize it and obtain the following lemma:

Lemma 1. *The well-founded model* $\mathcal{M}_{\mathsf{P}}$ *of the program of Example 3, is not extensional.*

In light of the above negative results, the next theorem suggests that the infinite-valued model adaptation of Bezem's technique is a more suitable candidate for capturing the extensional semantics of general $\mathcal{H}$ programs.

Theorem 1. *The infinite-valued model of every program of $\mathcal{H}$ is extensional.*

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

7 Extensionality of Stratified Programs

In this section we argue that the well-founded model of a *stratified* higher-order program [16] enjoys the extensionality property. 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 $\rho_1 \to \cdots \to \rho_n \to \pi'$, where $n \geq 1$.

Definition 14. *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:*

1. *for every* $i \leq m$*, if* A_i *starts with a predicate constant* q*, then* $stratum(\mathsf{q}) \leq stratum(\mathsf{p})$*;*
2. *for every* $i \leq m$*, if* A_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* $stratum(\mathsf{q}) \leq stratum(\mathsf{p})$*;*
3. *for every* $i \leq n$*, if* B_i *starts with a predicate constant* q*, then* $stratum(\mathsf{q}) < stratum(\mathsf{p})$*;*
4. *for every* $i \leq n$*, 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* $stratum(\mathsf{q}) < stratum(\mathsf{p})$*;*

where $stratum(\mathsf{r}) = i$ *if* r *belongs to* S_i*.*

Evidently, the stratification for classical logic programs [1] is a special case of the above definition.

Example 4. It is straightforward to see that the program:

```
p Q ← ∼(Q a)
q a ←
```

is stratified. However, it is easy to check that the program:

```
p Q ← ∼(Q a)
q a a ← p (q a)
```

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

As it turns out, stratified higher-order logic programs have an extensional two-valued well-founded model.

Theorem 2. *The well-founded model* $\mathcal{M}_{\mathsf{P}}$ *of a stratified program* P *is extensional and does not assign the value* 0*.*

8 The Restrictions of 3-Valued Approaches

In this section we re-examine the counterexample of Example 3 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 strong assumptions regarding the behaviour of negation in such programs.

Under the infinite-valued adaptation of Bezem's approach and also under the domain-theoretic infinite-valued approach of [4], 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`. Under the truth domain in both approaches, i.e. $\mathbb{V}^\infty$, predicate `p` corresponds to the infinite-valued relation: $p = \{(v, v) \mid v \in \mathbb{V}^\infty\}$ while predicate `q` corresponds to the relation: $q = \{(F_\alpha, F_{\alpha+2}) \mid \alpha < \Omega\} \cup \{(0,0)\} \cup \{(T_\alpha, T_{\alpha+2}) \mid \alpha < \Omega\}$ where Ω is the first uncountable ordinal. Obviously, the relations p and q are different as sets and therefore it is not a surprise that under both the infinite-valued adaptation of Bezem's semantics presented in this dissertation and the semantics of [4], the atoms `(s p)` and `(s q)` have different truth values.

Assume now that we want to devise an (alternative to the well-founded extension of Bezem's semantics presented in this dissertation) extensional three-valued semantics for $\mathcal{H}$ programs. Under such a semantics, it seems reasonable to assume that `p` and `q` would correspond to the same three-valued relation, namely $\{(v, v) \mid v \in \{\mathit{false}, 0, \mathit{true}\}\}$. Notice however that `p` and `q` are expected to have a different *operational* behaviour. 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 when 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 a three-valued extensional semantics for all higher-order logic programs with negation is not possible.

However, the above discussion is based mainly on our experience regarding the behaviour of first-order logic programs with negation. One could advocate a semantics under which `(s q)` will also return the value *false*, arguing that the definition of `q` uses two negations which cancel each other. This cancellation of double negations is not an entirely new idea; for example, for certain extended propositional programs, the semantics based on approximation fixpoint theory has the same effect (see for example Denecker et al. [10][page 185, Example 1]). We have recently developed such an extensional three-valued semantics for higher-order logic programs with negation, using an approach based on approximation fixpoint theory in [8].

9 Conclusions and Future Work

We have for the first time adapted Bezem's technique to define an extensional semantics for higher-order programs with negation, achieved through the infinite-valued approach [18]. On the other hand, we have shown that an adaptation of the technique under the well-founded or the stable model semantics does not in general lead to an extensional semantics. Finally, we have defined the notions of stratification and local stratification and proven that the class of stratified programs is a notable exception to the previous negative result.

It poses an interesting open question whether the class of *locally stratified higher-order logic programs* (see [16]) is well-behaved with respect to extensionality, or not. Another matter worth looking into is the relationships between the infinite-valued extension of Bezem's semantics presented in this dissertation and its domain theoretic counterpart. The most intriguing question, perhaps, is the comparative evaluation of the infinite-valued extensions of Bezem's semantics and the domain theoretic semantics against the three-valued domain theoretic semantics of [8].

References

1. Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a theory of declarative knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1988.
2. Marc Bezem. Extensionality of simply typed logic programs. In Danny De Schreye, editor, *Logic Programming: The 1999 International Conference, Las Cruces, New Mexico, USA, November 29 - December 4, 1999*, pages 395–410. MIT Press, 1999.
3. Marc Bezem. An improved extensionality criterion for higher-order logic programs. In Laurent Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 203–216. Springer, 2001.
4. Angelos Charalambidis, Zoltán Ésik, and Panos Rondogiannis. Minimum model semantics for extensional higher-order logic programming with negation. *TPLP*, 14(4-5):725–737, 2014.
5. Angelos Charalambidis, Konstantinos Handjopoulos, Panos Rondogiannis, and William W. Wadge. Extensional higher-order logic programming. *ACM Trans. Comput. Log.*, 14(3):21, 2013.
6. Angelos Charalambidis, Panos Rondogiannis, and Ioanna Symeonidou. Equivalence of two fixed-point semantics for definitional higher-order logic programs. In Ralph Matthes and Matteo Mio, editors, *Proceedings 10th International Workshop on Fixed Points in Computer Science, (FICS 2015), Berlin, Germany, September 11-12, 2015*, volume 191 of *EPTCS*, pages 18–32, 2015.

7. Angelos Charalambidis, Panos Rondogiannis, and Ioanna Symeonidou. Equivalence of two fixed-point semantics for definitional higher-order logic programs (extended version of conference paper). *Theoretical Computer Science*, 668:27–42, 2017.
8. Angelos Charalambidis, Panos Rondogiannis, and Ioanna Symeonidou. Approximation fixpoint theory and the well-founded semantics of higher-order logic programs (*in press*). *Theory and Practice of Logic Programming*, arXiv:1804.08335, 2018, 2018. (To be presented at the 34th International Conference on Logic Programming (ICLP 2018), Oxford, UK, July 14 - 17 2018).
9. Weidong Chen, Michael Kifer, and David Scott Warren. Hilog: A foundation for higher-order logic programming. *The Journal of Logic Programming*, 15(3):187 – 230, 1993.
10. Marc Denecker, Maurice Bruynooghe, and Joost Vennekens. Approximation fixpoint theory and the semantics of logic and answers set programs. In Esra Erdem, Joohyung Lee, Yuliya Lierler, and David Pearce, editors, *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, volume 7265 of *Lecture Notes in Computer Science*, pages 178–194. Springer, 2012.
11. Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, 1991.
12. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes)*, pages 1070–1080. MIT Press, 1988.
13. Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, New York, NY, USA, 1st edition, 2012.
14. Panos Rondogiannis and Ioanna Symeonidou. Extensional semantics for higher-order logic programs with negation. In Loizos Michael and Antonis C. Kakas, editors, *Logics in Artificial Intelligence - 15th European Conference (JELIA 2016), Larnaca, Cyprus, November 9-11, 2016, Proceedings*, volume 10021 of *Lecture Notes in Computer Science*, pages 447–462, 2016.
15. Panos Rondogiannis and Ioanna Symeonidou. The intricacies of three-valued extensional semantics for higher-order logic programs. *Theory and Practice of Logic Programming*, 17(5-6):974–991, 2017. (Presented at the 33rd International Conference on Logic Programming (ICLP 2017), Melbourne, Australia, August 28 - September 1 2017. **Best Paper Award**).
16. Panos Rondogiannis and Ioanna Symeonidou. Extensional Semantics for Higher-Order Logic Programs with Negation (Extended version of conference paper). *Logical Methods in Computer Science*, Volume 14, Issue 2, July 2018.
17. Panos Rondogiannis and Ioanna Symeonidou. The intricacies of three-valued extensional semantics for higher-order logic programs (*in press*). In *27th International Joint Conference on Artificial Intelligence (IJCAI 2018) ("Best Sister Conferences" track), Stockholm, Sweden, July 13-19, 2018, Proceedings*, 2018.
18. Panos Rondogiannis and William W. Wadge. Minimum model semantics for logic programs with negation-as-failure. *ACM Trans. Comput. Log.*, 6(2):441–467, 2005.
19. William W. Wadge. Higher-order horn logic programming. In Vijay A. Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, Oct. 28 - Nov 1, 1991*, pages 289–303. MIT Press, 1991.