Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)

The Intricacies of Three-Valued Extensional Semantics for Higher-Order Logic Programs*

Panos Rondogiannis, Ioanna Symeonidou
National and Kapodistrian University of Athens
prondo@di.uoa.gr, i.symeonidou@di.uoa.gr

Abstract

In this paper we examine the problem of providing a purely extensional three-valued semantics for higher-order logic programs with negation. We demonstrate that a technique that was proposed by M. Bezem for providing extensional semantics to positive higher-order logic programs, *fails* when applied to higher-order logic programs with negation. On the positive side, we demonstrate that for stratified higher-order logic programs, extensionality is indeed achieved by the technique. We analyze the reasons of the failure of extensionality in the general case, arguing that a three-valued setting can not 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.

1 Introduction

Recent research [Wadge, 1991; Bezem, 1999; 2001; Charalambidis *et al.*, 2013; 2017; 2018; Rondogiannis and Symeonidou, 2018] has investigated the possibility of providing *extensional* semantics to higher-order logic programming (HOLP). Extensionality facilitates the use of standard set theory in order to reason about programs, at the price of a relatively restricted syntax.

There exist two research directions for providing extensional semantics to higher-order logic programs. The first one [Wadge, 1991; Charalambidis *et al.*, 2013; 2014; 2018] has been developed using domain-theoretic tools. The second approach [Bezem, 1999; 2001; Rondogiannis and Symeonidou, 2018] is based on processing the ground instantiation of the program. The two research directions are not unrelated: it has been shown by Charalambidis et al. [2017] that for a broad class of positive programs, the two approaches coincide.

In this paper we focus on the second approach, initially proposed by Bezem [1999; 2001] for *positive* higher-order logic programs. Recently, it was demonstrated by Rondogiannis and Symeonidou [2018] that by combining the technique of Bezem with the infinite-valued semantics of Rondogiannis and Wadge [2005], we obtain an extensional semantics for higher-order logic programs with negation. At the same time, a negative result was also established: by combining the technique of Bezem 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 [2018] whether the combination of the technique of Bezem 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, the main contributions of which 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. This indicates that the addition of negation to higher-order logic programming is not such a straightforward task as it was possibly initially anticipated.
- 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 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 strong assumptions regarding the behaviour of negation in higher-order logic programs.

The next two sections motivate in an intuitive way the proposed approach. The remaining sections develop the material in a formal way. The proofs of all results can be found in the original paper [Rondogiannis and Symeonidou, 2017].

2 Extensional HOLP

W. W. Wadge [1991] suggested that if we appropriately restrict the syntax of HOLP, then we can obtain a denotational semantics in which predicates denote sets, much like the traditional semantics of higher-order functional programming. The most crucial syntactic restriction imposed by Wadge (and later independently by M. Bezem [1999; 2001]), is the following:

*This paper is an abridged version of a paper with the same tile, that won a best-paper award at the ICLP-2017 conference.

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, each of these clauses violates Wadge's restriction:

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

Under the extensional approach, 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` it is *guaranteed* that any higher-order predicate 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".

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 Introduction

In this section we give an intuitive description of the semantic technique for positive higher-order logic programs proposed by Bezem [1999; 2001] 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:

```
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 [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 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 [1999; 2001]. As we are going to see in the subsequent sections, when the ground instantiation of the program is interpreted under the well-founded semantics, the semantics we obtain is not always extensional.

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:*

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

We will use τ to denote an arbitrary type. 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* $\rho \rightarrow \pi$ *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 2 *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 [Bezem, 1999; 2001] 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 the meaning of predicates and data objects is fixed across all (Herbrand) interpretations. 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* $U_{\mathsf{P},o}$*, an element in a specified domain of truth values.*

The truth domain used in [Bezem, 1999; 2001] is the traditional two-valued one, as only positive programs are studied. In our attempt to extend the well-founded semantics, in this paper we consider Herbrand interpretations with a three-valued truth domain, i.e. $\{false, 0, true\}$.

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* $I(\mathsf{A}) \geq min\{I(\mathsf{L}_1), \ldots, I(\mathsf{L}_n)\}$ *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 semantic approach 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 investigate if the well-founded model [Gelder *et al.*, 1991] enjoys the extensionality property, formally defined by Bezem [1999; 2001] 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 3. 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* $\rho = \rho' \rightarrow \pi$ *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 [Bezem, 1999; 2001] (i.e., partial equivalences). Whether they are moreover reflexive (i.e., 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* $\mathsf{E} \in U_{\mathsf{P},\rho}$*,* $\mathsf{E} \cong_{I,\rho} \mathsf{E}$*.*

For a more thorough and formal presentation of the notions discussed in this section, the reader may refer to the full version of the present paper [Rondogiannis and Symeonidou, 2017].

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 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* $\{(v, v) \mid v \in \{false, 0, true\}\}$*.*

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.*

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 [Rondogiannis and Symeonidou, 2018] 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 $\rho_1 \rightarrow \cdots \rightarrow \rho_n \rightarrow \pi'$, 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 *is a term that starts with a predicate constant* q*, then stratum*$(\mathsf{q}) \leq$ *stratum*(p)*;*
2. *for every* $i \leq m$*, 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 stratum*$(\mathsf{q}) \leq$ *stratum*(p)*;*
3. *for every* $i \leq n$*, if* B_i *starts with a predicate constant* q*, then stratum*$(\mathsf{q}) <$ *stratum*(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*(p)*;*

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

Evidently, the stratification for classical logic programs [Apt *et al.*, 1988] 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* $\iota \rightarrow \iota \rightarrow o$ *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 two-valued well-founded model.

Theorem 1 *The well-founded model* $\mathcal{M}_{\mathsf{P}}$ *of a stratified program* P *is extensional and does not assign the value* 0*.*

Despite the above result, we have not yet been able to clarify whether the class of *locally stratified higher-order logic programs* (see Rondogiannis and Symeonidou [2018]) is well-behaved with respect to extensionality, or not.

8 The Restrictions of 3-Valued Approaches

In this section 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.

Consider again the program of Section 6. Under the infinite-valued adaptation of Bezem's approach by Rondogiannis and Symeonidou [2018] and also under the domain-theoretic infinite-valued approach by Charalambidis et al. [2014], the semantics of that program *is* extensional. Both approaches differentiate the meaning of `p` from the meaning of `q`, which correspond to different *infinite-valued* relations. Therefore, it is not a surprise that in both cases, the atoms `(s p)` and `(s q)` have different truth values.

Assume now that we want to devise an (alternative to the one presented in this paper) 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 $\{(v, v) \mid v \in \{false, 0, true\}\}$.

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. [2012][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 Charalambidis et al. [2018]. It is an interesting project for future research to evaluate the merits of each approach and the relationships between them.

References

[Apt *et al.*, 1988] 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.

[Bezem, 1999] 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.

[Bezem, 2001] 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.

[Charalambidis *et al.*, 2013] Angelos Charalambidis, Konstantinos Handjopoulos, Panos Rondogiannis, and William W. Wadge. Extensional higher-order logic programming. *ACM Trans. Comput. Log.*, 14(3):21, 2013.

[Charalambidis *et al.*, 2014] 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.

[Charalambidis *et al.*, 2017] Angelos Charalambidis, Panos Rondogiannis, and Ioanna Symeonidou. Equivalence of two fixed-point semantics for definitional higher-order logic programs. *Theor. Comput. Sci.*, 668:27–42, 2017.

[Charalambidis *et al.*, 2018] Angelos Charalambidis, Panos Rondogiannis, and Ioanna Symeonidou. Approximation fixpoint theory and the well-founded semantics of higher-order logic programs (*in press*). 2018.

[Denecker *et al.*, 2012] 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.

[Gelder *et al.*, 1991] 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.

[Gelfond and Lifschitz, 1988] 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.

[Rondogiannis and Symeonidou, 2017] Panos Rondogiannis and Ioanna Symeonidou. The intricacies of three-valued extensional semantics for higher-order logic programs. *TPLP*, 17(5-6):974–991, 2017.

[Rondogiannis and Symeonidou, 2018] Panos Rondogiannis and Ioanna Symeonidou. Extensional semantics for higher-order logic programs with negation (*in press*). *Logical Methods in Computer Science*, abs/1701.08622, 2018.

[Rondogiannis and Wadge, 2005] 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.

[Wadge, 1991] 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.