A "Conservative" Approach to Extending Answer Set Programming with Non-Herbrand Functions

Marcello Balduccini

Kodak Research Laboratories
Eastman Kodak Company
Rochester, NY 14650-2102 USA
marcello.balduccini@gmail.com

Abstract In this paper we propose an extension of Answer Set Programming (ASP) by non-Herbrand functions, i.e. functions over non-Herbrand domains. Introducing support for such functions allows for an economic and natural representation of certain kinds of knowledge that are comparatively cumbersome to represent in ASP. The key difference between our approach and other techniques for the support of non-Herbrand functions is that our extension is more "conservative" from a knowledge representation perspective. In fact, we purposefully designed the new language so that (1) the representation of relations is fully retained; (2) the representation of knowledge using non-Herbrand functions follows in a natural way from the typical ASP strategies; (3) the semantics is an extension of the the semantics of ASP from [9], allowing for a comparatively simple incorporation of various extensions of ASP such as weak constraints, probabilistic constructs and consistency-restoring rules.

1 Introduction

In this paper we describe an extension of Answer Set Programming (ASP) [9,13,2] called ASP$\{$f$\}$, and aimed at simplifying the representation of non-Herbrand functions.

In logic programming, functions are typically interpreted over the Herbrand Universe, with each functional term $f(x)$ mapped to its own canonical syntactical representation. That is, in most logic programming languages, the value of an expression $f(x)$ is $f(x)$ itself, and thus strictly speaking $f(x) = 2$ is false. This type of functions, the corresponding languages and efficient implementation of solvers is the subject of a substantial amount of research (e.g. [7,4,14]).

When representing certain kinds of knowledge, however, it is sometimes convenient to use functions with *non-Herbrand domains* (*non-Herbrand functions* for short), i.e. functions that are interpreted over domains other than the Herbrand Universe. For example, when describing a domain in which people enter and exit a room over time, it may be convenient to represent the number of people in the room at step s by means of a function $occupancy(s)$ and to state the effect of a person entering the room by means of a statement such as

$$occupancy(S + 1) = occupancy(S) + 1$$

where S is a variable ranging over the possible time steps in the evolution of the domain.

Of course, in most logic programming languages, non-Herbrand functions can still be represented, but the corresponding encodings are not as natural and declarative as the one above. For instance, a common approach consists in representing the functions of interest using relations, and then characterizing the functional nature of these relations by writing auxiliary axioms. In ASP, one would encode the above statement by (1) introducing a relation $occupancy'(s, o)$, whose intuitive meaning is that $occupancy'(s, o)$ holds iff the value of $occupancy(s)$ is o; and (2) re-writing the original statement as a rule

$$occupancy'(S + 1, O + 1) \leftarrow occupancy'(S, O). \tag{1}$$

The characterization of the relation as representing a function would be completed by an axiom such as

$$\neg occupancy'(S, O') \leftarrow occupancy'(S, O),\ O \neq O'. \tag{2}$$

which intuitively states that $occupancy(s)$ has a unique value. The disadvantage of this representation is that the functional nature of $occupancy'(s, o)$ is only stated in (2). When reading (1), one is given no indication that $occupancy'(s, o)$ represents a function – and, before finding statements such as (2), one can make no assumption about the functional nature of the relations in a program when a combination of (proper) relations and non-Herbrand functions are present. As a consequence, the declarativity of the rules is penalized.

In comparison to other methods allowing for a direct representation of non-Herbrand functions, we view the language proposed in this paper as an extension of ASP that is "conservative" from a knowledge representation standpoint, and that is achieved by a rather small modification of the original semantics. By conservative we mean that the proposed language not only allows for a representation of non-Herbrand functions that is natural and direct, but also retains the key properties of the underlying language of ASP. In particular, we designed our language so that:

– The ease of representation of ASP is retained;
– It is possible to represent, and reason about, incomplete information regarding both relations and non-Herbrand functions;
– The representation of knowledge regarding non-Herbrand functions follows formalization strategies similar to those used in ASP. For example, the encoding of a default "normally $f = 2$" should be syntactically similar to the encoding of a default "normally p."
– The semantics of the new language is a modular extension of the semantics of ASP as defined e.g. in [9].

The last requirement allows for a comparatively simple incorporation into our language of extensions of ASP, such as weak constraints [5], probabilistic constructs [3] and consistency restoring rules [1]. Although discussing the implementation of the proposed language is outside the scope of the present paper, it is worth noting that the last requirement also opens the door to the implementation of the language within most state-of-the-art ASP solvers.

The other requirements may seem straightforward, but they are in one way or another violated by most approaches to extending ASP with non-Herbrand functions that are found in the literature. Some simple examples of this are presented next, while a more thorough discussion can be found later in the paper.

The existing approaches can be categorized into two groups, depending on whether they allow for partial functions or not. The approaches described in [10], [12], [15] and [8] define languages that deal with total functions, whereas [6] uses partial functions.

One limitation of the approaches that require total functions is that they force one to model lack of knowledge by means of multiple answer sets, whereas in ASP one is free to model lack of knowledge *either* with multiple answer sets (such as $\{p\}$, $\{\neg p\}$) *or* by the lack of the corresponding literals in an answer set (e.g. answer set $\{q\}$ states that q is believed to be true, and that nothing is known about p and $\neg p$). Not only this in itself involves a substantial difference in knowledge representation strategies and limits one's ability to represent and reason about incomplete information, but it also favors the derivation of *unsupported conclusions*: literals that are not in the head of any rule, and yet occur in an answer set. This is a drastic change of direction from ASP, in which supportedness is a fundamental property.

The language of [6] allows for the representation of partial functions, and in this respect allows for an approach to knowledge representation that is closer to that of ASP. That language however does not allow strong negation. The lack of strong negation appears to force the introduction of a special comparison operator $\#$ to express the fact that the two functions being compared are not only different, but also both defined. A further difference is that the semantics of the language of [6] is based on Quantified Equilibrium Logic rather than on [9].

The rest of the paper is organized as follows. The next two sections describe the syntax and the semantics of the proposed language. In the following section we discuss the topic of knowledge representation with non-Herbrand functions, both in ASP$\{$f$\}$ and in other comparable languages from the literature. Next, we discuss the relationship between the proposed language and ASP and use such relationship to establish some important properties of ASP$\{$f$\}$. Finally, we draw conclusions and discuss future work.

2 The Syntax of ASP$\{$f$\}$

In this section we define the syntax of ASP$\{$f$\}$. Because in this paper we focus exclusively on non-Herbrand functions, from now on we drop the "non-Herbrand" attribute. (Allowing for Herbrand functions does not involve technical difficulties, but would lengthen the presentation.)

The syntax of ASP$\{$f$\}$ is based on a signature $\Sigma = \langle \mathcal{C}, \mathcal{F}, \mathcal{R} \rangle$ whose elements are, respectively, sets of *constants*, *function symbols* and *relation symbols*. A *term* is an expression $f(c_1, \ldots, c_n)$ where $f \in \mathcal{F}$, and c_i's are 0 or more constants. An *atom* is an expression $r(c_1, \ldots, c_n)$, where $r \in \mathcal{R}$, and c_i's are constants. The set of all terms (resp., atoms) that can be formed from Σ is denoted by $\mathcal{T}$ (resp., $\mathcal{A}$). A *t-atom* is an expression of the form $f = g$, where f is a term and g is either a term or a constant. We call *seed t-atom* a t-atom of the form $f = v$, where v is a constant. Any t-atom that is not a seed t-atom is a *dependent t-atom*. Thus, given a signature with $\mathcal{C} = \{a, b, 0, 1, 2, 3, 4\}$ and $\mathcal{F} = \{occupancy, seats\}$, expressions $occupancy(a) = 2$ and $seats(b) = 4$ are seed t-atoms, while $occupancy(b) = seats(b)$ is a dependent t-atom.

A *regular literal* is an atom a or its strong negation $\neg a$. A *t-literal* is a t-atom $f = g$ or its strong negation $\neg(f = g)$, which we abbreviate $f \neq g$. A *dependent t-literal* is any t-literal that is not a seed t-atom. A *literal* is a regular literal or a t-literal. A *seed literal* is a regular literal or a seed t-atom. Given a signature with $\mathcal{R} = \{room\_evacuated\}$, $\mathcal{F} = \{occupancy, seats\}$ and $\mathcal{C} = \{a, b, 0, \ldots, 4\}$, $room\_evacuated(a)$, $\neg room\_evacuated(b)$ and $occupancy(a) = 2$ are seed literals (as well as literals); $room\_evacuated(a)$ and $\neg room\_evacuated(b)$ are also regular literals; $occupancy(b) \neq 1$ and $occupancy(b) = seats(b)$ are dependent t-literals, but they are not regular or seed literals.

A *rule* r is a statement of the form:

$$h \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n \qquad (3)$$

where h is a seed literal and l_i's are literals. Similarly to ASP, the informal reading of r is that a rational agent who believes $l_1, \ldots, l_m$ and has no reason to believe $l_{m+1}, \ldots, l_n$ must believe h. Given a signature with $\mathcal{R} = \{room\_evacuated, door\_stuck, room\_occupied, room\_maybe\_occupied\}$, $\mathcal{F} = \{occupancy\}$, $\mathcal{C} = \{0\}$, the following is an example of ASP{f} rules encoding knowledge about the occupancy of a room:

$$\begin{array}{l} r_1 : occupancy = 0 \leftarrow room\_evacuated,\ not\ door\_stuck. \\ r_2 : room\_occupied \leftarrow occupancy \neq 0. \\ r_3 : room\_maybe\_occupied \leftarrow not\ occupancy = 0. \end{array}$$

Intuitively, rule r_1 states that the occupancy of the room is 0 if the room has been evacuated and there is no reason to believe that the door is stuck. Rule r_2 says that the room is occupied if its occupancy is different from 0. On the other hand, r_3 aims at drawing a weaker conclusion, stating that the room *may* be occupied if there is no explicit knowledge (i.e. reason to believe) that its occupancy is 0.

Given rule r from (3), $head(r)$ denotes h; $body(r)$ denotes $\{l_1, \ldots, not\ l_n\}$; $pos(r)$ denotes $\{l_1, \ldots, l_m\}$; $neg(r)$ denotes $\{l_{m+1}, \ldots, l_n\}$.

A *constraint* is a special type of rule with an empty head, informally meaning that the condition described by the body of the rule must never be satisfied. A constraint is considered a shorthand of:

$$\bot \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n, not\ \bot$$

where $\bot$ is a fresh atom. Thus, the constraint

$$\leftarrow room\_occupied,\ door\_stuck.$$

states that it is impossible for the room to be occupied when the door is stuck.

A *program* is a pair $\Pi = \langle \Sigma, P \rangle$, where Σ is a signature and P is a set of rules. Whenever possible, in this paper the signature is implicitly defined from the rules of Π, and Π is identified with its set of rules. In that case, the signature is denoted by $\Sigma(\Pi)$ and its elements by $\mathcal{C}(\Pi)$, $\mathcal{F}(\Pi)$ and $\mathcal{R}(\Pi)$. A rule r is *positive* if $neg(r) = \emptyset$. A program Π is *positive* if every $r \in \Pi$ is positive. A program Π is also *t-literal free* if no t-literals occur in the rules of Π.

For practical purposes, it is often convenient to use variables in programs. In ASP{f}, variables can be used in place of constants and terms. The *grounding of a rule* r is the set of all the syntactically valid rules obtained by replacing every variable of r with an element of $\mathcal{C} \cup \mathcal{T}$. The *grounding of a program* Π is the set of the groundings of the rules of Π. A syntactic element of the language is *ground* if it is variable-free and *non-ground* otherwise. Thus, the fact that a room is unoccupied at a any step S in the evolution of a domain whenever the room is not accessible can be expressed by the non-ground rule:

$$occupancy(S) = 0 \leftarrow not\_accessible(S).$$

Given possible time steps $\{0, 1, 2\}$, the grounding of the rule is:

$$\begin{array}{c} occupancy(0) = 0 \leftarrow not\_accessible(0). \\ occupancy(1) = 0 \leftarrow not\_accessible(1). \\ occupancy(2) = 0 \leftarrow not\_accessible(2). \end{array}$$

3 Semantics of ASP{f}

The semantics of a non-ground program is defined to coincide with the semantics of its grounding. The semantics of ground ASP{f} programs is defined below. In the rest of this section, we consider only ground terms, literals, rules and programs and thus omit the word "ground."

A set S of seed literals is *consistent* if (1) for every atom $a \in \mathcal{A}$, $\{a, \neg a\} \not\subseteq S$; (2) for every term $t \in \mathcal{T}$ and v_1, $v_2 \in \mathcal{C}$ such that $v_1 \neq v_2$, $\{t = v_1, t = v_2\} \not\subseteq S$. Hence, $S_1 = \{p, \neg q, f = 3\}$ and $S_2 = \{q, f = 3, g = 2\}$ are consistent, while $\{p, \neg p, f = 3\}$ and $\{q, f = 3, f = 2\}$ are not. Incidentally, $\{p, \neg q, f = g, g = 2\}$ is not a set of seed literals, because $f = g$ is not a seed literal.

The *value* of a term t w.r.t. a consistent set S of seed literals (denoted by $val_S(t)$) is v iff $t = v \in S$. If, for every $v \in \mathcal{C}$, $t = v \notin S$, the value of t w.r.t. S is *undefined*. The value of a constant $v \in \mathcal{C}$ w.r.t. S ($val_S(v)$) is v itself. For example given S_1 and S_2 as above, $val_{S_2}(f)$ is 3 and $val_{S_2}(g)$ is 2, whereas $val_{S_1}(g)$ is undefined. Given S_1 and a signature with $\mathcal{C} = \{0, 1\}$, $val_{S_1}(1) = 1$.

A seed literal l is *satisfied* by a consistent set S of seed literals iff $l \in S$. A dependent t-literal $f = g$ (resp., $f \neq g$) is *satisfied* by S iff both $val_S(f)$ and $val_S(g)$ are defined, and $val_S(f)$ is equal to $val_S(g)$ (resp., $val_S(f)$ is different from $val_S(g)$). Thus, seed literals q and $f = 3$ are satisfied by S_2; $f \neq g$ is also satisfied by S_2 because $val_{S_2}(f)$ and $val_{S_2}(g)$ are defined, and $val_{S_2}(f)$ is different from $val_{S_2}(g)$. Conversely, $f = g$

is not satisfied, because $val_{S_2}(f)$ is different from $val_{S_2}(g)$. The t-literal $f \neq h$ is also not satisfied by S_2, because $val_{S_2}(h)$ is undefined. When a literal l is satisfied (resp., not satisfied) by S, we write $S \models l$ (resp., $S \not\models l$).

An *extended literal* is a literal l or an expression of the form *not* l. An extended literal *not* l is satisfied by a consistent set S of seed literals ($S \models$ *not* l) if $S \not\models l$. Similarly, $S \not\models$ *not* l if $S \models l$. Considering set S_2 again, extended literal *not* $f = h$ is satisfied by S_2, because $f = h$ is not satisfied by S_2.

Finally, a set E of extended literals is satisfied by a consistent set S of seed literals ($S \models E$) if $S \models e$ for every $e \in E$.

We begin by defining the semantics of ASP{f} programs for *positive* programs.

A set S of seed literals is *closed* under positive rule r if $S \models head(r)$ whenever $S \models pos(r)$. Hence, set S_2 described earlier is closed under $f = 3 \leftarrow g \neq 1$ and (trivially) under $f = 2 \leftarrow r$, but it is not closed under $p \leftarrow f = 3$, because $S_2 \models f = 3$ but $S_2 \not\models p$. S is closed under Π if it is closed under every rule $r \in \Pi$.

Finally, a set S of seed literals is an *answer set* of a positive program Π if it is consistent and closed under Π, and is minimal (w.r.t. set-theoretic inclusion) among the sets of seed literals that satisfy such conditions. Thus, the program:

$$\begin{array}{l} p \leftarrow f = 2. \\ f = 2. \\ q \leftarrow q. \end{array}$$

has an answer set $\{f = 2, p\}$. The set $\{f = 2\}$ is not closed under the first rule of the program, and therefore is not an answer set. The set $\{f = 2, p, q\}$ is also not an answer set, because it is not minimal (it is a proper superset of another answer set). Notice that positive programs may have no answer set. For example, the program

$$\begin{array}{l} f = 3 \leftarrow \textit{not } p. \\ f = 2 \leftarrow \textit{not } q. \end{array}$$

has no answer set. Programs that have answer sets (resp., no answer sets) are called *consistent* (resp., *inconsistent*).

Positive programs enjoy the following property:

Proposition 1. *Every consistent positive program Π has a unique answer set.*

Next, we define the semantics of arbitrary ASP{f} programs.

The *reduct* of a program Π w.r.t. a consistent set S of seed literals is the set Π^S consisting of a rule $head(r) \leftarrow pos(r)$ (the *reduct* of r w.r.t. S) for each rule $r \in \Pi$ for which $S \models body(r) \setminus pos(r)$. From Proposition 1 it follows that the reduct w.r.t. a given set has a unique answer set.

Example 1. Consider a set of seed literals $S_3 = \{g = 3, f = 2, p, q\}$, and program Π_1:

$$\begin{array}{l} r_1 : p \leftarrow f = 2, \textit{not}\ g = 1, \textit{not}\ h = 0. \\ r_2 : q \leftarrow p, \textit{not}\ g \neq 2. \\ r_3 : g = 3. \\ r_4 : f = 2. \end{array}$$

and let us compute its reduct. For r_1, first we have to check if $S_3 \models body(r_1) \setminus pos(r_1)$, that is if $S_3 \models \textit{not}\ g = 1, \textit{not}\ h = 0$. Extended literal *not* $g = 1$ is satisfied by S_3 only if $S_3 \not\models g = 1$. Because $g = 1$ is a seed literal, it is satisfied by S_3 if $g = 1 \in S_3$. Since $g = 1 \notin S_3$, we conclude that $S_3 \not\models g = 1$ and thus *not* $g = 1$ is satisfied by S_3. In a similar way, we conclude that $S_3 \models \textit{not}\ h = 0$. Hence, $S_3 \models body(r_1) \setminus pos(r_1)$. Therefore, the reduct of r_1 is $p \leftarrow f = 2$. For the reduct of r_2, notice that *not* $g \neq 2$ is not satisfied by S_3. In fact, $S_3 \models \textit{not}\ g \neq 2$ only if $S_3 \not\models g \neq 2$. However, it is not difficult to show that $S_3 \models g \neq 2$: in fact, $val_{S_3}(g)$ is defined and $val_{S_3}(g) \neq 2$. Therefore, *not* $g \neq 2$ is not satisfied by S_3, and thus the reduct of Π_1 contains no rule for r_2. The reducts of r_3 and r_4 are the rules themselves. Summing up, $\Pi_1^{S_3}$ is:

$$\begin{array}{l} r'_1 : p \leftarrow f = 2. \\ r'_3 : g = 3. \\ r'_4 : f = 2. \end{array}$$

Finally, a consistent set S of seed literals is an *answer set* of program Π if S is the answer set of Π^S.

Example 2. By applying the definitions given earlier, it is not difficult to show that the answer set of $\Pi_1^{S_3}$ is $\{f = 2, g = 3, p\} = S_3$. Hence, S_3 is an answer set of $\Pi_1^{S_3}$. Consider instead $S_4 = S_3 \cup \{h = 1\}$. Clearly $\Pi_1^{S_4} = \Pi_1^{S_3}$. From the uniqueness of the answer sets of positive programs, it follows immediately that S_4 is not the answer set of $\Pi_1^{S_4}$. Therefore, S_4 is not an answer set of Π_1.

Most properties of ASP programs are also enjoyed by ASP{f}, such as:

Proposition 2. *For every ASP{f} program Π and set of constraints C formed from $\Sigma(\Pi)$, S is an answer set of $\Pi \cup C$ iff S is an answer set of Π that does not satisfy the body of any constraint from C.*

4 Knowledge Representation with Non-Herbrand Functions

In this section we demonstrate the use of ASP{f} for knowledge representation, and compare the corresponding formalizations with those from the existing literature. We start our discussion by addressing the encoding of defaults.

Consider the statements: (1) the value of $f(x)$ is a unless otherwise specified; (2) the value of $f(x)$ is b if $p(x)$ (this example is from [10]; for simplicity of presentation we use a constant as the argument of function f instead of a variable as in [10], but our argument does not change even in the more general case). These statements can be encoded in ASP{f} as follows:

$$P_1 = \begin{cases} r_1 : f(x) = a \leftarrow \textit{not}\ f(x) \neq a. \\ r_2 : f(x) = b \leftarrow p(x). \end{cases}$$

Rule r_1 encodes the default, and r_2 encodes the exception. It is worth stressing that the informal reading of r_1, according to the description given earlier in this paper, is "if there is no reason to believe that $f(x)$ is different from a, then $f(x)$ must be equal to a", which is essentially identical to the original problem statement. We argue that this representation of the default is, at least by ASP standards, natural and direct. Moreover, it is not difficult to see that the formalization follows a strategy similar to that of the formalization of defaults in ASP. Consider the statement "$q(x)$ holds unless otherwise specified". A common way of encoding it in ASP is with a rule $q(x) \leftarrow \textit{not}\ \neg q(x)$. Not only the informal reading of this rule ("if there is no reason to believe that $q(x)$ is false, then it must be true") is close to the informal reading of r_1, but the rules themselves have a similar structure as well. On the other hand, this is not the case of the language of weight constraint programs with evaluable functions [15], where a substantially different representation strategy is adopted, in which the default is encoded as:

$$f(x) = a \leftarrow [f(x) \neq a : 1]0.$$

In this case there is arguably little similarity between the body of this rule and the body of the default for $q(x)$, both syntactically *and* conceptually.

In the language of IF-programs [10], the default for $f(x)$ has an encoding rather similar to that of r_1:

$$f(x) = a \leftarrow \neg(f(x) \neq a)$$

Note that, in the language of IF-programs, $\neg$ has a meaning similar to that of *not* here. Where the language deviates from ASP is in the fact that in the language of IF-programs the above rule is equivalent to:

$$f(x) = a \ \vee \ f(x) \neq a.$$

The equivalence of the two encodings is somewhat problematic from the point of view of the language requirements that we are seeking to satisfy in this paper. In fact, in ASP the epistemic disjunction operator $\vee$ denotes a non-deterministic choice between two alternatives: a statement $p \ \vee \ q$ in ASP means that p and q are equally acceptable alternatives. In the language of IF-programs, instead, the disjunction operator appears to express a preference for the left-hand-side expression. In this sense, the representation of disjunctive knowledge in ASP and in [10] follows two very different strategies. (It is not difficult to show that ASP{f} can be naturally extended to allow for disjunction with a semantics following closely that of ASP).

Another language that allows for a direct representation of functions is that of CLINGCON [8]. However, the representation of defaults involving functions in CLINGCON may yield rather unintended results if one follows the typical ASP knowledge representation strategies.

Consider a modification of the default discussed earlier in which the value of $f(x)$ is 1 by default (CLINGCON only supports functions with numerical values), and let us assume that $f(x)$ ranges over the set $\{0, 1\}$. One might be tempted to encode it in the language of CLINGCON as:

$$\begin{array}{l} \$domain(0..1). \\ f(x)\ \$\texttt{==}\ 1 \leftarrow \textit{not}\ f(x)\ \$\texttt{!=}\ 1. \end{array}$$

where the first statement specifies the domain of the functions and the second statement formalizes the default, with prefix \$ denoting equality and inequality of functions. As one would expect, this program has an answer set, $\{f(x) = 1\}$, in which $f(x)$ has its default value of 1. However, the program also has a second, unintended answer set, $\{f(x) = 0\}$, in which $f(x)$ is assigned the non-default value of 0. Clearly, in CLINGCON defaults cannot be represented using the typical ASP techniques.

As mentioned in the Introduction, a difference between languages with partial functions and languages with total functions is the way incomplete or uncertain information can be encoded. Suppose we know that function $f(x)$ ranges over $\{a, b\}$, but we do not know its value. In ASP, this could be encoded by the program:

$$\begin{array}{l} f'(x,a) \leftarrow \textit{not}\ f'(x,b). \\ f'(x,b) \leftarrow \textit{not}\ f'(x,a). \\ \leftarrow f'(x,a),\ f'(x,b). \end{array} \tag{4}$$

The first two rules informally say that we know that the value of $f(x)$ is either a or b, but we do not know which one it is; the last rule characterizes $f'(x, v)$ as a function. It is not difficult to show that this program has two answer sets, $\{f'(x, a)\}$ and $\{f'(x, b)\}$, each corresponding to one possible assignment of value to $f(x)$. Alternatively, the same scenario can also be encoded in ASP by the program:

$$\begin{array}{l} \leftarrow f'(x,a),\ f'(x,b). \\ \leftarrow f'(x,V),\ V \neq a,\ V \neq b. \end{array} \tag{5}$$

where the first constraint is as before, and the second constraint restricts the domain of $f(x)$ to $\{a, b\}$. This program has a unique, empty answer set. The fact that no literal of the form $f'(x, v)$ occurs in the answer set implies that the value of $f(x)$ is not known.

Similarly, in languages that support partial functions, such as ASP{f} and [6], both methods can be applied for the representation of incomplete information about functions. For instance, in ASP{f} an equivalent of (4) is:

$$\begin{array}{l} f(x) = a \leftarrow \textit{not}\ f(x) = b. \\ f(x) = b \leftarrow \textit{not}\ f(x) = a. \end{array} \tag{6}$$

whose answer sets, along the lines of those of the ASP formalization, are $\{f(x) = a\}$ and $\{f(x) = b\}$. A formalization equivalent to (5) is an empty program (with a suitable signature), which has an empty answer set. According to the semantics defined earlier, an empty set entails that statements such as $f(x) = a$ and $f(x) \neq a$ are neither true nor false, implying that the value of $f(x)$ is unknown.

On the other hand, in languages that only allow for total functions, such as [10], [12], [15] and [8], the incompleteness of information about functions can only be represented by means of multiple answer sets. For example, in the language of [12] an empty program together with a specification of domain $\{a, b\}$ for $f(x)$ yields two answer sets, $\{f(x) = a\}$, $\{f(x) = b\}$. Because the functions specified by the formalizations are required to be total, there is no way to describe uncertainty about $f(x)$ by means of answer sets where the value of $f(x)$ unspecified.

Extending a common ASP methodology, the technique used in (6) to formalize the choice between a and b can also be easily extended to represent functions whose domains have more than two elements and to incorporate default values. Consider the statements (adapted from [10]): (1) the value $f(X)$ is a if $p(X)$; (2) otherwise, the value of $f(X)$ is arbitrary. Let the domain of variable X be given by a relation $dom(X)$, and let the possible values of $f(X)$ be encoded by a relation $val(V)$. A possible ASP{f} encoding of these statements is:

$$r_1 : f(X) = a \leftarrow\ p(X),\ dom(X).$$

$$r_2 : f(X) = V \leftarrow\ dom(X),\ val(V),\ \textit{not}\ p(X),\ \textit{not}\ f(X) \neq V.$$

Rule r_1 encodes the first statement. Rule r_2 formalizes the arbitrary selection of values for $f(X)$ in the default case. It is important to notice that, although r_2 follows a strategy of formalization of knowledge that is similar to that of ASP, the ASP{f} encoding is more compact than the corresponding ASP one. In fact, the ASP encoding requires the introduction of an extra rule formalizing the fact that $f(x)$ has a unique value:

$$r_1' : f'(X) = a \leftarrow\ p(X),\ dom(X).$$

$$r_2' : f'(X, V) \leftarrow\ dom(X),\ val(V),\ \textit{not}\ p(X),\ \textit{not}\ \neg f'(X, V).$$

$$r_3' : \neg f'(X, V') \leftarrow\ val(V),\ val(V'),\ V \neq V',\ f'(X, V).$$

Not only having to write r_3' is rather inconvenient, but this kind of rule may also have quite a negative impact on the performance of the ASP solvers used to compute the answer sets of the program. In fact, it is not difficult to show that the grounding of r_3' grows proportionally to the square of the size of the domain of $f(x)$. For functions with large domains, this growth can cause performance problems (and cause the grounding of rules like r_3' to become substantially larger than the grounding of the rest of the program). On the other hand, the grounding of the corresponding ASP{f} program does not suffer from such a growth, and a solver could in principle take advantage of that and compute the program's answer sets substantially faster.

A similar use of defaults is typically associated, in ASP, with the representation of dynamic domains. In this case, defaults are a key tool for the encoding of the law of inertia. Let us show how dynamic domains involving functions can be represented in ASP{f}. Consider a domain including a button b_i, which increments a counter c, and a button b_r, which resets it. At each time step, the agent operating the buttons may press either button, or none. A possible ASP{f} encoding of this domain is:

$$r_1 : val(c, S+1) = 0 \leftarrow\ pressed(b_r, S).$$

$$r_2 : val(c, S+1) = N+1 \leftarrow\ pressed(b_i, S),\ val(c, S) = N.$$

$$r_3 : val(c, S+1) = N \leftarrow\ val(c, S) = N,\ \textit{not}\ val(c, S+1) \neq N.$$

Rules r_1 and r_2 are a straightforward encoding of the effect of pressing either button (variable S denotes a time step). Rule r_3 is the ASP{f} encoding of the law of inertia for the value of the counter, and states that the value of c does not change unless it is forced to. For simplicity of presentation, it is instantiated for a particular function, but could be as easily written so that it applies to arbitrary functions from the domain. Rule r_3 follows the same encoding strategy used for relations in ASP, where the inertia law for a relational fluent p typically takes the form:

$$\begin{array}{l} p(S+1) \leftarrow p(S),\ \textit{not}\ \neg p(S). \\ \neg p(S+1) \leftarrow p(S),\ \textit{not}\ p(S). \end{array} \tag{7}$$

The only difference is in the fact that (7) uses two rules because p and $\neg p$ are treated separately due to syntactic restrictions of ASP. Incidentally, it is not difficult to see that (7) is also a valid encoding of inertia for relational fluents in ASP{f}.

Let us now consider a typical ASP encoding for the above domain:

$$r'_1 : val(c, S+1, 0) \leftarrow\ pressed(b_r, S).$$

$$r'_2 : val(c, S+1, N+1) \leftarrow\ pressed(b_i, S),\ val(c, S, N).$$

$$r'_3(a) : val(c, S+1, N) \leftarrow\ val(c, S, N),\ \textit{not}\ \neg val(c, S+1, N).$$

$$r'_3(b) : \neg val(c, S, N') \leftarrow\ val(c, S, N),\ N \neq N'.$$

Rules r'_1, r'_2 are similar to their ASP{f} counterparts. The only difference is that, taken out of context, r'_1 and r'_2 do not provide any indication that val is a function, and that as a consequence only one of $\{val(c, s, 0), val(c, s, 1), \ldots\}$ can hold at any step s. As mentioned earlier, this difference is rather important from a knowledge representation perspective, as it reduces the declarativity of the rules. Rule $r'_3(a)$ encodes the law of inertia. Because this encoding represents functions by means of relations, the rule depends on a suitable definition of the axioms of the uniqueness of value for val, formalized by $r'_3(b)$. As we discussed in the previous example, the grounding of $r'_3(b)$ can grow quite substantially – in fact, in practice, it can grow even more dramatically than in the previous example, because of the extra argument for the time step. The ASP{f} representation is thus not only more natural, but also potentially more efficient.

Coming back to the comparison between languages that allow partial functions and languages that do not allow them, another important distinguishing feature is the fact that in the languages with partial functions all conclusions are supported. A conclusion, i.e. a literal l from an answer set A of a program Π, is *supported* if it is in the head of some rule r whose reduct with respect to A has its body satisfied by A. In ASP, as well as in ASP$\{$f$\}$ and [6], all conclusions enjoy this property. Both from a practical perspective and from the standpoint of knowledge representation, this feature has the advantage that a programmer can look at a program and rather easily understand which literals may be in the program's answer sets, and which ones cannot be in any answer set. Similarly, it is not difficult, given a literal from an answer set, to identify which rules may have caused its derivation. In languages with total functions, on the other hand, conclusions are not required to be supported. To highlight the ramifications of this difference from a knowledge representation perspective, let us consider the graph coloring problem (see e.g. [6,12]). In this problem, one must assign a color to each node of a graph so that no two adjacent nodes have the same color. A possible ASP$\{$f$\}$ formalization is:

$$color(X) = V \leftarrow\ node(X),\ available\_color(V),\ \textit{not}\ color(X) \neq V.$$

$$\leftarrow arc(X,Y),\ color(X) = color(Y).$$

The first rule states that each node can be assigned an arbitrary color, thus making $color$ a total function. The second rule says that two adjacent nodes are not allowed to have the same color. In the language of [12], the graph coloring problem admits a solution that is even more compact:

$$\leftarrow arc(X,Y),\ color(X) = color(Y). \tag{8}$$

The reason why this is a solution to the graph coloring problem is because in [12] functions are total. Therefore, there is no need for an extra statement that forces $color(X)$ be defined. But at the same time, conclusions such as $color(n_1) = red$ will occur in the answer sets of (8) without occurring in the head of any rule. In this sense, (8) does not follow the typical knowledge representation strategies of ASP. In fact, it follows a quite opposite representation strategy: consider the ASP program consisting of a definition R of relation arc and of:

$$\leftarrow arc(X,Y),\ color(X,C),\ color(Y,C). \tag{9}$$

Because relation $color$ does not occur in the head of any rule, one can immediately conclude that the constraint is never triggered and that the program has thus a unique answer set consisting only of the definition of relation arc. Therefore, the color of any node X is unknown. That is, although program $R \cup (9)$ is syntactically very similar to $R \cup (8)$, their meanings are very different!

Finally, as pointed out in [6], in languages that allow for the representation of partial functions a statement such as "Louis XIV is not the king of France" may be intended in either one of two ways: (1) "whether France has a king or not, definitely Louis XIV is not the king of France", and (2) "France has a king, and it is not Louis XIV." The difference between the two readings is that in the second case the statement is true only if France is known to have a king (i.e. if the function is defined), whereas in the first case the statement is true even if we have no knowledge about France having a king. In [6], the apartness operator $\#$ is introduced, which informally states that the functions being compared (1) are defined and (2) have different values. So, the first reading of the sentence is encoded as *not* $king(france) = louisXIV$, whereas the second reading is encoded as $king(france) \# louisXIV$. In ASP{f}, on the other hand, one can achieve the same result by combining default negation and strong negation, as is normally done in ASP. More precisely, the first reading can be expressed in ASP{f} by the statement:

$$\textit{not } king(france) = louisXIV \tag{10}$$

which, as explained earlier, has the informal reading of "there is no reason to believe that LouisXIV is the king of France". The second reading can be encoded as

$$king(france) \neq louisXIV. \tag{11}$$

which informally states "the king of France is different from Louis XIV" (recall that in ASP{f} $king(france) \neq louisXIV$ is an abbreviation of $\neg(king(france) = louisXIV)$). According to the semantics of ASP{f} defined earlier, (10) is satisfied by a consistent set S of seed literals if $king(france) = louisXIV$ is not satisfied by S. Because $S \models king(france) = louisXIV$ iff $val_S(king(france))$ is defined and $val_S(king(france))$ is $louisXIV$, it follows that (10) is satisfied if either $val_S(king(france))$ is undefined, or $val_S(king(france))$ is different from $louisXIV$. On the other hand, (11) is satisfied by S iff $val_S(king(france))$ is defined *and* different from $louisXIV$. It is worth stressing that $king(france) \neq louisXIV$ is just a syntactic variant of $\neg(king(france) = louisXIV)$, which implies that, in ASP{f} as well, the use of default and strong negation allows avoiding the introduction of an ad-hoc comparison operator.

5 Relationship with ASP

In this section we establish some useful formal relationships between ASP{f} and ASP, and use them to derive some important properties of ASP{f}.

To distinguish between the definitions given above in the context of ASP{f} and the corresponding definitions in the context of ASP, in this section we prefix ASP{f}-related terms with ASP{f}. So, if literal l is satisfied by a consistent set S of seed literals, we say that it is *ASP{f}-satisfied.* Similarly we say that S is *ASP{f}-closed* under a rule r. When we refer to the traditional ASP definitions, we use prefix ASP and say for example *ASP-satisfied* and *ASP-closed.* We introduce a similar distinction in the notation, and use symbols $\models_{ASP\{f\}}$ and $\models_{ASP}$ respectively.

For t-literal free programs, it is not difficult to prove that the following proposition holds:

Proposition 3. *For every t-literal free program Π, A is an answer set of Π under the ASP{f} semantics iff A is an answer set of Π under the ASP semantics.*

Because of this result, when in this section we use the term "answer set", we leave the semantics implicitly defined by whether the corresponding program is t-literal free or not. With similar reasoning, we do not explicitly refer to a semantics when referring to the reduct of a program.

Let us now consider arbitrary ASP{f} programs. An ASP{f} extended literal e can be mapped into an ASP extended literal (defined, as usual in ASP, as a literal l or the expression *not* l) by replacing every occurrence of a t-literal $f = x$ (resp., $f \neq x$) in e by $eq(f, x)$ (resp., $\neg eq(f, x)$), where eq is a fresh relation symbol. We denote the *ASP-mapping* of e by $\alpha(e)$. The notion of ASP-mapping is extended to sets of extended literals ($\alpha(\{e_1, \ldots, e_k\})$), rules ($\alpha(r)$), and programs ($\alpha(\Pi)$) in a straightforward way. The inverse of the α mapping is denoted by α^{-1}.

Now, let Π be an ASP{f} program. We define the *ASP-completion of* Π to be $\gamma(\Pi) = \alpha(\Pi) \cup \sigma_r(\Pi)$ where $\sigma_r(\Pi)$ is formed as follows:

- For every term t from $\Sigma(\Pi)$ and $v, v' \in \mathcal{C}(\Pi)$ such that $v \neq v'$, $\sigma_r(\Pi)$ contains a rule $\neg eq(t, v') \leftarrow eq(t, v)$.
- For every pair of terms f, g from $\Sigma(\Pi)$, every $v, v_f, v_g \in \mathcal{C}(\Pi)$ such that $v_f \neq v_g$, $\sigma_r(\Pi)$ contains the rules:

$$eq(f, g) \leftarrow eq(f, v),\ eq(g, v). \qquad \neg eq(f, g) \leftarrow eq(f, v_f),\ eq(g, v_g).$$

The next definition will be used later to link answer sets of ASP{f} programs and of their ASP-completions. Let $\Sigma = \langle \mathcal{C}, \mathcal{F}, \mathcal{R} \rangle$ S be a set of seed literals from Σ. The *ASP-completion* of S ($c(S)$) is:

$$\begin{aligned} c(S) = \alpha(S)\ \cup \\ &\{\neg eq(t, v') \mid t = v \in S \wedge v \in \delta(t) \wedge v' \in \delta(t) \wedge v \neq v'\}\ \cup \\ &\{eq(f, g) \mid f = v \in S \wedge v \in \delta(f) \wedge g = v \in S \wedge v \in \delta(g)\}\ \cup \\ &\{\neg eq(f, g) \mid f = v_f \in S \wedge v_f \in \delta(f) \wedge g = v_g \in S \wedge v_g \in \delta(g) \wedge v_f \neq v_g\} \end{aligned}$$

Intuitively, the ASP-completion of a set S of seed literals adds to S the dependent t-literals that are ASP{f}-entailed by S.

We are now ready to state the main results of this section on the relationship between ASP and ASP{f}.

Proposition 4. *For every ASP{f} program Π and every consistent set A of seed literals, $\alpha(\Pi^A) = \alpha(\Pi)^{c(A)}$.*

Proposition 5. *For every ASP{f} program Π: (1) if A is an answer set of Π then $c(A)$ is an answer set of $\gamma(\Pi)$; (2) if B is an answer set of $\gamma(\Pi)$ then there exists A such that $B = c(A)$ and A is an answer set of Π.*

From Propositions 4 and 5, the following properties follow:

Proposition 6. *The task of deciding whether a consistent set of seed literals is an answer set of an ASP{f} program is coNP-complete. The task of finding an answer set of an ASP{f} program is Σ_2^P-complete.*

The same propositions also allow to extend the Splitting Set Theorem [11] to ASP$\{$f$\}$. Let us call *splitting set* for an ASP$\{$f$\}$ program Π a set U of seed literals such that, for every rule $r \in \Pi$, if $head(r) \cap U \neq \emptyset$, then $body(r) \subseteq U$. The set of rules of $r \in \Pi$ such that $body(r) \subseteq U$ is denoted by $b_U(\Pi)$. The *partial evaluation*, $e_U(\Pi, X)$ of Π with respect to U and set of seed literals X is the set containing, for every $r \in \Pi$ such that $pos(r) \cap U \subseteq X$ and $neg(r) \cap U$ is disjoint from X, a rule r', where $head(r') = head(r)$, $pos(r') = pos(r) \setminus U$ and $neg(r') = neg(r) \setminus U$. We say that a *solution* to Π with respect to U is a pair $\langle X, Y\rangle$ of sets of seed literals such that (1) X is an answer set for $b_U(\Pi)$; (2) Y is an answer set for $e_U(\Pi \setminus b_U(\Pi), X)$; (3) $X \cup Y$ is consistent.

Proposition 7. *A set A of seed literals is an answer set of ASP$\{$f$\}$ program Π if and only if $A = X \cup Y$ for some solution $\langle X, Y\rangle$ to Π with respect to U.*

6 Conclusions and Future Work

In this paper we have defined the syntax and semantics of an extension of ASP that supports the direct use of non-Herbrand functions. As we have discussed throughout the paper, the ability to represent directly non-Herbrand functions has important advantages from both a knowledge-representation perspective, a practical perspective, and the perspective of solver performance. Compared to other approaches for the introduction of non-Herbrand functions in ASP, our language is more "conservative", in that it is intentionally defined so as to allow for the use of the same knowledge representation strategies of ASP. Because the definition of the semantics of ASP$\{$f$\}$ is based on the one from [9], it allows for a comparatively simple incorporation in the new language of certain extensions of ASP such as weak constraints, probabilistic constructs and consistency-restoring rules. The simplicity of the modification of the original semantics makes it also relatively straightforward to extend properties of ASP programs to ASP$\{$f$\}$ programs.

Although the results from the previous section could in principle be used to implement an ASP$\{$f$\}$ solver, the simplicity of our modification to the semantics from [9] makes it possible to extend state-of-the-art ASP solvers to provide direct support for ASP$\{$f$\}$. We have implemented an ASP$\{$f$\}$ solver using this strategy and compared its performance to the performance obtained with the translations to normal logic programs described in [6] and [10]. In our preliminary experimental results, the performance of our ASP$\{$f$\}$ solver is consistently more than an order of magnitude better than the performance obtained with the translation to normal logic programs.

A topic that is not addressed in this paper, is that of expressions with nested non-Herbrand functions. For example, in the language of [6] it is possible to write an expression such as $mother(father(mother(X)))$, and in the language of [12] one can write a rule such as $reached(hc(X)) \leftarrow reached(X)$. Although these expressions can be encoded in ASP$\{$f$\}$ using additional variables (e.g. $reached(Y) \leftarrow reached(X),\ Y = hc(X)$), we believe that a more direct support for expressions with nested non-Herbrand functions in ASP$\{$f$\}$ could not only allow for more compact rules, but could also be effectively exploited for an efficient implementation of the ASP$\{$f$\}$ solver.

Acknowledgments. We would like to thank Michael Gelfond, Daniela Inclezan and Stefan Woltran for useful feedback and suggestions.

References

1. Balduccini, M., Gelfond, M.: Diagnostic reasoning with A-Prolog. Journal of Theory and Practice of Logic Programming (TPLP) 3(4–5), 425–461 (Jul 2003)
2. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press (Jan 2003)
3. Baral, C., Gelfond, M., Rushton, N.: Probabilistic reasoning with answer sets. Journal of Theory and Practice of Logic Programming (TPLP) 9(1), 57–144 (2009)
4. Baselice, S., Bonatti, P.A.: A Decidable Subclass of Finitary Programs. Journal of Theory and Practice of Logic Programming (TPLP) 10(4–6), 481–496 (2010)
5. Buccafurri, F., Leone, N., Rullo, P.: Adding Weak Constraints to Disjunctive Datalog. In: Proceedings of the 1997 Joint Conference on Declarative Programming APPIA-GULP-PRODE'97 (1997)
6. Cabalar, P.: Functional Answer Set Programming. Journal of Theory and Practice of Logic Programming (TPLP) 11, 203–234 (2011)
7. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Enhancing ASP by Functions: Decidable Classes and Implementation Techniques. In: Proceedings of the Twenty-Fourth Conference on Artificial Intelligence. pp. 1666–1670 (2010)
8. Gebser, M., Ostrowski, M., Schaub, T.: Constraint Answer Set Solving. In: 25th International Conference on Logic Programming (ICLP09). vol. 5649 (2009)
9. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing 9, 365–385 (1991)
10. Lifschitz, V.: Logic Programs with Intensional Functions (Preliminary Report). In: ICLP11 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP11) (Jul 2011)
11. Lifschitz, V., Turner, H.: Splitting a logic program. In: Proceedings of the 11th International Conference on Logic Programming (ICLP94). pp. 23–38 (1994)
12. Lin, F., Wang, Y.: Answer Set Programming with Functions. In: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR2008). pp. 454–465 (2008)
13. Marek, V.W., Truszczynski, M.: The Logic Programming Paradigm: a 25-Year Perspective, chap. Stable Models and an Alternative Logic Programming Paradigm, pp. 375–398. Springer Verlag, Berlin (1999)
14. Syrjanen, T.: Omega-Restricted Logic Programs. In: Eiter, T., Faber, W., Truszczynski, M. (eds.) 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR01). Lecture Notes in Artificial Intelligence (LNCS), vol. 2173, pp. 267–279. Springer Verlag, Berlin (2001)
15. Wang, Y., You, J.H., Yuan, L.Y., Zhang, M.: Weight Constraint Programs with Functions. In: Erdem, E., Lin, F., Schaub, T. (eds.) 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR09). Lecture Notes in Artificial Intelligence (LNCS), vol. 5753, pp. 329–341. Springer Verlag, Berlin (Sep 2009)