Answer Set Solving and 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, and describe a solver for the new language. 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. Our approach stems from our interest in practical applications, and from the corresponding need to compute the answer sets of programs with non-Herbrand functions efficiently. Our extension of ASP is such that the semantics of the new language is obtained by a comparatively small change to the ASP semantics from (Gelfond and Lifschitz 1991). The nature of this change makes it possible to modify a state-of-the-art ASP solver in an incremental fashion, and use it for the computation of the answer sets of (a large class of) programs of the new language. The computation is rather efficient, as demonstrated by our experimental evaluation.

Introduction

In this paper we describe an extension of Answer Set Programming (ASP) (Gelfond and Lifschitz 1991; Marek and Truszczynski 1999; Baral 2003) called ASP$\{f\}$, and a solver for the new language.

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 (we refer the reader to e.g. (Calimeri et al. 2010; Baselice and Bonatti 2010; Syrjänen 2001)).

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

$$occupancy'(S+1, O+1) \leftarrow occupancy'(S, O). \quad (1)$$

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

$$\neg occupancy'(S, O') \leftarrow occupancy'(S, O),\ O \neq O'. \quad (2)$$

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.

Various extensions of ASP with non-Herbrand functions exist in the literature. In (Cabalar 2011), Quantified Equilibrium Logic is extended with support for equality. A subset of the general language, called FLP, is then identified which can be translated into normal logic programs. Such translation makes it possible to compute the answer sets of FLP programs using ASP solvers. (Lifschitz 2011) proposes instead the use of second-order theories for the definition of the semantics of the language. Again, a transformation is described, which removes non-Herbrand functions and makes it possible to use ASP solvers for the computation of the answer sets of programs in the extended language. In (Lin and Wang 2008; Wang et al. 2009) the semantics is based on the notion of reduct as in the original ASP semantics (Gelfond and Lifschitz 1991). For the purpose of computing answer sets, a translation is defined, which maps programs of the language from (Lin and Wang 2008; Wang et al. 2009) to constraint satisfaction problems, so that CSP solvers can be used for the computation of the answer sets of programs in the extended language. Finally, the language of CLINGCON (Gebser, Ostrowski, and Schaub 2009) extends ASP with elements from constraint satisfaction. The CLINGCON solver finds the answer sets of a program by interleaving the computations of an ASP solver and of a CSP solver.

Our investigation stems for our interest in practical applications, and in particular from the need for a knowledge representation language with non-Herbrand functions that can be used for such applications and that allows for an efficient computation of answer sets. From this point of view, the existing approaches have certain limitations.

The transformations to constraint satisfaction problems used in (Lin and Wang 2008; Wang et al. 2009) certainly allow for an efficient computation of answer sets using constraint solving techniques, as demonstrated by the experimental results in (Wang et al. 2009). On the other hand, the recent successes of solvers based on the Conflict-Driven Clause Learning technique (CDCL for short, see e.g. (Goldberg and Novikov 2002)) such as CLASP (Gebser et al. 2007) have shown that for certain domains CSP solvers perform poorly compared to CDCL-based solvers. For practical applications it is therefore important to ensure the availability of a CDCL-based solver as well. Furthermore, as observed in (Cabalar 2011), the requirement made in (Lin and Wang 2008; Wang et al. 2009) that non-Herbrand functions be total yields some counterintuitive results in certain knowledge representation tasks, which, from our point of view, limits the practical applications of the language. This argument also holds for CLINGCON. An additional limitation of CLINGCON is the fact that the interleaved computation it performs carries some overhead.

In both (Cabalar 2011) (where functions are partial) and (Lifschitz 2011) (where functions are total) the computation of the answer sets of a program is obtained by translating the program into a normal logic program, and then using state-of-the-art ASP solving techniques and solvers. Unfortunately, in both cases the translation to normal logic programs causes a substantial growth of the size of the translated (ground) program compared to the original (ground) program. Two, similar and often concurrent, reasons exist for this growth.

First of all, when a non-Herbrand function is removed and replaced by a relation-based representation, axioms that ensure the uniqueness of value of the function have to be introduced. In (Cabalar 2011), for example, when a function $f(\cdot)$ is removed, the following constraint is introduced:

$$\leftarrow holds\_f(X,V), holds\_f(X,W), V \neq W. \tag{3}$$

As usual, before an ASP solver can be used, this constraint must in turn be replaced by its ground instances, obtained by substituting every variable in it by a constant. This process causes the appearance of $|D_f|^2 \cdot |C_f|$ ground instances, where D_f and C_f are respectively the domain and the co-domain of function f. In the presence of functions with a sizable domain and/or co-domain, the number of ground instances of (3) can grow quickly and impact the performance of the solver rather substantially.

Secondly, certain syntactic elements of these extended languages, once mapped to normal logic programs, can also yield translations with large ground instances. Taking again (Cabalar 2011) as an example (the transformation in (Lifschitz 2011) appears to follow the same pattern), consider the FLP rule:

$$p(x) \leftarrow f(x) \# g(x). \tag{4}$$

which intuitively says that $p(x)$ must hold if f and g have different values for x *and* are both defined. During the transformation to normal logic programs, this rule is translated into:

$$p(x) \leftarrow Y \neq Z, holds\_f(x,Y), holds\_g(x,Z).$$

Similarly to the previous case, the number of ground instances of this rule grows proportionally with $|D_f|^2$, and in the presence of non-Herbrand functions with sizable domains, solver performance can be affected quite substantially. Although one might argue that it is possible to modify an ASP solver to guarantee that (3) is enforced without the need to explicitly specify it in the program, such a solution is unlikely to be applicable in the case of an arbitrary rule such as (4).

In response to these issues, in this paper we define an extension of ASP with non-Herbrand functions, called ASP{f}, that is obtained with a comparatively small modification to the semantics from (Gelfond and Lifschitz 1991). The nature of this change makes it possible to modify a state-of-the-art ASP solver in an incremental fashion, and to use it directly for the computation of the answer sets of (a large class of) ASP{f} programs. This prevents the phenomenon of the quadratic growth of the ground instance described above and results in a rather efficient computation, as demonstrated later in the paper.

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. Next, we describe our ASP{f} solver and report experimental results. Finally, we draw conclusions and discuss future work.

The Syntax of ASP{f}

In this section we define the syntax of ASP{f}. To keep the presentation simple, the version of ASP{f} described in this paper does not allow for Herbrand functions, and thus from now on we drop the "non-Herbrand" attribute. (Allowing for Herbrand functions is straightforward.)

The syntax of ASP$\{f\}$ is based on a signature $\Sigma = \langle \mathcal{C}, \mathcal{F}, \mathcal{R} \rangle$ whose elements are, respectively, finite 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:

$$h \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n \tag{5}$$

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 $\mathcal{R} = \{room\_evacuated, door\_stuck, room\_occupied, room\_maybe\_occupied\}$, $\mathcal{F} = \{occupancy\}$ and $\mathcal{C} = \{0\}$, the following is an example of ASP$\{f\}$ rules encoding knowledge about the occupancy of a room:

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

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 (5), $head(r)$ 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 constraint 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.

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.

Like in ASP, in ASP$\{f\}$ too 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 (its *ground instances*) 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 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 $\mathcal{C} = \{0, 1, 2\}$, the grounding of the rule is:

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

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. It is worth noting that the semantics of ASP$\{f\}$ is obtained from that of ASP in (Gelfond and Lifschitz 1991) by simply extending entailment to t-literals.

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 $S \models h$, where $head(r) = \{h\}$, 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 one 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

$$\begin{array}{l} f = 3. \\ f = 2. \end{array}$$

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 ASP{f} 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 $S \models body(r) \setminus pos(r)$.

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

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

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* $S_3 \models$ *not* $g = 1$, *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. *With the same reasoning, we conclude that* $S_3 \models$ *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* $g \neq 2$ *(which intuitively means "*g may *equal* 2*") is not satisfied by* S_3. *In fact,* $S_3 \models$ *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 an 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 an 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.

Knowledge Representation with ASP{f}

In this section we demonstrate the use of ASP{f} for the formalization of important types of knowledge. We start our discussion by addressing the encoding of defaults, and we also use this opportunity to give an illustration of the related languages.

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 (Lifschitz 2011); for simplicity of presentation we use a constant as the argument of function f instead of a variable as in (Lifschitz 2011), but our argument does not change even in the more general case). These statements can be encoded in ASP{f} as follows:

$$P_1 = \left\{ \begin{array}{l} r_1 : f(x) = a \leftarrow not\ f(x) \neq a. \\ r_2 : f(x) = b \leftarrow p(x). \end{array} \right.$$

Rule r_1 encodes the default, and r_2 encodes the exception. 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". In the language of weight constraint programs with evaluable functions (Wang et al. 2009) 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 the language of IF-programs (Lifschitz 2011), 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. The language of IF-programs also allows for the equivalent, alternative encoding of the default:

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

which appears to give to symbol $\vee$ a meaning similar to that of ordered disjunction (Brewka, Niemelä, and Syrjänen 2004).

In the language of CLINGCON (Gebser, Ostrowski, and Schaub 2009) the representation of defaults involving functions appears to yield unintended results if one follows the traditional 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:

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

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.

Finally, the encoding of the above default in the language of (Cabalar 2011) is:

$$f(x) = a \leftarrow \neg f(x) \# a.$$

where $\neg$ represents default negation and $\#$ is the *apartness operator*, which ensures not only that $f(x)$ is different from a, *but that it is also defined.*

Extending a common ASP methodology, the choice of value for a non-Herbrand function can be encoded in ASP{f} by means of default negation. Consider the statements (adapted from (Lifschitz 2011)): (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:

$$\begin{array}{l} r_1 : f(X) = a \leftarrow \\ \qquad p(X),\ dom(X). \end{array}$$

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

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:

$$r'_1 : f'(X) = a \leftarrow\ p(X),\ dom(X).$$

$$\begin{array}{l} r'_2 : f'(X, V) \leftarrow \\ \qquad dom(X),\ val(V),\ \textit{not}\ p(X),\ \textit{not}\ \neg f'(X, V). \end{array}$$

$$\begin{array}{l} r'_3 : \neg f'(X, V') \leftarrow \\ \qquad val(V),\ val(V'),\ V \neq V',\ f'(X, V). \end{array}$$

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:

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

$$\begin{array}{l} r_2 : val(c, S + 1) = N + 1 \leftarrow \\ \qquad pressed(b_i, S),\ val(c, S) = N. \end{array}$$

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

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.

Computing the Answer Sets of ASP{f} Programs

In this section we describe an algorithm, CLASP{f}, which computes the answer sets of ASP{f} programs. Although CLASP{f} is based on the CLASP algorithm (Gebser et al. 2007), the approach can be easily extended to other ASP solvers. In our description we follow the notation of (Gebser et al. 2007), to which the interested reader can refer for more details on the CLASP algorithm.

As customary, the algorithm operates on ground programs. To keep the presentation simple, we further assume that every program Π considered in this section contains, for every atom a from Π, a constraint $\leftarrow a, \neg a$ (usually this constraint is added automatically by the solver).

Given a literal l, a *signed literal* is an expression of the form $\mathbf{T}l$ or $\mathbf{F}l$. Given a signed literal σ, $\overline{\sigma}$, called the *complement* of σ, denotes $\mathbf{F}l$ if σ is $\mathbf{T}l$, and $\mathbf{T}l$ otherwise. An assignment A over some domain D is a sequence $\langle \sigma_1, \ldots, \sigma_n \rangle$ of signed literals for literals from D. The domain of A is denoted by $dom(A)$. The expression $A \circ B$ denotes the concatenation of assignments A and B. For an assignment A, we denote by A^T the set of literals l such that $\mathbf{T}l$ occurs in A; A^F is instead the set of literals l such that $\mathbf{F}l$ occurs in A.

A *nogood* is a set $\{\sigma_1, \ldots, \sigma_n\}$ of signed literals. An assignment A is a *solution* for a set Δ of nogoods if (1) $A^T \cup A^F = dom(A)$; (2) $A^T \cap A^F = \emptyset$; and (3) for every $\delta \in \Delta$, $\delta \not\subseteq A$. Given a nogood δ, a signed literal $\sigma \in \delta$ and an assignment A, $\overline{\sigma}$ is called *unit-resulting* for δ w.r.t. A if $\delta \setminus A = \{\sigma\}$ and $\overline{\sigma} \notin A$. *Unit propagation* is the process of iteratively extending A with unit-resulting signed literals until no signed literal is unit-resulting for any nogood in Δ.

At the core of the computation of the answer sets of a program in CLASP{f} is the process of mapping the program to a suitable set of nogoods. Such mapping is described next, beginning with the nogoods already used in CLASP.

Given a program Π, let $lit(\Pi)$ be the set of literals that occur in Π, $seed(\Pi)$ the set of seed literals that occur in Π, and $body(\Pi)$ be the collection of the bodies of the rules of Π. Furthermore, let the expression $body(l)$ denote the set of bodies of the rules of Π whose head is l.

Given a rule's body $\beta = \{l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n\}$, the expression $\delta(\beta)$ denotes the nogood

$$\{\mathbf{F}\beta, \mathbf{T}l_1, \ldots, \mathbf{T}l_m, \mathbf{F}l_{m+1}, \ldots, \mathbf{F}l_n\}.$$

The expression $\Delta(\beta)$ denotes instead the set of nogoods

$$\{\{\mathbf{T}\beta, \mathbf{F}l_1\}, \ldots, \{\mathbf{T}\beta, \mathbf{F}l_m\}, \\ \{\mathbf{T}\beta, \mathbf{T}l_{m+1}\}, \ldots, \{\mathbf{T}\beta, \mathbf{T}l_n\}\}.$$

Next, given a literal l such that $body(l) = \{\beta_1, \ldots, \beta_k\}$, the expression $\Delta(l)$ denotes the set of nogoods

$$\{\{\mathbf{F}l, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}l, \mathbf{T}\beta_k\}\}.$$

Finally, the expression $\delta(l)$ denotes the nogood

$$\{\mathbf{T}l, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\}.$$

Given a program Π, let Δ_Π denote:

$$\begin{array}{l}\{\delta(\beta) \mid \beta \in body(\Pi)\} \cup \\ \{\delta \in \Delta(\beta) \mid \beta \in body(\Pi)\} \cup \\ \{\delta(l) \mid l \in seed(\Pi)\} \cup \\ \{\delta \in \Delta(l) \mid l \in lit(\Pi)\}.\end{array}$$

Intuitively, in Δ_Π, $\delta(l)$ is applied only to seed t-atoms because dependent t-literals do not occur in the head of rules.

It can be shown (Gebser et al. 2007) that Δ_Π can be used to find the answer sets of tight, t-literal free, programs. To find the answer sets of non-tight programs, one needs to introduce *loop nogoods*. For a program Π and some $U \subseteq lit(\Pi)$, expression $EB_\Pi(U)$ denotes the collection of the *external bodies* of U, i.e.

$$\{body(r) \mid r \in \Pi, head(r) \in U, body(r) \cap U = \emptyset\}.$$

Given a literal $l \in U$ and $EB_\Pi(U) = \{\beta_1, \ldots, \beta_k\}$, the *loop nogood* of l is

$$\lambda(l, U) = \{\mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k, \mathbf{T}l\}.$$

The set of loop nogoods for program Π is

$$\Lambda_\Pi = \bigcup_{U \subseteq lit(\Pi), U \neq \emptyset} \{\lambda(l, U) \mid l \in U\}.$$

Using a similar result from (Gebser et al. 2007) it is not difficult to prove the following property:

Theorem 1 *For every ASP{f} program* Π that contains no dependent t-literals, $X \subseteq lit(\Pi)$ *is an answer set of* Π *iff* $X = A^T \cap lit(\Pi)$ *for a solution* A *for* $\Delta_\Pi \cup \Lambda_\Pi$.

Next, we introduce nogoods for the computation of the answer sets of programs containing dependent t-literals. Given a dependent t-atom l of the form $f = g$ (resp., $f \neq g$), a pair of seed t-atoms $f = v$ and $g = w$ formed from $\Sigma(\Pi)$ is a *satisfying pair for* l if $v = w$ (resp., $v \neq w$) and a *falsifying pair for* l otherwise. Let $\{\langle f = v_1, g = w_1 \rangle, \ldots \langle f = v_k, g = w_k \rangle\}$ be the set of satisfying pairs for l. The expression $\rho^+(l)$ denotes the set of nogoods

$$\{\{\mathbf{F}l, \mathbf{T}f = v_1, \mathbf{T}g = w_1\}, \ldots, \{\mathbf{F}l, \mathbf{T}f = v_k, \mathbf{T}g = v_k\}\}.$$

Let $\{\langle f = v_1, g = w_1 \rangle, \ldots \langle f = v_k, g = w_k \rangle\}$ be the set of falsifying pairs for l. The expression $\rho^-(l)$ denotes the set of nogoods

$$\{\{\mathbf{T}l, \mathbf{T}f = v_1, \mathbf{T}g = w_1\}, \ldots, \{\mathbf{T}l, \mathbf{T}f = v_k, \mathbf{T}g = v_k\}\}.$$

Intuitively the nogoods in $\rho^+(l)$ and $\rho^-(l)$ enforce the truth or falsity of a dependent t-literal when suitable seed t-atoms are true.

Finally, given a dependent t-literal l, let $terms(l)$ denote the set of terms that occur in l, and, for every term f that occurs in l, let $rel(f)$ denote the set of seed t-atoms of the form $f = v$ for some $v \in \mathcal{C}(\Pi)$. Intuitively $rel(f)$ is the set of seed t-atoms that are relevant to the value of term f. The expression $\kappa(l)$ denotes the set of nogoods

$$\bigcup_{f \in terms(l)} (\{\mathbf{T}l\} \cup \{\mathbf{F}s \mid s \in rel(f)\}).$$

Intuitively $\kappa(l)$ states that l cannot be true if one of its terms is undefined.

Let $dep(\Pi)$ be the set of dependent t-literals in a program Π. Θ_Π denotes:

$$\begin{array}{l}\{\rho^+(l) \mid l \in dep(\Pi)\} \cup \\ \{\rho^-(l) \mid l \in dep(\Pi)\} \cup \\ \{\kappa(l) \mid l \in dep(\Pi)\}.\end{array}$$

The following condition defines a (rather large) class of ASP{f} programs whose answer sets can be found using Θ_Π. Given a program Π, we say that Π contains a *t-loop* for seed t-atom l if, in the dependency graph for Π, there is a positive path from l to a t-literal l' such that $terms(l) \cap terms(l') \neq \emptyset$. A program containing a t-loop is for example

$$f = 2 \leftarrow f \neq 3.$$

In practice, for most domains from the literature there appear to be t-loop free encodings. The following result characterizes the answer sets of t-loop free ASP{f} programs.

Theorem 2 *For every t-loop free ASP{f} program* Π*,* $X \subseteq seed(\Pi)$ *is an answer set of* Π *iff* $X = A^T \cap seed(\Pi)$ *for a solution* A *for* $\Delta_\Pi \cup \Lambda_\Pi \cup \Theta_\Pi$.

From a high-level perspective, in the CLASP algorithm the answer sets of ASP programs are computed by iteratively (1) performing unit propagation on the nogoods for the program and (2) non-deterministically assigning a truth value to a signed literal. Unfortunately, performing unit propagation on the nogoods in Θ_Π is inefficient, because in the worst case sets $\rho^+(l)$ and $\rho^-(l)$ exhibit quadratic growth. However, the conditions expressed by those nogoods can be easily checked algorithmically. Let $\text{VALUE}(f, A)$ be a function that returns v if signed literal $\mathbf{T}f = v$ occurs in assignment A. Given A and a dependent t-literal $f = g$, unit propagation on $\rho^+(f = g)$ can be performed by checking if $\text{VALUE}(f, A)$ = $\text{VALUE}(g, A)$ and, if so, by adding $\mathbf{T}f = g$ to A. A similar approach applies to the unit propagation for the other elements of Θ_Π.

Using this technique, unit propagation on the nogoods of Θ_Π can be performed in constant time w.r.t. the number of seed t-atoms in the program. (The reader may be wondering about the cases such as the one in which the truth of $\mathbf{T}f = v$ together with $\text{VALUE}(f, A)$ can be used to infer $\text{VALUE}(g, A)$. It can be shown that support for this type of scenario can be dropped without affecting the soundness and completeness of the solver.)

Function fLOCALPROPAGATION(Π, ∇, A), shown below, iteratively augments the result of unit propagation from CLASP's function LOCALPROPAGATION(Π, ∇, A) with the unit-resulting dependent t-literals derived from Θ_Π by function LOCALPROPAGATION$_\Theta(\Pi, \nabla, B)$. The iterations continue until a fixpoint is reached. (Function LOCALPROPAGATION(Π, ∇, A) in CLASP computes a fixpoint of unit propagation by adding to assignment A the unit-resulting literals derived from nogoods in Δ_Π and in ∇.)

```
Function: fLOCALPROPAGATION
Input: program $\Pi$, set $\nabla$ of nogoods, assignment $A$
Output: an extended assignment and a set of nogoods
$U \leftarrow \emptyset$
loop
    $B \leftarrow$ LOCALPROPAGATION$(\Pi, \nabla, A)$
    $A \leftarrow$ LOCALPROPAGATION$_\Theta(\Pi, \nabla, B)$
    if $A = B$ then return $A$
```

The algorithm for nogood propagation from (Gebser et al. 2007) is modified by replacing the call to LOCALPROPAGATION by a call to fLOCALPROPAGATION.

```
Function: NOGOODPROPAGATION
Input: program $\Pi$, set $\nabla$ of nogoods, assignment $A$
Output: an extended assignment and a set of nogoods
$U \leftarrow \emptyset$
loop
    $A \leftarrow$ fLOCALPROPAGATION$(\Pi, \nabla, A)$
    if $\delta \subseteq A$ for some $\delta \in \Delta_\Pi \cup \nabla$ or TIGHT$(\Pi)$ then
        return$(A, \nabla)$
    else
        $U \leftarrow U \setminus A^F$
        if $U = \emptyset$ then $U \leftarrow$ UNFOUNDEDSET$(\Pi, A)$
        if $U = \emptyset$ then return$(A, \nabla)$
        else let $p \in U$ in
            $\nabla \leftarrow \nabla \cup \{\lambda(p, U)\}$
            if $\mathbf{T}p \in A$ then return$(A, \nabla)$
            else $A \leftarrow A \circ (\mathbf{F}p)$
```

The main algorithm of CLASP{f}, shown below, is a straightforward modification of algorithm CDNL-ASP from (Gebser et al. 2007). For a description of the auxiliary functions, we refer the reader to (Gebser et al. 2007).

```
Function: CLASP{f}
Input: program $\Pi$
Output: an answer set of $\Pi$
$A \leftarrow \emptyset$
$\nabla \leftarrow \emptyset$
$dl \leftarrow 0$
loop
    $(A, \nabla) \leftarrow$ NOGOODPROPAGATION$(\Pi, \nabla, A)$
    if $\epsilon \subseteq A$ for some $\epsilon \in \Delta_\Pi \cup \nabla$ then
        if $dl = 0$ then return no answer set
        $(\delta, \sigma_{UIP}, k) \leftarrow$ CONFLICTANALYSIS$(\epsilon, \Pi, \nabla, A)$
        $\nabla \leftarrow \nabla \cup \{\delta\}$
        $A \leftarrow A \setminus \{\sigma \in A \,|\, k < dl(\sigma)\}$
        $dl \leftarrow k$
        $A \leftarrow A \circ (\overline{\sigma_{UIP}})$
    else if $A^T \cup A^F = lit(\Pi) \cup body(\Pi)$ then
        return $A^T \cap seed(\Pi)$
    else
        $\sigma_d \leftarrow$ SELECT$(\Pi, \nabla, A)$
        $dl \leftarrow dl + 1$
        $A \leftarrow A \circ (\sigma_d)$
```

Experimental Results

To evaluate the performance of the CLASP{f} algorithm, we have compared it with the method for computing the answer sets of programs with non-Herbrand functions used in (Cabalar 2011) and (Lifschitz 2011). In that method, given a program Π with non-Herbrand functions, (1) all occurrences of t-literals are replaced by regular ASP literals (e.g. $f = g$ is replaced by $eq(f, g)$), and (2) suitable equality and inequality axioms are added to Π. The answer sets of the resulting program are then computed using an ASP solver. It can be shown that the answer sets of the translation encode the answer sets of Π.

For our comparison we have chosen a planning task in which an agent starts at $(0, 0)$ on a $n \times n$ grid and has the goal of reaching a given position in k steps. The agent can move either up or to the right, by one cell at a time. Concurrent actions are not allowed. To make the task more challenging, the goal position is chosen so that the minimum number of actions needed to achieve the goal is equal to number of steps k. This domain has been selected because, in our experience on practical applications of ASP, solver performance decreases rapidly when parameter n is increased. This decrease in performance is due to the growth in the size of the grounding of the inertia axiom, and we are aware of no general-purpose technique to alleviate this issue in ASP programs.

The ASP{f} formalization, $\Pi_{\text{ASP\{f\}}}$ is show below. Constants k and n are specified at run-time. Symbol / used in the second-to-last rule denotes integer division in the dialect of CLASP (e.g. $3/2 = 1$).

$$\begin{aligned}
&step(0..k).\ loc(0..n-1).\\
&posx(0) = 0.\ posy(0) = 0.\\[1em]
&posx(S+1) = X+1 \leftarrow\\
&\quad step(S),\ step(S+1),\ loc(X),\ loc(X+1),\\
&\quad posx(S) = X,\ o(plusx, S).\\[1em]
&\leftarrow o(plusx, S), posx(S) = n-1.\\[1em]
&posy(S+1) = Y+1 \leftarrow\\
&\quad step(S),\ step(S+1),\ loc(Y),\ loc(Y+1),\\
&\quad posy(S) = Y,\ o(plusy, S).\\[1em]
&\leftarrow o(plusy, S), posy(S) = n-1.\\[1em]
&posx(S+1) = X \leftarrow\\
&\quad step(S),\ step(S+1),\ loc(X),\\
&\quad posx(S) = X,\ not\ posx(S+1) \neq posx(S).\\[1em]
&posy(S+1) = Y \leftarrow\\
&\quad step(S),\ step(S+1),\ loc(Y),\\
&\quad posy(S) = Y,\ not\ posy(S+1) \neq posy(S).\\[1em]
&1\{o(plusx, S), o(plusy, S)\}1 \leftarrow step(S),\ S < k.\\[1em]
&goal \leftarrow posx(k) = k/2,\ posy(k) = k - k/2.\\
&\leftarrow not\ goal.
\end{aligned}$$

The formalizations in the languages of (Cabalar 2011) and (Lifschitz 2011) are similar, and their translation to ASP (modulo renaming and reification of relations), denoted by Π_{ASP}, is:

$$\begin{aligned}
&step(0..k).\ loc(0..n-1).\\
&posx(0,0).\ posy(0,0).\\[1em]
&posx(S+1, X+1) \leftarrow\\
&\quad step(S),\ step(S+1),\ loc(X),\ loc(X+1),\\
&\quad posx(S, X),\ o(plusx, S).\\[1em]
&\leftarrow o(plusx, S), posx(S, n-1).\\[1em]
&posy(S+1, Y+1) \leftarrow\\
&\quad step(S),\ step(S+1),\ loc(Y),\ loc(Y+1),\\
&\quad posy(S, Y),\ o(plusy, S).\\[1em]
&\leftarrow o(plusy, S), posy(S, n-1).\\[1em]
&\neg posx(S, X2) \leftarrow\\
&\quad step(S),\ loc(X1),\ loc(X2),\\
&\quad X1 \neq X2,\ posx(S, X1).\\[1em]
&\neg posy(S, Y2) \leftarrow\\
&\quad step(S),\ loc(Y1),\ loc(Y2),\\
&\quad Y1 \neq Y2, posy(S, Y1).\\[1em]
&posx(S+1, X) \leftarrow\\
&\quad step(S),\ step(S+1),\ loc(X),\\
&\quad posx(S, X),\ not\ \neg posx(S+1, X).\\[1em]
&posy(S+1, Y) \leftarrow\\
&\quad step(S),\ step(S+1),\ loc(Y),\\
&\quad posy(S, Y),\ not\ \neg posy(S+1, S).\\[1em]
&1\{o(plusx, S), o(plusy, S)\}1 \leftarrow step(S),\ S < k.\\[1em]
&goal \leftarrow posx(k, k/2),\ posy(k, k - k/2).\\
&\leftarrow not\ goal.
\end{aligned}$$

Figure 1 shows a comparison of the time, in seconds, to find one answer set using $\Pi_{\text{ASP\{f\}}}$ and using Π_{ASP}. The results have been obtained for various values of parameters k and n. As the table shows, the time for $\Pi_{\text{ASP\{f\}}}$ is consistently more than an order of magnitude better than of Π_{ASP}, even though the code for the support of non-Herbrand functions in the implementation of CLASP{f} is still largely unoptimized. The CLASP{f} solver used here is an extension of CLINGO 2.0.2, and can be downloaded from `http://marcy.cjb.net/clingof/`. To ensure the fairness of the comparison, the answer sets of the ASP encoding have been computed using CLINGO 2.0.2. The experiments were performed on a computer with an Intel Q6600 processor at 2.4GHz, 1.5GB RAM and Linux Fedora Core 11.

Conclusions and Future Work

In this paper we have defined the syntax and semantics of an extension of ASP by non-Herbrand functions. Although the semantics of our language is a minimal modification of the semantics of ASP from (Gelfond and Lifschitz 1991), it allows for an efficient implementation in ASP solvers, as demonstrated by our experimental comparison with the solving techniques for other languages supporting non-Herbrand functions. Although the language of (Lin and Wang 2008; Wang et al. 2009) is also supported by an efficient solver, that solver uses CSP solving techniques rather than ASP solving techniques. Our motivation in developing a language that could be supported by an efficient ASP-based solver stems from the fact, demonstrated in the literature, that the relative performance of CSP and ASP solving algorithms varies depending on the domain of application, and so the availability of an efficient ASP-based solver is important.

Currently, the solving algorithm described in this paper is only applicable to a (large) class of ASP{f} programs. We expect that it will be possible to develop nogoods similar to the loop nogoods of CLASP and extend our algorithm to arbitrary ASP{f} programs.

Acknowledgments. The author would like to thank Michael Gelfond, Daniela Inclezan and the anonymous reviewers for their useful comments and suggestions.

$k = 3$		
n	$\Pi_{\text{ASP\{f\}}}$	Π_{ASP}
100	0.000	0.045
200	0.016	0.282
500	0.115	1.919
1000	0.513	8.273
1500	1.203	21.300
2000	2.429	43.092

$k = 5$		
n	$\Pi_{\text{ASP\{f\}}}$	Π_{ASP}
100	0.011	0.063
200	0.044	0.467
500	0.212	3.149
1000	1.012	13.787
1500	2.515	37.024
2000	4.283	70.591

$k = 7$		
n	$\Pi_{\text{ASP\{f\}}}$	Π_{ASP}
100	0.018	0.108
200	0.076	0.555
500	0.458	4.530
1000	1.766	21.432
1500	4.626	56.341
2000	7.712	103.737

Figure 1: Performance comparison between $\Pi_{\text{ASP\{f\}}}$ + CLASP{f} and Π_{ASP} + CLINGO.

References

Baral, C. 2003. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press.

Baselice, S., and Bonatti, P. A. 2010. A Decidable Subclass of Finitary Programs. *Journal of Theory and Practice of Logic Programming (TPLP)* 10(4–6):481–496.

Brewka, G.; Niemelä, I.; and Syrjänen, T. 2004. Logic Programs wirh Ordered Disjunction. 20(2):335–357.

Cabalar, P. 2011. Functional Answer Set Programming. *Journal of Theory and Practice of Logic Programming (TPLP)* 11:203–234.

Calimeri, F.; Cozza, S.; Ianni, G.; and Leone, N. 2010. Enhancing ASP by Functions: Decidable Classes and Implementation Techniques. In *Proceedings of the Twenty-Fourth Conference on Artificial Intelligence*, 1666–1670.

Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007. Conflict-Driven Answer Set Solving. In Veloso, M. M., ed., *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, 386–392.

Gebser, M.; Ostrowski, M.; and Schaub, T. 2009. Constraint Answer Set Solving. In *25th International Conference on Logic Programming (ICLP09)*, volume 5649.

Gelfond, M., and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing* 9:365–385.

Goldberg, E., and Novikov, Y. 2002. BerkMin: A Fast and Robust Sat-Solver. In *Proceedings of Design, Automation and Test in Europe Conference (DATE-2002)*, 142–149.

Lifschitz, V. 2011. Logic Programs with Intensional Functions (Preliminary Report). In *ICLP11 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP11)*.

Lin, F., and Wang, Y. 2008. Answer Set Programming with Functions. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR2008)*, 454–465.

Marek, V. W., and Truszczynski, M. 1999. *The Logic Programming Paradigm: a 25-Year Perspective*. Springer Verlag, Berlin. chapter Stable Models and an Alternative Logic Programming Paradigm, 375–398.

Syrjänen, T. 2001. Omega-Restricted Logic Programs. In Eiter, T.; Faber, W.; and Truszczynski, M., eds., *6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR01)*, volume 2173 of *Lecture Notes in Artificial Intelligence (LNCS)*, 267–279. Springer Verlag, Berlin.

Wang, Y.; You, J.-H.; Yuan, L.-Y.; and Zhang, M. 2009. Weight Constraint Programs with Functions. In Erdem, E.; Lin, F.; and Schaub, T., eds., *10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR09)*, volume 5753 of *Lecture Notes in Artificial Intelligence (LNCS)*, 329–341. Springer Verlag, Berlin.