An Answer Set Solver for non-Herbrand Programs: Progress Report

Marcello Balduccini[1]

1 **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. Our approach stems for 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 [8]. This 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.

1998 ACM Subject Classification I.2.4 Knowledge Representation Formalisms and Methods

Keywords and phrases Answer Set Programming, non-Herbrand Functions, Answer Set Solving, Knowledge Representation and Reasoning

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.49

1 Introduction

In this paper we describe an extension of Answer Set Programming (ASP) [8, 12, 2] 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. [5, 3, 13]).

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

$$occupancy(S+1) = O+1 \leftarrow occupancy(S) = O$$

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

Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 49–60

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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.

Various extensions of ASP with non-Herbrand functions exist in the literature. In [4], 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. [10] 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 [11, 14] the semantics is based on the notion of reduct as in the original ASP semantics [8]. For the purpose of computing answer sets, a translation is defined, which maps programs of the language from [11, 14] 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 [7] 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 [11, 14] certainly allow for an efficient computation of answer sets using constraint solving techniques, as demonstrated by the experimental results in [14]. On the other hand, the recent successes of CDCL-based solvers (see e.g. [9]) such as CLASP [6] 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 [4], the requirement made in [11, 14] 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 arguments also holds for

CLINGCON. An additional limitation of CLINGCON is the fact that the interleaved computation it performs carries some overhead.

In both [4] (where functions are partial) and [10] (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 [4], 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 [4] as an example (the transformation in [10] 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 are defined for x and have different values. 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 [8]. 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.

2 The Syntax of ASP{f}

In this section we define the syntax of ASP{f}. To keep the presentation simple, in this paper the version of ASP{f} described here 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:

$r_1 : occupancy = 0 \leftarrow room\_evacuated,\ not\ door\_stuck.$
$r_2 : room\_occupied \leftarrow occupancy \neq 0.$
$r_3 : room\_maybe\_occupied \leftarrow not\ occupancy = 0.$

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 $\mathcal{C}$. 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.

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. It is worth noting that the semantics of ASP{f} is obtained from that of ASP in [8] 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 $\{p \leftarrow f = 2. \qquad f = 2. \qquad q \leftarrow q.\}$ has one answer sets, $\{f = 2, p\}$. 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 $\{f = 3 \leftarrow \mathit{not}\ p. \qquad f = 2 \leftarrow \mathit{not}\ q.\}$ 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 $\mathit{head}(r) \leftarrow \mathit{pos}(r)$ (the *reduct* of r w.r.t. S) for each rule $r \in \Pi$ for which $S \models \mathit{body}(r) \setminus \mathit{pos}(r)$.

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

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

and let us compute its reduct. For r_1, first we have to check if $S_3 \models \mathit{body}(r_1) \setminus \mathit{pos}(r_1)$, that is if $S_3 \models \mathit{not}\ g = 1, \mathit{not}\ h = 0$. Extended literal $\mathit{not}\ g = 1$ 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 $\mathit{not}\ g = 1$ is satisfied by S_3. In a similar way, we conclude that $S_3 \models \mathit{not}\ h = 0$. Hence, $S_3 \models \mathit{body}(r_1) \setminus \mathit{pos}(r_1)$. Therefore, the reduct of r_1 is $p \leftarrow f = 2$. For the reduct of r_2, notice that $\mathit{not}\ g \neq 2$ is not satisfied by S_3. In fact, $S_3 \models \mathit{not}\ g \neq 2$ 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, $\mathit{val}_{S_3}(g)$ is defined and $\mathit{val}_{S_3}(g) \neq 2$. Therefore, $\mathit{not}\ g \neq 2$ 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 $\{r'_1 : p \leftarrow f = 2,\ r'_3 : g = 3,\ r'_4 : f = 2\}$

Finally, a consistent set S of seed literals is an *answer set* of 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.

4 Knowledge Representation with ASP{f}

In this section we demonstrate the use of ASP{f} for the formalization of key types of knowledge. 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 similar to, and inspired by, one from [10]). These statements

can be encoded in ASP{f} by $P_1 = \{r_1 : f(x) = a \leftarrow not\ f(x) \neq a.,\ r_2 : f(x) = b \leftarrow p(x).\}$. 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".

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 [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 $\{r_1 : f(X) = a \leftarrow p(X),\ dom(X).,\ r_2 : f(X) = V \leftarrow dom(X),\ val(V),\ not\ p(X),\ not\ f(X) \neq V.\}$. Rule r_1 encodes the first statement. Rule r_2 formalizes the arbitrary selection of values for $f(X)$ in the default case.

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:

$$r_1 : val(c, S+1) = 0 \leftarrow pressed(b_r, S).$$
$$r_2 : val(c, S+1) = N+1 \leftarrow pressed(b_i, S),\ val(c, S) = N.$$
$$r_3 : val(c, S+1) = N \leftarrow val(c, S) = N,\ not\ val(c, S+1) \neq val(c, S).$$

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.

Formal results about ASP{f} that are useful for knowledge representation tasks can be found in [1].

5 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 [6], the approach can be easily extended to other ASP solvers. In our description we follow the notation of [6], 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 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, $\delta(l) = \{\mathbf{T}l, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\}$.

Given a program Π, let Δ_Π denote $\{\{\delta(\beta) \mid \beta \in body(\Pi)\} \cup\ \{\delta \in \Delta(\beta) \mid \beta \in body(\Pi)\}\ \cup\ \{\delta(l) \mid l \in seed(\Pi)\} \cup\ \{\delta \in \Delta(l) \mid l \in lit(\Pi)\}\}$. 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 [6] 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 $\Lambda_\Pi = \bigcup_{U \subseteq lit(\Pi), U \neq \emptyset} \{\lambda(l, U) \mid l \in U\}$. The following property follows from a similar result from [6]:

▶ **Theorem 3.** *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-literal 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 $\{\rho^+(l) \mid l \in dep(\Pi)\} \cup\ \{\rho^-(l) \mid l \in dep(\Pi)\} \cup\ \{\kappa^-(l) \mid l \in dep(\Pi)\}$.

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 programs.

▶ **Theorem 4.** *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 $\textsc{value}(f, A)$ 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 $\textsc{value}(f, A) = \textsc{value}(g, A)$ 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 $\textsc{value}(f, A)$ can be used to infer $\textsc{value}(g, A)$. It can be shown that support for this type of scenario can be dropped without affecting the soundness and completeness of the solver.)

Function $\textsc{fLocalPropagation}(\Pi, \nabla, A)$, shown below, iteratively augments the result of unit propagation from CLASP's function $\textsc{LocalPropagation}(\Pi, \nabla, A)$ with the unit-resulting dependent t-literals derived from Θ_Π. The iterations continue until a fixpoint is reached. (Function $\textsc{LocalPropagation}(\Pi, \nabla, A)$ 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 Π, set ∇ of nogoods, assignment A
Output: an extended assignment and a set of nogoods
$U \leftarrow \emptyset$
loop
  $B \leftarrow \textsc{LocalPropagation}(\Pi, \nabla, A)$
  $A \leftarrow \textsc{LocalPropagation}_\Theta(\Pi, \nabla, B)$
  if $A = B$ then return A

The algorithm for nogood propagation from [6] is modified by replacing the call to LocalPropagation by a call to fLocalPropagation. The main algorithm of CLASP{f} is obtained in a similar way from algorithm CDNL-ASP from [6].

6 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 [4] and [10]. 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.

$$
\begin{array}{l}
step(0..k).\ \ loc(0..n-1).\ \ posx(0) = 0.\ \ posy(0) = 0. \\
posx(S+1) = X+1 \leftarrow \\
\qquad step(S),\ step(S+1),\ loc(X),\ loc(X+1),\ posx(S) = X,\ o(plusx, S). \\
\leftarrow o(plusx, S), posx(S) = n-1. \\
posy(S+1) = Y+1 \leftarrow \\
\qquad step(S),\ step(S+1),\ loc(Y),\ loc(Y+1),\ \ posy(S) = Y,\ o(plusy, S). \\
\leftarrow o(plusy, S), posy(S) = n-1. \\
posx(S+1) = X \leftarrow \\
\qquad step(S),\ step(S+1),\ loc(X),\ \ posx(S) = X,\ not\ posx(S+1) \neq posx(S). \\
posy(S+1) = Y \leftarrow \\
\qquad step(S),\ step(S+1),\ loc(Y),\qquad posy(S) = Y,\ not\ posy(S+1) \neq posy(S). \\
1\{o(plusx, S), o(plusy, S)\}1 \leftarrow step(S),\ S < k. \\
goal \leftarrow posx(k) = k/2,\ posy(k) = k - k/2. \\
\leftarrow not\ goal.
\end{array}
$$

Program Π_{ASP}, omitted to save space, is an ASP encoding of the problem obtained by the usual formalization techniques; it is also equivalent, modulo renaming and reification of relations, to the translation of the formalizations in the languages of [4] and [10]. Table 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. 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.

7 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 comparatively small modification of the semantics of ASP from [8], it allows for an efficient implementation in

Table 1 Performance comparison between $\Pi_{\mathrm{ASP}\{\mathrm{f}\}}$ + CLASP{f} and Π_{ASP} + CLINGO.

	$k=3$		$k=5$		$k=7$	
n	$\Pi_{\mathrm{ASP}\{\mathrm{f}\}}$	Π_{ASP}	$\Pi_{\mathrm{ASP}\{\mathrm{f}\}}$	Π_{ASP}	$\Pi_{\mathrm{ASP}\{\mathrm{f}\}}$	Π_{ASP}
100	0.000	0.045	0.011	0.063	0.018	0.108
200	0.016	0.282	0.044	0.467	0.076	0.555
500	0.115	1.919	0.212	3.149	0.458	4.530
1000	0.513	8.273	1.012	13.787	1.766	21.432
1500	1.203	21.300	2.515	37.024	4.626	56.341
2000	2.429	43.092	4.283	70.591	7.712	103.737

ASP solvers, as demonstrated by our experimental comparison with the solving techniques for other languages supporting non-Herbrand functions. Although the language of [11, 14] is also supported by an efficient solver, that solver uses CSP solving techniques rather than ASP solving techniques. Currently, the ASP{f} solving algorithm is only applicable to a (large) subclass of ASP{f} programs. We expect that it will be possible to extend our algorithm to arbitrary programs by introducing additional nogoods.

References

1 Marcello Balduccini. *Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, chapter 3. A "Conservative" Approach to Extending Answer Set Programming with Non-Herbrand Functions, pages 23–39. Lecture Notes in Artificial Intelligence (LNCS). Springer Verlag, Berlin, Jun 2012.

2 Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, Jan 2003.

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

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

5 Francesco Calimeri, Susanna Cozza, Giovanbattista Ianni, and Nicola Leone. Enhancing ASP by Functions: Decidable Classes and Implementation Techniques. In *Proceedings of the Twenty-Fourth Conference on Artificial Intelligence*, pages 1666–1670, 2010.

6 Martin Gebser, Benjamin Kaufmann, Andre Neumann, and Torsten Schaub. Conflict-Driven Answer Set Solving. In Manuela M. Veloso, editor, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392, 2007.

7 Martin Gebser, Max Ostrowski, and Torsten Schaub. Constraint Answer Set Solving. In *25th International Conference on Logic Programming (ICLP09)*, volume 5649, 2009.

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

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

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

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

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

13 Tommi Syrjänen. Omega-Restricted Logic Programs. In Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczynski, editors, *6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR01)*, volume 2173 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 267–279. Springer Verlag, Berlin, 2001.

14 Yisong Wang, Jia-Huai You, Li-Yan Yuan, and Mingyi Zhang. Weight Constraint Programs with Functions. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR09)*, volume 5753 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 329–341. Springer Verlag, Berlin, Sep 2009.