ASP with non-Herbrand Partial Functions: a Language and System for Practical Use

MARCELLO BALDUCCINI
Eastman Kodak Company
(*e-mail:* marcello.balduccini@gmail.com)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Dealing with domains involving substantial quantitative information in Answer Set Programming (ASP) often results in cumbersome and inefficient encodings. Hybrid "CASP" languages combining ASP and Constraint Programming aim to overcome this limitation, but also impose inconvenient constraints – first and foremost that quantitative information must be encoded by means of total functions. This goes against central knowledge representation principles that contribute to the power of ASP, and makes the formalization of certain domains difficult. ASP{f} is being developed with the ultimate goal of providing scientists and practitioners with an alternative to CASP languages that allows for the efficient representation of qualitative and quantitative information in ASP without restricting one's ability to deal with incompleteness or uncertainty. In this paper we present the latest outcome of such research: versions of the language and of the supporting system that allow for practical, industrial-size use and scalability. The applicability of ASP{f} is demonstrated by a case study on an actual industrial application.
Keywords: Answer Set Programming, non-Herbrand Functions, Constraint Answer Set Programming, Practical Applications

1 Introduction

Attempts to use Answer Set Programming (ASP) (Gelfond and Lifschitz 1991; Marek and Truszczynski 1999) to formalize domains involving a substantial amount of quantitative information often result in cumbersome, error-prone and inefficient encodings. Hybrid "CASP" (Lierler 2012) languages combining ASP and Constraint Programming (Gebser et al. 2009; Balduccini 2009; Lin and Wang 2008) overcome this limitation, but also impose rather inconvenient constraints – first and foremost that quantitative information must be encoded by means of total functions. This goes against central knowledge representation principles that contribute to the power of ASP, and complicates the formalization of certain domains such as dynamic domains and domains involving uncertain or incomplete inputs, which appear often in practical applications. Some languages that overcome these limitations have been developed (Cabalar 2011), but these studies have focused on the knowledge representation aspects of the issue. For the implementation, these approaches rely on a translation to ASP, and thus remain as inefficient as ASP in dealing with quantitative information.

We have developed the language of ASP with non-Herbrand functions (ASP{f}) (see e.g. (Balduccini 2012a)) with the ultimate goal of providing scientists and practitioners with a language and a solver that allows for an efficient representation of quantitative information without the restrictions imposed by CASP languages. Our work has also been motivated by the desire to in-

vestigate to what extent an ASP solver can deal effectively with a combination of qualitative and quantitative information without being coupled with a Constraint Programming solver.

In our earlier work, we have defined a first, simple version of the ASP{f} language and solver, and demonstrated the improvements over ASP over toy domains. That version of language and solver was intended as a proof-of-concept, and was too simple for serious practical use. In this paper we present the outcome of the next installment of the research on this topic: versions of the language and of the supporting system that are geared towards practical, industrial-size use and scalability. We also demonstrate the practical advantages of ASP{f} by reporting the results of a case study on an actual industrial application.

2 Language Definition

In this section we define the syntax and the semantics of the latest version of ASP{f}, which expand upon (Balduccini 2012a). Examples of the use of ASP{f} for knowledge representation are given in the online appendix.

Syntax. The syntax of ASP{f} is based on a signature $\Sigma = \langle \mathcal{C}, \mathcal{F}, \mathcal{R} \rangle$ whose elements are, respectively, finite disjoint sets of *constants*, *function symbols* and *relation symbols*. Some constants and function symbols are numerical (e.g. constants 1 and 5) and have the standard interpretation.[1] A *simple (non-Herbrand) term* is an expression $f(c_1, \ldots, c_n)$ where $f \in \mathcal{F}$, and c_i's are 0 or more constants. Intuitively, constants from $\mathcal{C}$ are given a Herbrand interpretation, while simple terms (possibly 0-ary) are given a non-Herbrand interpretation. Traditional Herbrand terms, if needed, can be included in $\mathcal{C}$ as atomic constants. An *arithmetic term* is either a simple term where f is a numerical function, or an expression constructed from simple terms and numerical constants using arithmetic operations, such as $(f(c_1) + g(c_2))/2$ and $|f(c_1) - g(c_2)|$. 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 simple terms that can be formed from Σ is denoted by $\mathcal{S}$; the set of all atoms from Σ is denoted by $\mathcal{A}$. A *regular literal* is an atom a or its strong negation $\neg a$. A *simple n-atom* (n-atom stands for *non-Herbrand atom*), is an expression of the form $f \circ\!\!\mathrm{p}\ g$, where f and g are simple and/or arithmetic terms and $\circ\!\!\mathrm{p} \in \{=, \neq, \leq, <, >, \geq\}$ ($=$ and $\neq$ enjoy the symmetric property).

Next, we introduce the notion of aggregate terms with a syntax similar to the constructs found in the language of GRINGO (Gebser et al. 2009) and in ASP-Core-2.[2] An *aggregate element* is an expression of the form

$$l_1 : \ldots : l_m : \mathit{not}\ l_{m+1} : \ldots : \mathit{not}\ l_n,$$

where every l_i is a regular literal or a simple n-atom. An *aggregate term* is an expression of the form

$$aggr[e_1 = t_1, e_2 = t_2, \ldots, e_n = t_n]$$

where $aggr \in \{min, max, sum\}$, e_i's are aggregate elements, and t_i's are arithmetic terms (often called *weights*). Within each argument of an aggregate term, both e_i and t_i are optional: an expression t_i is considered to be an abbreviation of $true = t_i$ and an expression e_i stands for $e_i = 1$. Furthermore, an expression of the form $count\{e_1, e_2, \ldots, e_n\}$ is an abbreviation of $sum[e_1 = 1, e_2 = 1, \ldots, e_n = 1]$. Simple terms, arithmetic terms and aggregate terms are called *terms*.

[1] In the rest of the paper, whether an element of Σ is numerical will be clear from the context.
[2] https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03b.pdf

An *n-atom* is an expression of the form $f \ \mathtt{op} \ g$, where f and g are simple terms, arithmetic terms, or aggregate terms and $\mathtt{op}$ is a connective as in simple n-atoms. For example, $f(x) = sum[p(x) : d(x) = g(x), q(x) : d(x) = h(x)]$, where $f(x), g(x), h(x)$ are simple terms and $p(x), q(x), d(x)$ are atoms, is an n-atom. Its intuitive meaning is that the value of $f(x)$ is $v_{g(x)} + v_{h(x)}$, where $v_{g(x)}$ is the value of $g(x)$ if $p(x)$ and $d(x)$ hold and 0 otherwise, and similarly for $v_{h(x)}$.

A *seed n-atom* is an n-atom of the form $f = v$ (or $v = f$) where f is a simple term and v is a constant. All other n-atoms are called *dependent*. So for example $f(x) = 3$ and $f(x) = a$, where $f(x)$ is a simple term and a is a constant, are seed n-atoms, while $f(x) = g(x)$, where $g(x)$ is a simple term, and $f(x) = sum[p(x) : d(x) = g(x), q(x) : d(x) = h(x)]$ from the above example are dependent n-atoms and not seed n-atoms.

Finally, a *literal* is either a regular literal or an n-atom. A *seed literal* is a regular literal or seed n-atom.

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

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

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

A *constraint* is a special type of regular rule with an empty head, informally meaning that the condition described by the body must never be satisfied. A constraint is considered to be a shorthand of $\bot \leftarrow l_1, \ldots, l_m, \mathit{not}\ l_{m+1}, \ldots, \mathit{not}\ l_n, \mathit{not}\ \bot$ 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 *n-atom free* if no n-atoms occur in the rules of Π.

Variables and Grounding. As in ASP, variables can be used in ASP{f} for a more compact notation. The *grounding of a rule* r is the set of all the rules (its *ground instances*) obtained by replacing every variable of r with an element[3] of $\mathcal{C}$ and by performing any arithmetic operation over numerical constants. For example, one of the groundings of $p(X + Y) \leftarrow r(X), q(Y)$ is $p(5) \leftarrow r(3), q(2)$. Variables within the scope of aggregate terms are grounded as in aggregate literals. For instance, if variable X has domain $\{0, 1, 2\}$, which we assume specified by domain predicate $d(\cdot)$ below, then an n-atom $f = sum[p(X) : d(X) = g(X)]$ is grounded as: $f = sum[p(0) = g(0), p(1) = g(1), p(2) = g(2)]$. It is important to note that, by construction of the language, arithmetic terms can be used directly as weights, which allows for the rather compact grounding shown above. Contrast this with a traditional aggregate literal $F = sum[p(X) : d(X) : g(X, V) = V]$, whose grounding must explicitly list all the possible values of V: $F = sum[p(0) : g(0, v_0) = v_0,\ p(0) : g(0, v_1) = v_1,\ \ldots, p(2) : g(2, v_0) = v_0,\ \ldots]$.

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 contains neither variables nor arithmetic operations over numerical constants and *non-ground* otherwise. For example, $p(5)$ is ground while $p(X+Y)$ and

[3] The replacement is with elements of suitable sorts.

$p(3+2)$ are non-ground. For more details on grounding, including the grounding of aggregates, we refer the reader to (Syrjänen 1998; Gebser et al. 2009).

Semantics. 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 $t \in \mathcal{S}$ 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.

The *value* of a simple 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. The value of an arithmetic term t w.r.t. S is obtained by applying the usual rules of arithmetic to the values of the terms in t w.r.t. S, suitably extended to deal with undefined values[4]. 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 $l \in S$. Any other simple n-atom $f \circ\!\!\mathrm{p}\ g$ is satisfied by S iff both $val_S(f)$ and $val_S(g)$ are defined, and they satisfy the equality or inequality relation $\circ\!\!\mathrm{p}$ according to the usual arithmetic interpretation. 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 n-atom $f \neq h$ is also not satisfied by S_2, because $val_{S_2}(h)$ is undefined.

An aggregate element $l_1 : \ldots : l_m$: *not* $l_{m+1} : \ldots :$ *not* l_n is satisfied by a consistent set S of seed literals iff, for every $1 \leq i \leq m$, l_i is satisfied by S and, for every $(m+1) \leq i \leq n$, l_i is not satisfied by S. The value of an aggregate term $aggr[e_1 = t_1, e_2 = t_2, \ldots, e_n = t_n]$ is obtained by applying to the multiset $\{val_S(t_i) \mid e_i$ satisfied by $S \wedge val_S(t_i)$ defined$\}$ the arithmetic interpretation of operator $aggr$. Note that, because of the availability of partial functions in the language, it is possible to capture in a natural way the fact that the value of $min(\{\})$ and $max(\{\})$ is undefined.

We complete the definition of the notion of satisfaction of arbitrary n-atoms by stating that an arbitrary dependent n-atom $f \circ\!\!\mathrm{p}\ g$ is satisfied by S iff both $val_S(f)$ and $val_S(g)$ are defined, and they satisfy the equality or inequality relation $\circ\!\!\mathrm{p}$ according to the usual arithmetic interpretation.

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 $e \in E$.

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 $r \in \Pi$.

[4] For example, if the value of f is undefined, $0 \cdot f$ is 0 but $0 + f$ is undefined.

Definition 1
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 $\{p \leftarrow f = 2.\ \ f = 2.\ \ q \leftarrow q.\}$ has one answer sets, $\{f = 2, p\}$. Set $\{f = 2\}$ is not closed under the first rule of the program, and therefore is not an answer set. 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 $\{f = 3.\ \ f = 2 \leftarrow q.\ \ q.\}$ has no answer set. Programs that have answer sets (resp., no answer sets) are called *consistent* (resp., *inconsistent*). It is not difficult to prove that positive programs have at most one answer set. Next, we define the semantics of arbitrary ASP{f} programs.

Definition 2
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= $\{(r_1)\ p \leftarrow f = 2, \mathit{not}\ g = 1, \mathit{not}\ h = 0.\ \ (r_2)\ q \leftarrow p, \mathit{not}\ g \neq 2.\ \ (r_3)\ g = 3.\ \ (r_4)\ f = 2.\}$ 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 $S_3 \models \mathit{not}\ g = 1, \mathit{not}\ h = 0$. 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, $S_3 \models body(r_1) \setminus pos(r_1)$. Therefore, the reduct of r_1 is $p \leftarrow f = 2$. For the reduct of r_2, notice that $\mathit{not}\ g \neq 2$ 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, $val_{S_3}(g)$ is defined and $val_{S_3}(g) \neq 2$. 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 $\{(r'_1)\ p \leftarrow f = 2,\ (r'_3)\ g = 3,\ (r'_4)\ f = 2\}$.

Definition 3
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.

3 The ASP{f} Solver CLINGO{f}

In this section we describe CLINGO{f}, a solver for ASP{f}. CLINGO{f} is derived from CLINGO 2.0.2 and as such it includes both grounder and inference engine in a monolithic structure. In the rest of this section, by CLINGO we refer to version 2.0.2 of language and system.

In CLINGO{f}, the n-atom connectives $\{=, \neq, \leq, <, >, \geq\}$ are prefixed by $\#$ and written for example $\#=$. For typographical reasons, however, throughout this section we use the alternate

writing $\{=_{\#}, \neq_{\#}, \leq_{\#}, <_{\#}, >_{\#}, \geq_{\#}\}$. The signature of a program is implicitly defined from the program rules, as usual. The only exception is the definition of the non-Herbrand function symbols, which is accomplished by declarations of the form:

$$\text{\#nherb } f/n.$$

where f is a function symbol and $n \geq 0$ is its arity. If the same function symbol is to be used with different arities, multiple declarations must be provided. Expressions of the form $f(t_1, \ldots, t_n)$ where f/n does not occur in a #nherb declaration, are considered to be traditional Herbrand terms, and treated as they are in CLINGO. Moreover, any terms occurring out of the scope of an n-atom are treated as Herbrand terms. This allows for the use of common reification techniques. For example, the statements:

$$\begin{array}{l}\text{\#nherb } f/1.\\ \leftarrow is\_weight(f(x)), f(x) <_{\#} 0.\end{array}$$

allow one to concisely state that, if $f(x)$ encodes a weight, then its value must not be smaller than 0. Choice rules are allowed in CLINGO{f} with the usual syntax, extended to allow for the occurrence of seed n-atoms. The usual requirements from CLINGO also apply, both on the specification of domain predicates for variables and on the syntactic restrictions of choice rules and aggregate literals and aggregate terms.

Grounding. CLINGO{f} allows for the use of variables, and grounding follows the same approach used in CLINGO. Because of the scope-dependent interpretation of non-Herbrand terms discussed above, variables can also be used in place of non-Herbrand terms, allowing one to extend our reification example as follows:

$$\begin{array}{l}\text{\#nherb } f/1.\\ is\_weight(f(x)).\ is\_weight(f(y)). \qquad \neg is\_weight(f(z)). \qquad \leftarrow is\_weight(F), F <_{\#} 0.\end{array}$$

Now the last rule states that, *if F encodes a weight*, then its value must not be less than 0. Besides traditional variables, CLINGO{f} also allows for the use of a special kind of variable, aimed at reducing grounding size and increasing performance. Consider the program:

$$\begin{array}{l}\text{\#nherb } f/1.\\ d(0..100). \qquad 1\{f(x) =_{\#} X : d(X)\}1. \qquad f(y) =_{\#} X + 1 \leftarrow d(X), f(x) =_{\#} X.\end{array}$$

The choice rule states that $f(x)$ may have any value in the range $[0, 100]$. The last rule intuitively states that $f(y)$ is $f(x) + 1$. Although this specification is straightforward and concise, the size of the grounding of the last rule grows proportionally to the domain specified for X, because, at grounding time, no assumption can be made about the value of $f(X)$. For practical applications with more complex rules this is a bottleneck – sometimes up to the point of making the use of ASP impossible. In order to overcome this problem, CLINGO{f} introduces *non-Herbrand variables* (*n-variables*), represented by alphanumeric strings prefixed by _, such as $\_v$ and $\_x12$. N-variables are inspired by the simple observation that, *although in $f(x) =_{\#} X$ the value of $f(x)$ is not known at grounding time, $f(x)$ can only have one value at a time.* N-variables can be used in place of traditional variables, under certain restrictions, which we describe next.

We begin by observing that n-variables are considered ground expressions and thus are not affected by grounding. A *defining n-atom* for an n-variable ν is an n-atom of the form $\nu =_{\#} t$ (or $t =_{\#} \nu$), where t is a term. Given a ground rule r, we call an *index assignment* a function that associates a positive integer to every n-variable that occurs in r, and associates 0 to every

constant and simple term. The index assignment of a compound term is the maximum of the index assignments of the term's components. We say that rule r is *n-stratified* if there exists an index assignment i such that, for every n-variable ν in r, $pos(r)$ contains a defining n-atom $\nu =_{\#} t$ such that $i(\nu) > i(t)$.

We can now state the restrictions on the use of n-variables. N-variables can be used in place of traditional variables, as long as: (1) n-variables do not occur as arguments of simple terms or aggregate terms; (2) all rules are n-stratified.

Using n-variables, the above program can be re-written as:

#nherb $f/1$.
$d(0..100).$ $\quad 1\{f(x) =_{\#} X : d(X)\}1.$ $\quad f(y) =_{\#} \_x + 1 \leftarrow f(x) =_{\#} \_x.$

The last rule of this program is ground. Hence, the size of its grounding no longer varies with the domain specified by d. This may yield substantial performance improvements when rules involve a combination of variables with large domains. Next, we describe CLASP{f}, the inference engine used by CLINGO{f}. This version of the engine can be shown to be sound and complete for ASP{f} programs whose dependency graph does not contain positive paths connecting n-atoms where the same simple term occurs (see (Balduccini 2012a) for more details).

Inference Engine. In the description that follows, we assume that all the rules in the program have already been grounded with the usual techniques. For simplicity of presentation, we also assume that every n-variable occurs in at most one rule. To illustrate the algorithm, we will make use of the following program, Π_1:

#nherb $f/1, g/1, h/1$.
$(r_1)\ f(x) =_{\#} 3.$ $\quad (r_2)\ p \leftarrow f(x) >_{\#} 2.$ $\quad (r_3)\ h(x) =_{\#} \_v \leftarrow f(x) =_{\#} \_v.$
$(r_4)\ q \leftarrow g(x) \neq_{\#} 3.$ $\quad (r_5)\ s \leftarrow$ *not* $g(x) =_{\#} 2.$

The first step in the CLASP{f} algorithm consists in replacing all the occurrences of $=_{\#}$ in the body of the rules with a special connective $\equiv_{\#}$. The new connective has the same semantics of $=_{\#}$, but it allows for a decoupling of the n-atoms in the head and in the body of rules that is needed for the correct support of n-variables. Program Π_1 is then transformed into program Π_2, obtained from Π_1 by replacing r_5 by $(r_5')s \leftarrow$ *not* $g(x) \equiv_{\#} 2$.

The main loop of CLASP{f} is shown below. The algorithm is modeled after the CLASP algorithm, extended with an extra propagation step. The algorithm is summarized here in such a way that highlights the components that are most relevant to our description:

- Loop
  - — Repeat
    - – Perform unitPropagation
    - – Perform nonHerbrandPropagation
  - — Until no changes in truth value detected
  - — If inconsistency is detected, then backtrack
  - — If search is complete, then return answer set and terminate
  - — Otherwise, make a guess
- End Loop

The inconsistency check, the completeness check and the guess step are defined as in CLASP. We refer the reader to (Gebser et al. 2007) for a detailed description of those, and of the unitPropagation algorithm. In the scope of the present paper, it is sufficient to view unit-propagation as a

step in which inferences are made about the truth value of the literals in the program, based on the truth value of other literals and on the structure of the rules they occur in. In the CLASP{f} algorithm, literals can be either true, false, or undecided. The first two indicate that the algorithm has determined that the corresponding literal will occur (respectively, will not occur) in the answer set being constructed. If a literal is undecided, it means that no such determination has been made. The construction of an answer set is complete only when all the literals have been been made either true or false.

A property of unit-propagation that is important in the scope to the present paper is (here presented in simplified form): literals that do not occur in the head of any rule are made false. In CLASP{f}, this part of unitPropagation is modified so that it does not apply to n-atoms occurring in the body of rules. The rationale for this design decision is that the truth of such n-atoms is determined intensionally, using the values of the terms that occur in them. So, given program Π_2, the unitPropagation step will not conclude that e.g. $f(x) >_{\#} 2$ is false, although no rule with that n-atom in the head exists. The only truth value determined by unitPropagation at this stage is for $f(x) =_{\#} 3$, which is made true because the body if r_1 is trivially satisfied.

Next, nonHerbrandPropagation is performed. This is the algorithm that truly provides support for the language extensions of CLINGO{f}. As discussed in (Balduccini 2012a), nonHerbrandPropagation uses an intensional definition of the nogoods that implement the CLINGO{f} language elements. To save space, in this paper we omit listing the corresponding nogoods.

The algorithm maintains a table of the values currently assigned to the simple terms and to the n-variables that occur in the program. A value can be a constant or the special value $\langle$*undef*$\rangle$. The value of an arithmetic or aggregate term is computed from the values of the simple terms and constants that occur in it, as discussed in Section 2. The nonHerbrandPropagation algorithm consists of three steps:

- determine if any simple term can be assigned a value (including $\langle$*undef*$\rangle$);
- determine if any n-variable can be assigned a value (including $\langle$*undef*$\rangle$);
- determine the truth values of n-atoms in the bodies of rules from the values of their terms.

The determination of the values assigned to simple terms follows the rules:

- the value of a simple term t is $\langle$*undef*$\rangle$ if all the seed n-atoms for t in the head of the rules of the program are false;
- the value of a simple term t is v if there exists a seed n-atom $f =_{\#} v$ that is true;
- the value of a simple term t is v if there exists a seed n-atom $f =_{\#} \nu$ that is true, where ν is an n-variable assigned value v.

In the case of program Π_2, from the fact that $f(x) =_{\#} 3$ is true, the above rules determine that $f(x)$ must be assigned value 3. Because no seed n-atoms for $g(x)$ occur in the head of any rule, the first rule also concludes that the value of $g(x)$ is $\langle$*undef*$\rangle$. The determination of the values assigned to the n-variables is made as follows:

- n-variable ν occurring in rule r is assigned value v if, for every defining n-atom of the form $\nu =_{\#} t$ in $pos(r)$, the n-atom's right-hand-side has value v;
- n-variable ν occurring in rule r is assigned value $\langle$*undef*$\rangle$ if two defining n-atoms $\nu =_{\#} t_1$, $\nu =_{\#} t_2$ exist in $pos(r)$, such that the value of t_1 is different from the value of t_2.

Going back to program Π_2, from rule r_3 and the fact that $f(x)$ has value 3, it follows that n-variable $\_x$ also has value 3. Finally, the determination of the truth values of the n-atoms from the values of the terms occurring in them is performed as follows:

- an n-atom $f \circ\!\!\mathrm{p}\ g$ in the body of a rule is true if f and g are assigned values different from $\langle \textit{undef} \rangle$ and satisfy the usual interpretation of $\circ\!\!\mathrm{p}$ (where $\equiv_\#$ is interpreted as $=_\#$);
- an n-atom $f \circ\!\!\mathrm{p}\ g$ in the body of a rule is false if:
  - — f and g are assigned values different from $\langle \textit{undef} \rangle$ and do not satisfy the usual interpretation of $\circ\!\!\mathrm{p}$, or
  - — f and g are both assigned values, and the value of at least one of them is $\langle \textit{undef} \rangle$.

For program Π_2, it is not difficult to see that, because $g(x)$ has value $\langle \textit{undef} \rangle$, $g(x) \neq_\# 2$ and $g(x) \equiv_\# 2$ are false. On the other hand, $f(x) >_\# 2$ and $f(x) \equiv_\# \_v$ are true (the latter, because $\_v$ has value 3). The truth value of $h(x) =_\# \_v$ is not affected by the above rules because that n-atom occurs in the head. This completes the nonHerbrandPropagation step.

At this point, if the truth value of any literal has changed, unitPropagation is invoked again. In the case of Π_2, unitPropagation will detect that the body of r_4 is not satisfied, and thus make q false. On the other hand, the bodies of r_2, r_3 and r_5' are satisfied, and thus p, $h(x) =_\# \_v$ and s are made true. Finally, nonHerbrandPropagation is executed again. This time, from the fact that $h(x) =_\# \_v$ is true and that the value of $\_v$ is 3, it is concluded that the value of $h(x)$ is also 3. The construction of the answer set is now complete, and the algorithm terminates. For more complex programs, the algorithm would proceed to interleaving truth value guesses and backtracking as usual. Results from (Balduccini 2012b) also apply to the current version of language and solver:

Proposition 1
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.

Proposition 2 (*Splitting Set Theorem (Lifschitz and Turner 1994)*)
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.

Given a program Π, we say that Π contains an *n-loop* for seed n-atom l if, in the dependency graph for Π, there is a positive path from a to an n-atom a' such that a and a' share some non-Herbrand term. A program containing an n-loop is for example $f =_\# 2 \leftarrow f \neq_\# 3$. In practice, for most domains from the literature there appear to be straightforward n-loop free encodings.

Proposition 3
For every n-loop free ASP{f} program Π, CLINGO{f} is sound and complete.

4 A Case Study

In earlier work (Balduccini 2012a) we have conducted an experimental comparison of ASP{f} and ASP over toy domains. Although encouraging, those results were too limited to provide an assessment of scalability and practical use. Thus, in this section we demonstrate the performance improvements of ASP{f} over ASP on an actual industrial application. The application is being developed as part of a larger commercial product. In this scenario, an intelligent agent receives orders for the production and shipping of goods of various kinds and is expected to return corresponding cost estimates. Each order specifies the quantity and kind of good to be produced, the deadline by which the cost estimate must be sent to the customer, and the destination to which the produced goods must be shipped. The agent is also given a database of production units and

of their attributes. For each production unit, the database specifies the location of the production unit, needed to calculate shipping costs, and pricing tables for production and shipping. The production pricing table lists the goods that the given production unit is capable of producing. The table also specifies the per-item production cost. Because typically production units give discounts for bulk-quantity orders, the per-item production cost given by the table is a function of the order quantity. The shipping pricing table is organized in a similar way, except that the shipping cost returned is independent of the goods being shipped, and is instead a function of the total shipment weight and of the shipping destination. Notice that all the goods in a shipment must have the same destination.

The agent's goal is that of assigning each order to a production unit in a way that minimizes the total production and shipping cost of the orders on-hand. In order to do this, the agent is allowed to combine multiple orders and assign them to the same production unit. In doing so, the agent may achieve cost savings in one of two ways. If the orders are for the same type of goods, the agent may be able to access a better bulk-rate production cost, since what matters is the total quantity being ordered to a production unit. If the orders are for different types of goods, but have the same shipping destination, the agent may be able to access a better bulk-rate shipping cost, since what matters is the total weight shipped.

For every order, the agent is allowed to decide to calculate an estimate immediately, or to wait until more orders are received, as long as waiting does not violate the order's estimation deadline. To make this decision, the agent considers the set of orders currently on-hand, as well as information about the average orders received daily and their typical destinations. This information is used to form a projection of the amount of units on order over time, and to identify times in which the quantity and type of orders are "at peak" w.r.t. production, shipping, or both. Intuitively, production and shipping peaks are equally good, while a concurrent peak of both is the most desirable. Ideally, when making decisions about order assignments, the agent should identify the peak times of every order, and hold each order until the latest occurrence of its most favorable peak, unless otherwise instructed by a human operator. The orders that are at their most favorable peak (including orders whose estimates are due immediately), or manually identified by the operator, are the ones considered in the assignments to production units.

In this case study, we consider a simplified scenario in which the agent must (1) form projections for all the orders on-hand; (2) identify peaks in the projections; (3) determine which orders should be assigned and which orders should be held; (4) calculate the production and shipping costs for a fixed set of assignments of orders to production units (simulating the manual override by a human operator). In preliminary experiments, this instances had proven quite computationally demanding for ASP encodings, and motivated the initial work on ASP$\{$f$\}$. (The complex structure of this domain makes the development of a CASP encoding impractical.)

In our experiments, we have assessed the performance improvements yielded by the dedicated treatment of non-Herbrand functions in ASP$\{$f$\}$ by comparing our ASP$\{$f$\}$ encoding with one written in the dialect of GRINGO and run in CLINGO and CMODELS. Samples of the corresponding encodings are given in the online appendix.

The experiments were run on a computer with an Intel Core i7 processor at 3GHz with otherwise negligible CPU load. Memory usage was limited to 3 GB. The timeout was 600 seconds. The version of CLINGO used was 0.3.3. The version of CMODELS used was 3.83. The experiments consisted in the use of each solver and corresponding encoding to solve 60 problem instances. The instances were obtained by progressively elaborating a basic instance consisting of 4 orders, 4 good types and 3 production units. The instances were organized in 3 categories

with 20 instances each: in category C1, the quantity of goods in each order was progressively increased by 1, while the other parameters were kept constant; in category C2, the number of orders was progressively increased by 1; in category C3, the initial number of orders was 20 and was progressively increased by 4. These problem instances were designed to test the scalability of the encodings in conditions close to those expected in the actual use of the system.

The performance of the encodings was measured with respect to both time and memory usage. In all figures below, executions that resulted in out-of-memory are marked by "X". Overall, the ASP{f} encoding performed much better than the ASP encoding, and the performance difference *increased* with the difficulty of the instances, which indicates better scalability for ASP{f}. The results for category C1 are shown in Figure 1. Here and below, the numbers on the x-axis denote the problem instances. Depending on the figure, the y-axis shows either execution time or memory usage of each instance. As shown in Figure 1, the ASP{f} encoding was up to $1,000$ times faster than the performance for the ASP encoding with the best solver on the instances in which the ASP encoding returned a valid answer. In about half of the instances, the ASP encoding ran out of memory, but judging by the slopes of the performance curves, the difference is likely to increase even more if more memory is made available. A similar behavior was observed for category C2 (see Figure 2), where the ASP{f} encoding was more than 300 times faster than the ASP encoding on the instances for which the ASP encoding returned a valid answer. For the most difficult instances, the ASP encoding ran out of memory. In category C3 (see Figure 3), the ASP encoding ran out of memory for every instance, which is not surprisingly since the instances in this category are designed to be harder than the hardest instance from category C2. The time taken by the ASP encoding to run out of memory in CLINGO decreased as the instances became harder, which simply shows that CLINGO succeeded in filling up the allotted memory faster. Overall, the performance of the ASP{f} encoding was remarkably consistent w.r.t. both the time taken and the memory used. Throughout all the categories, time increased slowly as the instances became harder, with the solver taking only about 10 seconds on the hardest instance of the set. Memory usage also grew slowly, with a maximum usage of about 100 MB for the hardest instance. The results reported in this paper confirm the earlier findings from (Balduccini 2012a), and indicate that the use of ASP{f} yields both more efficient grounding (in execution time and in memory usage) and faster solving even in large, practical applications.

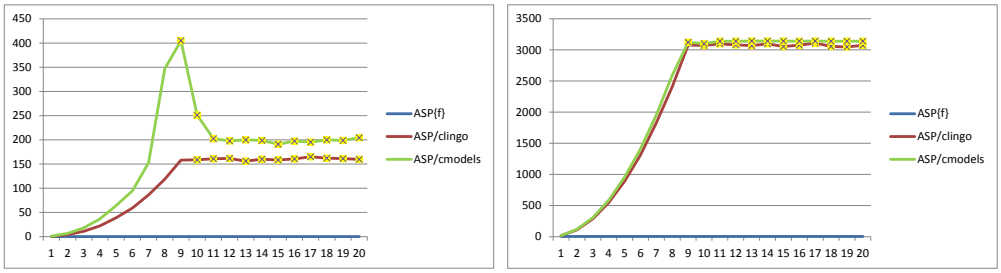

Figure 1. Category C1: time (left; in sec) and memory (right; in MB)

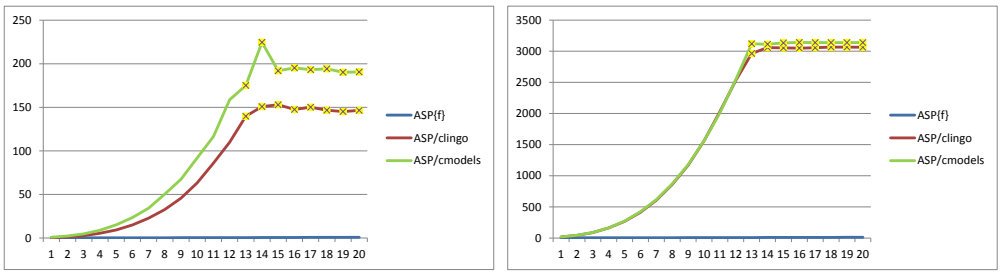

Figure 2. Category C2: time (left; in sec) and memory (right; in MB)

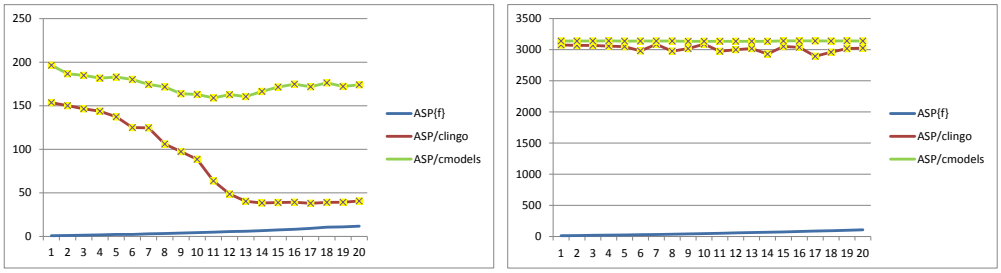

Figure 3. Category C3: time (left; in sec) and memory (right; in MB)

5 Conclusions

In this paper we have described versions of ASP{f} and of its solver that allow for practical, industrial-size use. This is a substantial improvement over earlier versions, which were proof-of-concepts designed for toy problems. ASP{f} is intended as an alternative to CASP languages, and is aimed for scientists and practitioners formalizing domains involving substantial quantitative information, but would like to have the ability, lacking in CASP languages, to represent incomplete and partial information in a direct and convenient way.

The performance of the solver, assessed on the actual industrial application that had initially motivated the development of ASP{f}, was extremely satisfactory from all points of view. This result not only confirms that ASP{f} is a viable language for the representation of hybrid qualitative-quantitative knowledge, but also appears to indicate that CLASP-style algorithms may in general be capable of dealing effectively with this kind of information without being coupled with a Constraint Programming solver. Of course, more work on algorithms and experimentation must be performed before such a broad claim can be made. The latest version of the solver can be downloaded from *http://www.mbalduccini.tk/clingof/*.

References

BALDUCCINI, M. 2009. Representing Constraint Satisfaction Problems in Answer Set Programming. In *ICLP09 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP09)*.

BALDUCCINI, M. 2012a. Answer Set Solving and Non-Herbrand Functions. In *Proceedings of the 14th International Workshop on Non-Monotonic Reasoning (NMR'2012)*, R. Rosati and S. Woltran, Eds.

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

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

GEBSER, M., KAMINSKI, R., OSTROWSKI, M., SCHAUB, T., AND THIELE, S. 2009. On the input language of ASP grounder gringo. In *10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR09)*, E. Erdem, F. Lin, and T. Schaub, Eds. Lecture Notes in Artificial Intelligence (LNCS), vol. 5753. Springer Verlag, Berlin, 502–508.

GEBSER, M., KAUFMANN, B., NEUMANN, A., AND SCHAUB, T. 2007. Conflict-Driven Answer Set Solving. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, M. M. Veloso, Ed. 386–392.

GEBSER, M., OSTROWSKI, M., AND SCHAUB, T. 2009. Constraint Answer Set Solving. In *25th International Conference on Logic Programming (ICLP09)*. Vol. 5649.

GELFOND, M. AND LIFSCHITZ, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing 9*, 365–385.

LIERLER, Y. 2012. On the Relation of Constraint Answer Set Programming Languages and Algorithms. In *Proceedings of the 26th Conference on Artificial Intelligence (AAAI'12)*. MIT Press.

LIFSCHITZ, V. AND TURNER, H. 1994. Splitting a logic program. In *Proceedings of the 11th International Conference on Logic Programming (ICLP94)*. 23–38.

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. 1998. Implementation of logical grounding for logic programs with stable model semantics. Tech. Rep. 18, Digital Systems Laboratory, Helsinki University of Technology.

Online appendix for the paper

ASP with non-Herbrand Partial Functions: a Language and System for Practical Use

published in Theory and Practice of Logic Programming

MARCELLO BALDUCCINI
Eastman Kodak Company
(*e-mail:* marcello.balduccini@gmail.com)

submitted 10 April 2013; revised 23 May 2013; accepted 23 June 2013

Appendix A Sample of ASP{f} and ASP Encodings from the Case Study

In order to give a sample of the encodings used in the case study, in this section we show key rules for dealing with production estimation. The rules used for dealing with shipping estimation are similar.

As typical, the main input to the system is described by facts. For example, the following ASP{f} facts describe an order, r_1, for 10 items of product p_1 with quoting deadline 13 (e.g. 13 days in the future) and destination l_{10} (of course the declaration needs to occur only once in the program).

#nherb $quantity/1$.
#nherb $product/1$.
#nherb $deadline/1$.
#nherb $destination/1$.

$rfq(r_1)$.
$product(r_1) =_{\#} p_1$.
$quantity(r_1) =_{\#} 10$.
$deadline(r_1) =_{\#} 13$.
$destination(r_1) =_{\#} l_{10}$.

For consistency with the terminology in use in the application domain, orders are identified by relation rfq, which stands for *request for quote*. The ASP encoding of the same information is:

$rfq(r_1)$.
$product(r_1, p_1 1)$.
$quantity(r_1, 10)$.
$deadline(r_1, 13)$.
$destination(r_1, l_{10})$.

Information about the average quantity of a product p on order per day can be specified by means of a fact $daily\_order(p) =_{\#} q$. If not specified, this value is assumed to be 0 in the ASP{f}

encoding by the rule

$$daily\_order(P) =_{\#} 0 \leftarrow product(P), \mathit{not}\ daily\_order(P) \neq_{\#} 0.$$

and in the ASP encoding by the rules:

$$daily\_order(P, 0) \leftarrow product(P), \mathit{not}\ \neg daily\_order(P, 0).$$
$$\neg daily\_order(P, 0) \leftarrow daily\_order(P, Q), Q \neq 0.$$

The determination of the production peaks is centered around the ASP{f} rule (some auxiliary atoms have been omitted here and below to simplify the presentation):

$$\begin{array}{l} at\_peak\_production(RFQ\_ID, D) \leftarrow \\ \quad rfq(RFQ\_ID), \\ \quad product(RFQ\_ID) =_{\#} P, \\ \quad deadline(RFQ\_ID) =_{\#} QD, \\ \quad D \leq QD, \\ \quad expected\_on\_order(P, D) =_{\#} max[expected\_on\_order(P, DAY_p) : DAY_p \leq QD]. \end{array}$$

The rule informally states that order RFQ_ID is at peak for production on day D if D is no later than the quoting deadline, RFQ_ID is for product P, and the expected quantity of product P on order on day D is equal to the maximum quantity of P expected to be on order on each day from now until the quoting deadline. The corresponding rule in the ASP encoding is:

$$\begin{array}{l} at\_peak\_production(RFQ\_ID, D) \leftarrow \\ \quad rfq(RFQ\_ID), \\ \quad product(RFQ\_ID, P), \\ \quad deadline(RFQ\_ID, QD), \\ \quad D \leq QD, \\ \quad expected\_on\_order(P, D, Q), \\ \quad Q = max[expected\_on\_order(P, DAY_p, Q_p) = Q_p : DAY_p \leq QD]. \end{array}$$

Function $expected\_on\_order$ is defined in the ASP{f} encoding by rules such as:

$$\begin{array}{l} expected\_on\_order(P, D) =_{\#} \\ \qquad\qquad \_today\_quantity - \_due\_quantity + M * \_daily\_order \leftarrow \\ \quad product(RFQ\_ID) =_{\#} P, \\ \quad current\_time(T), \\ \quad M = D - T, \\ \quad daily\_order(P) =_{\#} \_daily\_order, \\ \quad M <_{\#} avg\_deadline(P), \\ \quad \_today\_quantity =_{\#} sum[rfq(RFQ\_ID_p) : product(RFQ\_ID_p) =_{\#} P \\ \qquad\qquad = \ quantity(RFQ\_ID_p)], \\ \quad \_due\_quantity =_{\#} sum[due\_quantity(P, D') : D' < D]. \end{array}$$

This rule intuitively states that the quantity expected to be on order for product P on day D is obtained by subtracting, from the quantity on order now, the quantity due between now and day D, and then adding to this value the average daily orders for$P$ multiplied by the number of days in the period considered. (This formula is motivated by domain-specific considerations.)

The corresponding rule in the ASP encoding is:

$$
\begin{aligned}
&expected\_on\_order(P, D, TODAY\_Q - DUE\_Q + M * DAILY\_ORDER) \leftarrow \\
&\quad product(RFQ\_ID, P), \\
&\quad current\_time(T), \\
&\quad M = D - T, \\
&\quad daily\_order(P, DAILY\_ORDER), \\
&\quad avg\_deadline(P, AVG\_DLINE), \\
&\quad M < AVG\_DLINE, \\
&\quad TODAY\_PAGES = sum[quantity(RFQ\_ID_p, Q) = Q : rfq(RFQ\_ID_p) \\
&\qquad\qquad : product(RFQ\_ID_p, P)], \\
&\quad DUE\_PAGES = sum[due\_quantity(P, D', Q) = Q : D' < D].
\end{aligned}
$$

Appendix B Knowledge Representation in ASP{f}: an Overview

In this section we give an overview of how ASP{f} can be used for some classical knowledge representation tasks. An extended discussion can be found in (Balduccini 2012). More advanced knowledge representation topics are addressed in (Balduccini and Gelfond 2012).

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)). These statements can be encoded in ASP{f} as follows:

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

Rule r_1 encodes the default, and r_2 encodes the exception. The informal reading of r_1 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 can be encoded in ASP{f} by means of default negation. Consider the statements (again, 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:

$$
\begin{aligned}
&(r_1)\ f(X) = a \leftarrow p(X),\ dom(X). \\
&(r_2)\ f(X) = V \leftarrow dom(X),\ val(V), \textit{not } p(X),\ \textit{not } f(X) \neq V.
\end{aligned}
$$

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:

$$
\begin{aligned}
&(r'_1)\ f'(X) = a \leftarrow\ p(X),\ dom(X). \\
&(r'_2)\ f'(X, V) \leftarrow dom(X),\ val(V),\ \textit{not } p(X),\ \textit{not } \neg f'(X, V). \\
&(r'_3)\ \neg f'(X, V') \leftarrow val(V),\ val(V'),\ V \neq V',\ f'(X, V).
\end{aligned}
$$

The behavior of a dynamic domain consisting of a button b_i, which increments a counter c,

and a button b_r, which resets it, can be encoded in ASP{f} by:

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

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 fluent, but could be as easily written so that it applies to arbitrary fluents from the domain.

References

BALDUCCINI, M. 2012. Answer Set Solving and Non-Herbrand Functions. In *Proceedings of the 14th International Workshop on Non-Monotonic Reasoning (NMR'2012)*, R. Rosati and S. Woltran, Eds.

BALDUCCINI, M. AND GELFOND, M. 2012. Language ASPf with Arithmetic Expressions and Consistency-Restoring Rules. In *ICLP12 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP12)*.

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