Language ASP{f} with Arithmetic Expressions and Consistency-Restoring Rules

Marcello Balduccini[1] and Michael Gelfond[2]

[1] Kodak Research Laboratories
Eastman Kodak Company
Rochester, NY 14650-2102 USA
marcello.balduccini@gmail.com
[2] Computer Science Department
Texas Tech University
Lubbock, TX 79409 USA
michael.gelfond@ttu.edu

Abstract In this paper we continue the work on our extension of Answer Set Programming by non-Herbrand functions and add to the language support for arithmetic expressions and various inequality relations over non-Herbrand functions, as well as consistency-restoring rules from CR-Prolog. We demonstrate the use of this latest version of the language in the representation of important kinds of knowledge.

1 Introduction

In this paper we describe an extension of Answer Set Programming (ASP) [12,16,4] called ASP{f,cr}. This work continues our research on the introduction of non-Herbrand functions in ASP.

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, if equality is understood as identity, $f(x) = 2$ 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. [8,6]).

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 and O ranges over natural numbers.

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. Moreover, in ASP relational encodings of functions often pose performance issues. For example, the grounding of rule (2) grows with $O(|D|^2)$ where D is the range of variables O and O'. If $|D|$ is large, which is often the case especially with numerical variables, the size of the grounding can affect very negatively the overall solver performance.

Various extensions of ASP with non-Herbrand functions exist in the literature. In [7], 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, although the performance issues due to the size of the grounding remain. [14] proposes instead the use of second-order theories for the definition of the semantics of the language. Again, a transformation is (partially) 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. As with the previous approach, the performance issues are present. In [15,17] the semantics is based on the notion of reduct as in the original ASP semantics [12]. For the purpose of computing answer sets, a translation is defined, which maps programs of the language from [15,17] 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 [9] 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. All the approaches except for [7] support only total functions. While the approaches from [15,17,9] are computationally efficient, the approaches of [7,14], based on translations to ASP, are affected by performance issues due to the size of the grounding. Finally, in [1,2] we have proposed an extension of ASP with non-Herbrand functions, called ASP$\{\mathrm{f}\}$, that supports partial functions and is computationally more efficient than [7].

In the present paper, we extend the definition of ASP{f} from [1,2] further, by adding to it support for full-fledged arithmetic expressions and for consistency restoring rules from CR-Prolog [3]. We also contribute our perspective to the current debate on the usefulness of partial vs. total functions and on the role of non-Herbrand languages in general by demonstrating the use of our extended language for the representation of important types of knowledge and for the encoding of some classical scenarios, pointing out the differences with other approaches and with ASP encodings.

The rest of the paper is organized as follows. In the next section we extend ASP{f} with full-fledged arithmetic expressions. In the following section we introduce consistency-restoring. The resulting language is called ASP{f,cr}. Next, we address high-level issues of knowledge representation in ASP{f,cr} and we demonstrate the use of ASP{f,cr} for the formalization of some classical problems. Finally, we draw conclusions and discuss future work.

2 ASP{f} with Arithmetic Expressions

In this section we summarize the syntax and the semantics of ASP{f} [1,2], and extend the language with support for arithmetic expressions over non-Herbrand functional terms. For simplicity, in the rest of this paper we drop the attribute "non-Herbrand" and simply talk about functions and (functional) terms.

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. numerical constants 1 and 5) and have the standard interpretation.[3] A *simple 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 *arithmetic term* is either a simple term where f is a numerical function, or an expression constructed from such simple terms and numerical constants using arithmetic operations, such as $(f(c_1) + g(c_2))/2$ and $|f(c_1) - g(c_2)|$. Simple terms and arithmetic terms are called *terms*. 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 *term-atom*, or *t-atom*, is an expression of the form $f \ \mathtt{op} \ g$, where f and g are terms and $\mathtt{op} \in \{=, \neq, \leq, <, >, \geq\}$. A *seed t-atom* is a t-atom of the form $f = v$ such that f is a simple term and v is a constant. All other t-atoms are called *dependent*.

A *regular literal* is an atom a or its strong negation $\neg a$. A *literal* is either an atom a, its strong negation $\neg a$, or a t-atom. Any literal that is not a dependent t-atom is called *seed literal*.

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

$$h \leftarrow l_1, \ldots, l_m, \mathit{not}\ l_{m+1}, \ldots, \mathit{not}\ l_n \tag{3}$$

where h is a seed literal and l_i's are literals. Similarly to ASP, the informal reading of r is that a rational agent who believes $l_1, \ldots, l_m$ and has no reason to believe $l_{m+1}, \ldots, l_n$ must believe h.

[3] In the rest of the paper, whether an element of Σ is numerical will be clear from the context.

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 of the constraint must never be satisfied. A constraint is considered 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 *t-atom free* if no t-atoms occur in the rules of Π.

As in ASP, variables can be used 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 of $\mathcal{C}$[4] 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 $p(5) \leftarrow r(3), q(2)$. 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 $p(3 + 2)$ are non-ground.

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 [12] by simply extending entailment to t-atoms.

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 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, if the values of all the terms in t are defined; otherwise its value is undefined.[5] Finally, 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 literal l is *satisfied* by a consistent set S of seed literals under the following conditions: (1) if l is a seed literal, then l is satisfied by S iff $l \in S$; (2) if l is a dependent

[4] The replacement is with constants of suitable sort. We omit the details of this process to save space.

[5] This definition does not adequately capture the value of expressions such as $0 * f$ in the presence of undefined terms. We plan to address this and some related issues in a later paper.

t-atom of the form $f \circ\!\!p\ g$, then l is *satisfied* by S iff both $val_S(f)$ and $val_S(g)$ are defined, and they satisfy the equality or inequality relation $\circ\!\!p$ 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 t-atom $f \neq h$ 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 $e \in E$.

Next, we define the semantics of ASP{f}. 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 $r \in \Pi$.

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, $\{f = 2, p\}$. 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 $\{f = 3.\ \ f = 2 \leftarrow q.\ \ q.\}$ 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.

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 $S_3 = \{g = 3, f = 2, p, q\}$, 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 $\{r'_1 : p \leftarrow f = 2,\ r'_3 : g = 3,\ r'_4 : f = 2\}$

The semantics of arbitrary ASP{f} programs is given by the following definition:

Definition 3. *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.

3 ASP{f,cr}: Consistency-Restoring Rules in ASP{f}

In this section we extend ASP{f} by consistency-restoring rules from CR-Prolog [3]. We denote the extended language by ASP{f,cr}. As discussed in the literature on CR-Prolog, consistency-restoring rules are convenient for the formalization of various types of knowledge and of reasoning tasks. Later in this paper we show how consistency-restoring rules are useful for the formalization of knowledge about non-Herbrand functions as well.

A *consistency-restoring rule* (or *cr-rule*) is a statement of the form:

$$h \stackrel{+}{\leftarrow} l_1, \ldots, l_m, \mathit{not}\ l_{m+1}, \ldots, \mathit{not}\ l_n. \tag{4}$$

where h is a seed literal and l_i's are literals. The intuitive reading of the statement is that a reasoner who believes $\{l_1, \ldots, l_m\}$ and has no reason to believe $\{l_{m+1}, \ldots, l_n\}$, *may possibly* believe h. The implicit assumption is that this possibility is used as little as possible, only when the reasoner cannot otherwise form a non-contradictory set of beliefs.

By *ASP{f,cr} program* we mean a pair $\langle \Sigma, \Pi \rangle$, where Σ is a signature and Π is a set of rules and cr-rules over Σ.

Given an ASP{f,cr} program Π, we denote the set of its rules by Π^r and the set of its cr-rules by Π^{cr}. By $\alpha(r)$ we denote the rule obtained from cr-rule r by replacing symbol $\stackrel{+}{\leftarrow}$ with $\leftarrow$. Given a set of cr-rules R, $\alpha(R)$ denotes the set obtained by applying α to each cr-rule in R. The semantics of ASP{f,cr} programs is defined in two steps.

Definition 4 (Answer Sets of CR-Rule Free Programs). *The* answer sets *of a cr-rule free ASP{f,cr} program are the answer sets of the corresponding ASP{f} program.*

Definition 5. *Given an arbitrary ASP{f,cr} program Π, a subset R of Π^{cr} is an* abductive support *of Π if $\Pi^r \cup \alpha(R)$ is consistent and R is set-theoretically minimal among the sets satisfying this property.*

Definition 6 (Answer Sets of Arbitrary Programs). *For an arbitrary ASP{f,cr} program Π, a set of literals A is an* answer set *of Π if it is an answer set of the program $\Pi^r \cup \alpha(R)$ for some abductive support R of Π.*

Although out of the scope of the present paper, it is also possible to extend ASP{f,cr} to allow for the specification of CR-Prolog-style preferences over cr-rules.

4 Knowledge Representation with ASP{f,cr}

In this section we demonstrate the use of ASP{f,cr} for the formalization of certain types of knowledge. Whenever appropriate, we also compare with ASP and with other extensions of ASP by non-Herbrand functions.

Consider a scenario in which data from a town registry about births and deaths is used to determine who should receive a certain tax bill. The registry lists the year of birth and the year of death of a person. If a person is alive, no year of death is in the registry. The tax bill should only be sent to living people who are between 18 and 65 years old. To ensure robustness, we want to be able to deal with (infrequently) missing information. Hence, whenever there is uncertainty (represented by an atom $uncertain(p)$) about whether a person should receive the tax bill or not, a manual check will be performed. The first requirement can be encoded in ASP{f,cr} by the rule:

$$\begin{array}{l} bill(P) \leftarrow \\ \quad person(P), \\ \quad age(P) \geq 18, age(P) \leq 65, \\ \quad \textit{not } uncertain(P), \\ \quad \textit{not } \neg bill(P). \end{array}$$

Relation $person$ defines a list of people known to the system. To shorten the rules, from now on we will implicitly assume the occurrence of an atom $person(P)$ in every rule where P occurs. The condition *not* $\neg bill(P)$ allows one to specify exceptions in the usual way. For example, the tax might be waived for low-income people:

$$\neg bill(P) \leftarrow low\_income(P).$$

Similarly, condition *not* $uncertain(P)$ ensures that the bill is not sent if there is uncertainty about whether the person is subject to the tax.

Next, we define a person's current age based on their year of birth. Following intuition, we define age only for people who are alive.

$$has\_birth\_year(P) \leftarrow birth\_year(P) = X. \tag{5}$$
$$\neg has\_birth\_year(P) \leftarrow \textit{not } has\_birth\_year(P). \tag{6}$$

$$has\_death\_year(P) \leftarrow death\_year(P) = X. \tag{7}$$
$$\neg has\_death\_year(P) \leftarrow \textit{not } has\_death\_year(P). \tag{8}$$

$$\neg alive(P) \leftarrow has\_death\_year(P). \tag{9}$$
$$alive(P) \leftarrow has\_birth\_year(P), \neg has\_death\_year(P). \tag{10}$$

$$age(P) = X \leftarrow alive(P), X = current\_year - birth\_year(P). \tag{11}$$

Rules (5) and (7) determine when information about a person's birth and death year is available. Rules (6) and (8) formalize the closed world assumption (CWA) of relations has_birth_year and has_death_year. Although this encoding of CWA is common practice in ASP, it plays an important role in the distinction between languages with partial functions and languages with total functions, as we discuss later. Rule (9) states that it is possible to conclude that a person is dead if a year of death is found in the registry. Rule (10) states that a person is alive if the registry contains a year of birth and does not contain information about the person's death. Finally, rule (11) calculates a person's age. $current\_year$ is a function of arity 0 whose value corresponds to the current year.

The next set of rules deals with the possibility of information missing from the registry. One important case to consider is that in which information about a person's death is accidentally missing from the registry. In (10) a person is assumed to be alive unless evidence exists about the person's death. This modeling choice is justified because missing information is assumed to be infrequent. Exceptional conditions can be dealt with by requesting a manual check on whether the person should receive the tax bill. Rule (12) below states that one such case is when a person's age according to the registry is beyond that person's maximum life span.

$$uncertain(P) \leftarrow alive(P), age(P) \geq max\_span(P). \tag{12}$$
$$max\_span(P) = 92 \leftarrow \textit{not } max\_span(P) \neq 92. \tag{13}$$
$$max\_span(P) = 100 \leftarrow long\_lived\_family(P). \tag{14}$$

$$uncertain(P) \leftarrow \textit{not } alive(P), \textit{not } \neg alive(P). \tag{15}$$

Rule (13) provides a simple definition of a person's maximum life span, stating that, normally, a person's maximum life span is 92 years. Note that the rule is written in the form of a *default over non-Herbrand functions*. This makes it possible for example to predict a different life span depending on a person's medical or family history. Along

the lines of [5], the reading of (13) is "if P's maximum life span *may be* 92, then it is 92." Generally speaking, an expressions of the form *not* $f \neq g$ can be viewed to intuitively state "f *may* be (equal to) g". Rule (14) encodes one possible exception to the default, for people from families with a history of long life spans. Rule (15) covers instead the case in which the system couldn't determine if a person is dead or alive. The formalization of this type of test has already been covered in the literature and is shown here for completeness, and using a a slightly simplified encoding. A discussion on this topic and proposals for more sophisticated formalizations can be found in [10,11].

It is currently a source of debate [14,7] whether support for partial functions should be allowed in languages with non-Herbrand functions. Although of course from the point of view of computational complexity partial and total functions in this context are equivalent, we believe that the following elaboration of the tax-bill scenario appears to show that the availability of partial functions is indeed important.

First of all, notice that, in a language that only supported total functions, the scenario discussed so far would have to be formalized by introducing a special constant. This special constant is to be used when the birth or death years are unknown. For the sake of this discussion, let us denote the special constant by $<undef>$.

Notice that, to allow for the use of $<undef>$, a design requirement would have to be imposed on the town registry so that entries that do not have a value are set to $<undef>$. It is worth mentioning that one might be tempted to avoid the use of $<undef>$ and instead reason by cases, one for each possible value of the year of birth or death. This approach however does not appear to work well in this scenario. In fact, in this scenario it is important to know whether the year of death is present in the registry at all – see e.g. rules (5) and (7). When reasoning by cases, it is not possible to reason about this circumstance from within the program, unless rather sophisticated extensions of ASP such as [10,11] are used.

Going back to our scenario, suppose that we want to incorporate in our system information from a database of deadly car accidents. Suppose the database consists of statements of the form $died(p, y)$, where p is a person and y is the year in which the deadly accident occurred. If there are inconsistencies between the town registry and the accident database, we would like to give precedence to the former. This can be easily formalized in ASP{f,cr} with:

$$death\_year(P) = Y \leftarrow died(P, Y), \textit{not } death\_year(P) \neq Y. \tag{16}$$

Informally, the rule states that, if p is reported to have died in a car accident in year y, then that is assumed to be p's death year, unless the town registry contains information to the contrary.

It is important to notice that our ASP{f,cr} formalization makes it possible to incorporate the car accident database in a completely incremental fashion. No changes are needed to the rules we showed earlier. This is possible mainly because $death\_year$ is a partial function.

Let us see now how using a language with total functions would affect the incorporation of the car accident database. Let us consider a situation in which no death year is

specified for person p in the town registry, but an entry $died(p, 1998)$ exists in the car accident database. As discussed above, in a language that only supported total functions, $death\_year(p)$ would have to be set to $<undef>$ in the town registry. Hence, the body of rule (16) would never be satisfied.

To the best of our knowledge, working around this issue when using a language with total functions is non-trivial and the solutions are characterized by reduced elaboration tolerance. For example, one possible method consists in introducing a relation $determined\_death\_year(P)$ that encodes the *overall* belief of the system about a person's death year. By default, the value of $determined\_death\_year(P)$ is obtained from the town registry. When that value is undefined, a death year can be derived from the car accident database by means of a rule similar to (to avoid using a specific language from the literature, we write the rule in the syntax of ASP{f,cr}):

$$\begin{aligned} determined\_death\_year(P) = Y \leftarrow \\ \quad died(P, Y), \\ \quad \textit{not } determined\_death\_year(P) \neq Y. \end{aligned}$$

Furthermore, any rule that previously involved relation $death\_year$ would have to be modified to use the new relation. This process, although seemingly harmless on the surface, tends to be error-prone and the corresponding encoding is hardly elaboration tolerant. Every time information from another database had to be incorporated, more changes to the existing program are required – for example, the reader may want to consider would happen if one had to incorporate information about births from e.g. a health insurance database.

Up to this point we have discussed cases in which the use of total functions appears to cause some issues. One may be wondering whether it is possible to represent total functions in ASP{f,cr}, and if there are any drawbacks.

To address this topic, let us suppose that we would like to determine the number of dependents of a person. This information could be used for example in order to ensure that certain individuals are waived from paying the tax discussed earlier. For simplicity of presentation, however, we discuss the determination of the number of dependents independently of the code shown earlier.

Let us assume that the number of a person's dependents is found in their tax return, if one exists. If no tax return has been filed, we would like to consider separately each possible case, corresponding to a number of dependents ranging between 0 and max_d. The number of dependents can then be viewed as a total function.

In our formalization, an atom of the form $return\_dep(p, d)$ states that p has d dependents according to p's latest tax return (or, equivalently, a function could be used instead of a relation). We will represent the number of a person p's dependents by means of a function $dependents(p)$.

A straightforward formalization, Π_1^i, of such a total function is:

$$dependents(P) = D \leftarrow return\_dep(P, D). \tag{17}$$

$$dependents(P) = V \leftarrow \textit{not } dependents(P) \neq V. \tag{18}$$

Rule (17) states that the number of dependents can be obtained from the tax return, if available. Rule (18) states that a person can have any number of dependents, unless there is reason to believe otherwise. Here and below we assume that variable V ranges over the domain $[0, max_d]$ (this can be easily implemented by adding a condition $dom(V)$ and a suitable definition of relation dom).

Π_1^i formalizes the nature of total function $dependents$ for simple situations. However, suppose that one wanted to take into account the case in which p's tax return was audited and the number of p's dependents found to be different from what was stated in the tax return. Rule (17) does not properly deal with this case, because it prevents one from overriding a person's dependents based on information from the audit. So, a different formalization of total functions is needed to deal with more sophisticated examples.

One might then be tempted to rewrite (17) as a default and to add a suitable exception, obtaining Π_2^i:

$$dependents(P) = D \leftarrow return\_dep(P, D), \textit{not } dependents(P) \neq D. \quad (19)$$

$$dependents(P) = V \leftarrow \textit{not } dependents(P) \neq V. \quad (20)$$

$$dependents(P) = D \leftarrow assessed\_deps(P, D). \quad (21)$$

Unfortunately this formalization does not yield the intended answers because of the interaction between defaults (19) and (20): consider a person p_1 with 3 dependents according to their tax return, $I_1 = \{return\_deps(P, 3)\}$. One might expect $I_1 \cup \Pi_2^i$ to yield the conclusion $dependents(p) = 3$, but in fact the program has multiple answer sets, enumerating all the possible numbers of dependents between 0 and max_d. This is an instance of a phenomenon already studied in the literature (see e.g. [13]), which can be circumvented by properly prioritizing the defaults of Π_2^i. Doing so however tends to affect the elaboration tolerance of the encoding (e.g. in case further defaults must be added) and is somewhat cumbersome and error-prone.

A more robust and elaboration tolerant approach relies on the use of cr-rules. Intuitively, in this approach, a cr-rule determines when to trigger the default behavior of considering all the possible values of a total function. Consider the following program, Π_3^i:

$$dependents(P) = D \leftarrow return\_deps(P, D), \textit{not } dependents(P) \neq D. \quad (22)$$

$$dependents(P) = D \leftarrow assessed\_deps(P, D). \quad (23)$$

$$has\_dep\_info(P) \leftarrow dependents(P) = D. \quad (24)$$

$$\leftarrow \textit{not } has\_dep\_info(P). \quad (25)$$

$$dependents(P) = D \stackrel{+}{\leftarrow} . \quad (26)$$

Program Π_3^i is obtained from Π_2^i by replacing rule (20) by (24-26). Rules (24-25) intuitively state that the number of dependents must be known for every person. Cr-rule (26) intuitively states that it is possible to assume that a person has any number of dependents, but that this possibility should be used only if strictly necessary and in order to restore consistency.

It is not difficult to see that Π_3^i yields the expected conclusions. To this extent, it is important to notice that cr-rule (26) will only be used for people for whom no other dependent information is available. In fact, let I_3 be a set of facts providing partial information about the dependents of a group of people, and $U_3 = \{p_1, \ldots, p_u\}$ be the set of people from I_3 for whom no dependent information is available. According to Definition 5, any abductive support of $\Pi_3^i \cup I_3$ must contain, for every $p_i \in U_3$, a ground cr-rule $dependents(p_i) = d \stackrel{+}{\leftarrow}$ for some d. Let now R_3 be the set of all such cr-rules, and consider a person p' for whom dependent information is provided in I_3. The corresponding set $R_3' = R_3 \cup \{dependents(p') = d' \stackrel{+}{\leftarrow}\}$ is not an abductive support of $\Pi_3^i \cup I_3$ because it is not set-theoretically minimal. Hence, cr-rule (26) is only used for the people in U_3.

It is not difficult to see that this approach for the encoding of total functions in ASP{f,cr} is applicable in general, and that (24) can be rewritten as a general, domain-independent axiom (an example of a domain-independent axiom can be found in the next section).

5 Some Modeling and Solving Tasks in ASP{f,cr}

In this section we demonstrate the use of ASP{f,cr} for a sample of modeling and solving tasks from the literature. We also include a (partial) discussion of the features of our encodings in relation with other methods for representing non-Herbrand functions. We refer the reader to the description and original encodings from http://www.cs.uni-potsdam.de/∼torsten/kr12tutorial.

Water Buckets on a Scale (page 216). In this scenario, one bucket is placed on each arm of a two-armed scale. Each bucket initially contains an amount of water between 0 and 100. All amounts of water in this scenario are represented by integer values. At every time step, an agent must pour an amount k, $1 \leq k \leq max_w$ of water into one of the buckets.[6] The agent's goal is to balance the two buckets on the scale. The ASP{f,cr} encoding, Π^w, of this scenario is:

$bucket(a).\ bucket(b).$

$$1\{pour(B,T,K) : bucket(B) : K \geq 1 : K \leq max_w\}1 \leftarrow time(T), T < t. \tag{27}$$

$$poured(B,T) = K \leftarrow pour(B,T,K). \tag{28}$$

$$volume(B,T+1) = V \leftarrow V = volume(V,T) + poured(B,T). \tag{29}$$

$$volume(B,T+1) = V \leftarrow volume(B,T) = V, \mathit{not}\ volume(B,T+1) \neq volume(B,T). \tag{30}$$

$$heavier(B,T) \leftarrow bucket(B), bucket(C), time(T), volume(C,T) < volume(B,T). \tag{31}$$

$$\leftarrow bucket(B), heavier(B,t). \tag{32}$$

Rule (27) states that the agent can pour any allowed amount of water in any one bucket at every time step. For compactness, the rule uses the syntax of choice rules from

[6] We deviate slightly from the original scenario in that the agent is allowed to decide how much water is to be poured.

LPARSE and GRINGO. Extending the definition of ASP{f,cr} to support choice rules is trivial. Rule (28) states that the amount of water poured as a consequence of action $pour(b, t, k)$ is k. Rule (29) encodes a dynamic law; it states that when water is poured into a bucket, the volume of water in the bucket increases by the amount of water poured. We assume that a suitable domain has been specified for variable V. Rule (30) formalizes the inertia axiom. It states that the volume of water in a bucket stays the same unless it is forced to change. Rule (31) describes the conditions under which a buckets is heavier than the other. Finally, rule (32) states that it is impossible for a bucket to be heavier than the other at the end of the execution of the plan.

It is worth observing that, as prescribed by good knowledge representation principles, in Π^w the inertia axiom is written without references to the occurrence of any action. This allows for a rather elaboration tolerant encoding. In the original CLINGCON encoding, on the other hand, the inertia axiom mentions the occurrence of actions:

$$amount(B, T)\$ == 0 \leftarrow \mathit{not}\ pour(B, T), bucket(B), T < t. \tag{33}$$

$$volume(B, T + 1)\$ == volume(B, T) + amount(B, T) \leftarrow bucket(B), T < t. \tag{34}$$

This encoding is arguably less elaboration tolerant than Π^w: for example, the CLINGCON inertia axiom (33) must be modified whenever new actions are introduced in the representation, while the inertia axiom from Π_w does not have to be changed, and the whole program can be extended in a completely incremental fashion. In fact, ASP{f,cr} makes it possible to encode the inertia axiom in a form that is even more general than that of rule (30):[7]

$$num\_fluent(volume(B)) \leftarrow bucket(B). \tag{35}$$

$$val(N, T + 1) = val(N, T) \leftarrow num\_fluent(N), \mathit{not}\ val(N, T + 1) \neq val(N, T). \tag{36}$$

Rule (35) states that $volume(\cdot)$ is a "numerical fluent". Rule (36) states that the value of any numerical fluent remains the same over time unless it is forced to change. The advantage of this generalized form of the inertia axiom is that the corresponding rules apply without changes to any numerical fluent, so that now the addition of new numerical fluents to the encoding can be fully incremental as well.

From the point of view of the size of the grounding, the CLINGCON encoding is however superior to Π^w, because in Π^w rule (29) must be grounded for every possible value of variable V, while in the CLINGCON encoding the grounding is entirely independent of the volume of water in the buckets. On the other hand, the size of the grounding of Π^w is substantially better than the best ASP encodings that we are aware of. In the ASP encodings, in fact, the grounding of the inertia axiom grows proportionally to the *square* of the domain of variable V. A similar phenomenon can be observed in the encodings based on the languages of [7,14], since in those approaches computation of the answer sets is performed by translating the programs to ASP.

[7] To complete the encoding (29) and (31) have to be modified in a straightforward way to use $val(\cdot, \cdot)$ as well. From a technical perspective, in this encoding ground expressions $volume(a)$ and $volume(b)$ are viewed as constants. This is possible because no assumptions are made about the set of constants in our definition of the language. Alternatively, one could of course extend the language with Herbrand function symbols and of Herbrand terms, at the cost of a slightly more complex presentation.

N-Queens (page 176). In this scenario an agent must place n queens on an $n \times n$ chess board so that no queen can attack another. In this scenario the size of the grounding and the execution time tend to grow quickly with the increase of parameter n. In straightforward ASP encodings, the growth of the grounding is due to the tests ensuring that no queen can attack another. Π_1^q shows one possible ASP encoding:

$$\leftarrow queen(X_1, Y_1), queen(X_1, Y_2), Y_1 < Y_2.$$
$$\leftarrow queen(X_1, Y_1), queen(X_2, Y_1), X_1 < X_2.$$
$$\leftarrow queen(X_1, Y_1), queen(X_2, Y_2), X_1 < X_2, X_2 - X_1 = |Y_2 - Y_1|.$$

Conditions $Y_1 < Y_2$ and $X_1 < X_2$ are introduced in order to break symmetries. The last rule is the most problematic with respect to the size of the grounding, because its grounding grows roughly with $O(n^4)$. Several modifications of Π_1^q are known, which decrease the size of the grounding.[8] However, it is often argued that these modifications make the corresponding encodings either less declarative, or less elaboration tolerant. Certainly, most of the modifications achieve performance by a less straightforward encoding of the constraints of the problem.

It is then interesting to compare Π_1^q with a straightforward ASP{f,cr} encoding, Π_2^q:

$$\leftarrow Q_1 < Q_2, col(Q_1) = col(Q_2).$$
$$\leftarrow Q_1 < Q_2, row(Q_1) = row(Q_2).$$
$$\leftarrow Q_1 < Q_2, col(Q_2) - col(Q_1) = |row(Q_2) - row(Q_1)|.$$

Condition $Q_1 < Q_2$ performs basic symmetry breaking. Π_2^q uses two functions to encode the positions of the queens. *What is remarkable about Π_2^q is that the grounding of the last rule grows roughly with $O(n^2)$, although we argue that it is as straightforward an encoding of the requirement as the corresponding rule from Π_1^q.* As in the previous scenario, we expect a similar growth for comparable CLINGCON encodings, and a growth of $O(n^4)$ for the grounding of the encodings written in the languages of [7,14].

6 Conclusions and Future Work

In this paper we have defined the syntax and semantics of an extension of ASP by non-Herbrand functions with full-fledged arithmetic expressions and consistency-restoring rules. The resulting language ASP{f,cr} supports partial functions and we hope we have demonstrated that it allows for the encoding of rather sophisticated kinds of knowledge, including knowledge about total functions. Compared to similar languages, ASP{f,cr} strikes a remarkable balance between expressive power and efficiency of computation. In the previous section, the discussion on the efficiency of computation was based only on the size of the grounding of the corresponding encodings, but in [1] experimental evidence on solver performance was obtained using a prototype of an ASP{f} solver (available at http://marcy.cjb.net/clingof). We expect that a version of the solver including support for the extended language defined in this paper will be available soon. Once that becomes available, we plan to substantiate the discussion from the previous section with experimental results.

[8] See especially http://www.cs.uni-potsdam.de/∼torsten/kr12tutorial.

References

1. Balduccini, M.: Answer Set Solving and Non-Herbrand Functions. In: Rosati, R., Woltran, S. (eds.) Proceedings of the 14th International Workshop on Non-Monotonic Reasoning (NMR'2012) (Jun 2012)
2. Balduccini, M.: Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz, chap. 3. A "Conservative" Approach to Extending Answer Set Programming with Non-Herbrand Functions, pp. 23–39. Lecture Notes in Artificial Intelligence (LNCS), Springer Verlag, Berlin (Jun 2012)
3. Balduccini, M., Gelfond, M.: Logic Programs with Consistency-Restoring Rules. In: Doherty, P., McCarthy, J., Williams, M.A. (eds.) International Symposium on Logical Formalization of Commonsense Reasoning. pp. 9–18. AAAI 2003 Spring Symposium Series (Mar 2003)
4. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press (Jan 2003)
5. Baral, C., Gelfond, M.: Logic Programming and Knowledge Representation. Journal of Logic Programming 19(20), 73–148 (1994)
6. Baselice, S., Bonatti, P.A.: A Decidable Subclass of Finitary Programs. Journal of Theory and Practice of Logic Programming (TPLP) 10(4–6), 481–496 (2010)
7. Cabalar, P.: Functional Answer Set Programming. Journal of Theory and Practice of Logic Programming (TPLP) 11, 203–234 (2011)
8. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Enhancing ASP by Functions: Decidable Classes and Implementation Techniques. In: Proceedings of the Twenty-Fourth Conference on Artificial Intelligence. pp. 1666–1670 (2010)
9. Gebser, M., Ostrowski, M., Schaub, T.: Constraint Answer Set Solving. In: 25th International Conference on Logic Programming (ICLP09). vol. 5649 (2009)
10. Gelfond, M.: Strong Introspection. In: Dean, T., McKeown, K. (eds.) Proceedings of the 9th National Conference on Artifical Intelligence. pp. 386–391. AAAI Press/The MIT Press, Menlo Park, CA (Jul 1991)
11. Gelfond, M.: New Semantics for Epistemic Specifications. In: Delgrande, J.P., Faber, W. (eds.) 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR11). Lecture Notes in Artificial Intelligence (LNCS), vol. 6645, pp. 260–265. Springer Verlag, Berlin (2011)
12. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing 9, 365–385 (1991)
13. Gelfond, M., Son, T.C.: Reasoning with Prioritized Defaults. In: Third International Workshop, LPKR'97. Lecture Notes in Artificial Intelligence (LNCS), vol. 1471, pp. 164–224 (Oct 1997)
14. Lifschitz, V.: Logic Programs with Intensional Functions (Preliminary Report). In: ICLP11 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP11) (Jul 2011)
15. Lin, F., Wang, Y.: Answer Set Programming with Functions. In: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR2008). pp. 454–465 (2008)
16. Marek, V.W., Truszczynski, M.: The Logic Programming Paradigm: a 25-Year Perspective, chap. Stable Models and an Alternative Logic Programming Paradigm, pp. 375–398. Springer Verlag, Berlin (1999)
17. Wang, Y., You, J.H., Yuan, L.Y., Zhang, M.: Weight Constraint Programs with Functions. In: Erdem, E., Lin, F., Schaub, T. (eds.) 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR09). Lecture Notes in Artificial Intelligence (LNCS), vol. 5753, pp. 329–341. Springer Verlag, Berlin (Sep 2009)