Towards Answer Set Programming with Sorts

Evgenii Balai, Michael Gelfond, and Yuanlin Zhang

Texas Tech University, USA
{evgenii.balai, michael.gelfond, y.zhang}@ttu.edu

Abstract. Existing ASP languages lack support for conveniently specifying objects, their sorts and the sorts of the parameters of relations in an application domain. However, such support may allow a programmer to better structure the program, to automatically determine some syntax and semantic errors and to avoid thinking about safety of ASP rules — non-declarative conditions on rules required by existing ASP systems. In this paper, we define the syntax and semantics of a knowledge representation language $\mathcal{SPARC}$ which offers explicit constructs to specify objects, relations, and their sorts. The language expands CR-Prolog — an extension of ASP by consistency restoring rules. We introduce an implementation of $\mathcal{SPARC}$ based on its translation to DLV with weak constraints. A syntax checking algorithm helps to avoid errors related to misspellings as well as simple type errors. Another type checking algorithm flags program rules which, due to type conflicts, have no ground instantiations.

1 Introduction

A good knowledge representation methodology should allow one to:

- Identify and describe *sorts* (types, kinds, categories) of objects populating a given domain.
- Identify and classify these objects.
- Identify and precisely define objects *properties* and *relationships* between them.

ASP[1] based knowledge representation languages have powerful means for describing these properties and relationships but lack the means for conveniently specifying objects and their sorts as well as sorts of parameters of the domain relations. There were some attempts to remedy the problem. The #domain statements of lparse [2] — a popular grounder used for a number of ASP systems — define sorts for variables. Even though this device is convenient for simple programs it causes substantial difficulties for medium and large programs. It is especially difficult to put together pieces of programs written at different time and/or by different people. The same variable may be declared as ranging over different sorts by different #domain statements used in different parts of a program. So the process of merging these parts requires renaming of variables. This concern was addressed by Balduccini whose system, RSig[3] , provided an ASP programmer with means for specifying sorts of parameters of the language predicates. RSig is a simple (but very useful) extension of ASP which does not require any shift in perspective and involves only minor changes in existing programs. In this work we further develop the idea of RSIG by introducing a knowledge representation language $\mathcal{SPARC}$. In

addition to allowing the specification of program relations and their parameters $\mathcal{SPARC}$ provides a programmer with means for defining objects of the program domain and their sorts. This allows *better separation of concerns*. A programmer is encouraged to write rules which express general properties of the domain and do not necessarily refer to particular domain objects. Such rules can be used in conjunction with different collections of objects and/or different placement of objects into sorts. A simple syntax checking algorithm helps a programmer to *avoid errors* related to misspelling as well as simple type errors. (Despite their simplicity such errors are sometimes not easy to identify.) Explicit declaration of sorts allows a programmer to *avoid thinking about safety conditions* in program rules — a feature especially important when $\mathcal{SPARC}$ is used to *teach* declarative programming. Finally a *type checking algorithm* locating rules of the program which, because of the type restrictions on variables, do not have any ground instantiations is useful for determining more subtle potential problems. The paper defines the syntax and semantics of a version of $\mathcal{SPARC}$ defined on top of CR-Prolog — an ASP based language with consistency-restoring rules [4]. It also describes the corresponding syntax and type checking algorithms, and an algorithm for computing answer sets of a $\mathcal{SPARC}$ program based on reduction of such a program to DLV [5] — a language of disjunctive logic programs with weak constraints [6]. The preliminary description of the language and the latter algorithm has been presented in [7] in 2012. The new version of the language however is quite different from that presented in this workshop. The most important improvement is the completely new definition of sorts and domain objects of a program. An implementation of the SPARC system can be found at [8]. The paper is organized as follows. In the next section we define syntax and semantics of $\mathcal{SPARC}$. We then present syntax and type checking algorithms in Sections 3 and 4, and an algorithm for computing answer sets of a $\mathcal{SPARC}$ program in Section 5. Most of the paper can be understood by anyone familiar with logic programming under the answer set semantics. However, full understanding of Section 5 requires knowledge of CR-Prolog.

2 Syntax and Semantics of $\mathcal{SPARC}$

$\mathcal{SPARC}$ vocabulary consists of *variables*, *sort names*, *symbolic names*, *natural numbers*, equality ($=$) and inequality (!=) defined on arbitrary terms, *order relations* ($<, \leq$) on numbers and on symbolic names (ordering of symbolic names is lexicographic), and standard *arithmetic functions*. Variables and symbolic names are identifiers which start with capital and lower-case letters respectively; sort name is a symbolic name preceded by $\#$. The vocabulary is used to define $\mathcal{SPARC}$ *terms* which are divided into arithmetic and symbolic. An *arithmetic term* is defined as usual. A *symbolic term* is a symbolic name, or a variable, or a string of the form $f(t_1, \ldots, t_n)$ where f is a symbolic name and $t_1, \ldots, t_n$ are arithmetic or symbolic terms. A term $f(t_1, \ldots, t_n)$ is referred to as a *record* with the *record name* f (of arity n). A term is called *ground* if it contains no variables and no arithmetic operations. A *set expression* of $\mathcal{SPARC}$ is either a sort name,a collection of ground terms $\{t_1, \ldots, t_n\}$, or has the form $(A \odot B)$ where A and B are set expressions and $\odot$ is a set theoretic operation $+$, $-$ or $\times$. Parentheses can be omitted and standard preference is used to determine the order of operations. We

also need two special sorts *dom* and *nat* which belong to every program of $\mathcal{SPARC}$: the former consists of all ground terms from the signature of the program, and the sort *nat* of natural numbers between 0 and $maxint$.

Now we are ready to define the syntax of $\mathcal{SPARC}$. A $\mathcal{SPARC}$ program is constructed from four consecutive parts:
The **first part**, called *directives* consists of a (possibly empty) collection of statements of the form
`#const <identifier> = <natural_number>.`
`#maxint = <natural_number>.`
The **second part** of the program consists of the keyword **sorts** followed by a list of *sort definitions* — statement of the form (1) – (5) below. It is used to define

- objects of the program's domain (often referred to as *domain elements*) and
- sort names and their assignments to non-empty sets of domain elements.

The list consists of statements of the form
`sort_name = sort_expression`
where sort expressions are expressions appearing on the right-hand side of statements (1) – (5) below. Each such expression, E, defines a collection $\mathcal{D}(E)$ of ground terms which is assigned to the sort `sort_name`. In addition, if $\{t_1, \ldots, t_n\}$ occurs in the sort expression on the right then *every* t_i *together with its subterms* is added to the set, *dom*, of domain elements of the program.
Statement

$$sort\_name = set\_expr \tag{1}$$

defines a sort, $sort\_name$ using the set expression on the right. For example the sort definition consisting of statements

```
#blocks = {b1,b2}
#locations = #blocks + {table}
```

defines the program domain consisting of elements $\{b1, b2, table\}$; sort `#blocks` is mapped into $\{b1, b2\}$; and sort `#locations` mapped into $\{b1, b2, table\}$. The sort definition

```
#names = {name(bob,smith), name{mary,smith}}
```

defines the set `names` consisting of the two records on the right and expands the set of domain elements by these records and their subterms bob, $mary$, and $smith$.

Statement of the form

$$sort\_name = [n_1..n_2] \tag{2}$$

where n_1 and n_2 are natural numbers and $n_1 \leq n_2$ defines the sort $\{n : n_1 \leq n \leq n_2\}$.

Similarly if id_1 and id_2 are identifiers then the statement

$$sort\_name = [id_1..id_2] \tag{3}$$

defines the sort $\{id : |id_1| \leq |id| \leq |id_2|$ and $id_1 \leq id \leq id_2\}$ where $\leq_l$ is the lexicographic ordering on identifiers.

The next statement has the form

$$sort\_name = f(s_1(var_1), ..., s_n(var_n)) : cond \tag{4}$$

where f is a new symbolic name, $s_1, \ldots, s_n$ are previously defined sorts and $cond$ has the form $X \diamond Y$, where $X, Y \in \{var_1, \ldots var_n\}$ and $\diamond \in \{<, \leq, =, \neq\}$, or $C_1 \bullet C_2$, where C_1 and C_2 are conditions and $\bullet \in \{\vee, \wedge\}$. Both, the variables and the condition, can be omitted. The new sort is assigned a collection of records of the form $f(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are elements of sorts $s_1, \ldots, s_n$ satisfying condition $cond$. For instance, a statement

```
#actions = put(#blocks,#locations).
```

defines a new sort, $actions$, consisting of records of the form $put(b, l)$ where b is a block and l is a location. Note that, according to this definition, a record $put(b1, b1)$ is an action. Sometimes it is convenient to exclude this possibility. This can be achieved by the following alternative definition of $actions$:

```
#actions = put(#blocks(X),#locations(Y)) : X != Y.
```

Now a record $put(b, l)$ belongs to the sort $actions$ if b is a block, l is a location, and b and l are different. The statement

$$sort\_name = [b\_expr][b\_expr] \ldots [b\_expr] \tag{5}$$

defines concatenation of *basic* sorts, i.e., sorts consisting of identifiers and natural numbers; b_expr is the name of a basic sort or a list $t_1, \ldots, t_n$ of natural numbers and symbolic names or expressions of the form $n_1..n_2$ and $id_1..id_2$ where n_1, n_2 are natural numbers and id_1, id_2 are symbolic names. These sort definitions are useful when we want to define large basic sorts, e.g. a sort of blocks $b_1, \ldots, b_{100}$ can be defined as:

```
#blocks = [b][1..100]
```

Definition 1 (Sorts Definitions).
The *list of sort definitions* of a program is a sequence of statements of the form (1)–(5) such that no sort name occurs on the left-hand side of a statement more than once and no sort name occurs on the right-hand side of a statement if it was not previously defined.

The collection of sorts of a program consists of *sorts defined by sort definitions of the program* and sorts *dom* and *nat*.

Definition 2 (Domain Elements).
A ground term t of $\mathcal{SPARC}$ is an *element of the program's domain* if

1. t is a natural number belonging to sort nat or
2. t is defined by a sort definition of the form (2)-(5) or
3. there is a sort definition containing an occurrence of $\{.., t, ..\}$ or
4. t is a subterm of a term satisfying one of the above properties.

In the first two cases t belongs to at least one sort defined by the corresponding sort definition. The domain element defined by one of the last two clauses of the definition may or may not have such a sort. In this case it belongs to sort *dom* of the program.

We say that a *record name is defined by a program* Π if it occurs in one of the elements of Π's domain.

The **third part** of the program defines predicate symbols and sorts of their parameters. It starts with a keyword **predicates** and is followed by statements of the form

$$pred(sort\_name, \ldots, sort\_name)$$

where $pred$ is a new identifier and $sort_name$s are sort names defined by the sort definitions. The statement defines predicate symbol $pred$ and specifies its arity and the sorts of its parameters.

The first three sections of a $\mathcal{SPARC}$ program Π uniquely define the program's signature. To define rules of Π we need the following definitions:

Definition 3 (Program Term).
A $\mathcal{SPARC}$ term t is called a *term of* $\mathcal{SPARC}$ *program* Π if every ground subterm of t is an element of the program's domain and every record name occurring in t is defined by Π.

Let $p(s_1, \ldots, s_n)$ be a predicate declaration of Π. By $\Sigma(p)$ we denote the sequence $(s_1, \ldots, s_n)$. If p is a sort name, $\Sigma(p)$ is p.

Definition 4 (Program Atom).
A string $p(t_1, \ldots, t_n)$, where p is a predicate symbol or sort of Π and $t_1, \ldots, t_n$ are Π's terms, is an atom of Π if:

- Let $\Sigma(p)$ be $(s_1, s_2, \ldots, s_n)$
- for each $i \in \{1..n\}$:
  - if t_i is a ground symbolic term then t_i belongs to s_i,
  - if t_i is an arithmetic term without variables, s_i must contain the value of t_i (denoted by $val(t_i)$),
  - if t_i is an arithmetic term with variables and at least one arithmetic operation, s_i must contain at least one number.

An atom A of Π or its negation $\neg A$ are called literals of Π.

Example 1 (Program Π_0*).*
To see some examples consider a program Π_0 containing the following:

```
#const n = 1.
sorts
#s1 = {f(b)}.
#s2 = [0..n].
predicates
p(#s1,#s2).
```

It is easy to see that $\{b, f(b)\}$ where $b \in dom$ and $f(b) \in \#s1$ are non-numerical ground terms of Π_0; $p(X, X)$ is an atom of Π_0, while $p(X, f(b))$,$p(X, a)$ and $p(0, X)$ are not.

Definition 5 (Program Rules).
A *rule* of a $\mathcal{SPARC}$ program Π is a regular ASP rule

$$l_0 \vee \ldots \vee l_m \leftarrow l_{m+1}, \ldots, l_k, not\ l_{k+1} \ldots not\ l_n \tag{6}$$

or a CR-Prolog rule

$$l_0 \stackrel{+}{\leftarrow} l_1, \ldots, l_k, not\ l_{k+1} \ldots not\ l_n \tag{7}$$

where l's are literals of Π and l_0 is not formed by a sort name. We say that a rule is *ground* if it is constructed from ground literals.

The **fourth part** of a $\mathcal{SPARC}$ program starts with the keyword **rules** and is followed by a finite collection of rules of Π. *This completes our definition of syntax of $\mathcal{SPARC}$ programs*. In what follows sort definitions, predicate declarations and program rules of Π will be denoted by $\mathcal{S}(\Pi)$, $\mathcal{P}(\Pi)$, and $\mathcal{R}(\Pi)$ respectively.

To define the semantics of $\mathcal{SPARC}$ program Π we will define its *answer sets*. If the rules of Π are ground then answer sets of Π are answer sets of the collection of its ground rules. To define answer sets of a program Π with variables we need some terminology. A *ground instance* of a rule r of Π is a ground rule of Π which is the result of replacing variables of r by properly-sorted elements of the Π's domain; $ground(r)$ is the collection of all such instantiations; $ground(\Pi)$ is the union of $ground(r)$ for all rules of Π.

Definition 6 (Answer Sets).
Answer sets of a $\mathcal{SPARC}$ program Π are answer sets of an unsorted logic program $ground(\Pi)$.

Example 2 (Program Π_0 (continued)).
Let us now complete our program Π_0 by adding to it the rules:

```
p(f(b),0).
p(X,X).
p(f(b),Y+1) :- p(f(b),Y).
```

$ground(\Pi_0)$ consists of ground rules

```
p(f(b),0).
p(f(b),1) :- p(f(b),0).
```

Note that there is no subsitution of X in $p(X, X)$ which respects the sorts of p. Hence, the rule $p(X, X)$ has no ground instantiations; $\{p(f(b), 0), p(f(b), 1)\}$ is the only answer set of $ground(\Pi_0)$ and hence of Π_0. Notice that according to this definition we cannot expand Π_0 by the statement

```
p(X,f(b)).
```

since, according to our sort and predicate declarations, it would not be a rule of the resulting program.

2.1 Discussion

Notice that the above definition of $ground(\Pi)$ involves a non-obvious choice. We do not require the set $ground(r)$ to be non-empty. The alternative would be to prohibit such rules. Under this alternative definition Π_0 would not be a program of $\mathcal{SPARC}$. (Note that $\Pi_0 \cup \{p(X, a)\}$ is not a program under any of these definitions). Our choice is based on the methodology for writing $\mathcal{SPARC}$ programs which attempts to make them elaboration tolerant. We assume that normally programmers will be fully aware of sort, function, and predicate symbols of the program's signature but not necessarily about actual content of the sorts. As an example one may think about a programmer representing "blocks world" domain. He may structure the world in terms of sorts $steps$, $blocks$, $locations$, $actions$ and $fluents$, and predicate symbols $holds(fluents, steps)$ and $occurs(actions, steps)$, and write causal laws representing the domain, e.g. $holds(on(B, L), I + 1) \leftarrow occurs(put(B, L), I)$. Later he may define the sorts of the program including that of $actions$. Suppose this is done using the first definition of $actions$ from page 4. If the programmer later wants to use this knowledge for planning he may decide to exclude generating an action $put(B, B)$ by a constraint $\leftarrow occurs(put(B, B), I)$.
After further consideration the definition of $actions$ can be changed to the second version, which would leave our constraint without ground instantiations. Should this result in error? Our answer is "no". The rule will simply automatically disappear during the grounding process. We will however have an option of warning the programmer about such an event (see section 4).

3 Checking the Program Syntax

In this section we define a syntax checking algorithm for $\mathcal{SPARC}$ programs. Given program Π, the syntax check of directives and predicate declarations of Π is straightforward. Checking correctness of sort expressions involves checking their syntax, including non-emptiness of the sorts which can be done by a simple recursive algorithm. In the process we also mark all sorts which contain at least one number and create the list of names of all the program records. The rule part of the program is syntactically correct iff each of its rules is correct, i.e. if each rule is properly constructed from program atoms. The main work is performed by functions *IsAtom(A, Π)* and *IsTerm(T, Π)* which return *true* iff A and T are atom and term of Π respectively. Another important function, *ReduceTerm* uses sort definitions of Π, a ground term t and a sort s to construct a formula which evaluates to true iff $t \in s$. To be more precise we need the following definitions:

Definition 7 (Formula).

- $T \in D$, where T is a variable, a ground term or an arithmetic term, and D is a set of ground terms, is a *formula*,
- $t_1 \diamond t_2$, where t_1 and t_2 are terms and $\diamond \in \{=, \neq, \prec, \preceq\}$, is a *formula*, and
- if A and B are formulas then $(A \wedge B)$, $(A \vee B)$, and $\neg A$ are *formulas*.

Formula F is called *ground* if it does not contain variables.

Relation $\prec$ is defined on arbitrary terms; $X \prec Y$ iff X and Y are both symbolic names or both integers and $X < Y$. Otherwise $X \prec Y$ is false. Similarly for $\preceq$.

Definition 8 (Satisfiability). A formula $\mathcal{F}$ is *satisfied* by a substitution θ of variables of $\mathcal{F}$ by ground $\mathcal{SPARC}$ terms if the result, $\mathcal{F}(\theta)$, of this substitution is true.

Now we are ready to describe $IsAtom$ and $IsTerm$:

Algorithm 1: IsAtom

Input: a string of the form $p(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are $\mathcal{SPARC}$ terms, and a $\mathcal{SPARC}$ program Π.
Output: *true* if $p(t_1, \ldots, t_n)$ is an atom of Π and *false* otherwise.

if *p is not a sort or a predicate name of Π* **then**
    return false
Let $\Sigma(p)$ be $(s_1, s_2, \ldots, s_n)$
for each *t_i of $p(t_1, \ldots, t_n)$* **do**
    if *t_i is a ground term and $ReduceTerm(t_i, s_i, \Pi)$ is* false **then**
        return false
    if *t_i is an arithmetic term without variables and $ReduceTerm(val(t_i), s_i, \Pi)$ is* false **then**
        return false
    if *t_i is an arithmetic term with variables and at least one arithmetic operation and s_i does not contain a number* **then**
        return false
    if *t_i is not a ground term and $IsTerm(t_i, \Pi)$ is* false **then**
        return false
return true

Algorithm 2: IsTerm

Input: a $\mathcal{SPARC}$ term t and a program Π.
Output: *true* if t is a term of Π and *false* otherwise.

if *there exists a record name in t that is not defined by Π* **then**
    return false
for each *maximum ground subterm u of t* **do**
    if *u is a natural number such that $u > \#maxint$* **then**
        return false
    if *u is a symbolic term not occurring in the sort definitions of Π* **then**
        if *there is no sort s such that $ReduceTerm(u, s, \Pi)$ is* true **then**
            return false
return true

The only thing left is to define function $ReduceTerm(t, s, \Pi)$ mentioned above. Note that for our purpose it is sufficient to define it for a ground term t only. But we

introduce a more general algorithm which allows t to be non-ground. This will be useful in the next section.

Algorithm 3: ReduceTerm

Input: a term t and a sort expression E of a $\mathcal{SPARC}$ program Π.
Output: a formula $\mathcal{C}$ which is satisfiable if and only if there exists a substitution θ, such that $t\theta \in \mathcal{D}(E)$.

1 **if** E *is a sort name defined by a statement* $E = E_1$ **then**
2 $\quad \mathcal{C} := ReduceTerm(t, E_1, \Pi)$
3 **else if** E *is of the form* $E_1 \odot E_2$*, where* $\odot \in \{+, -, *\}$ **then**
4 $\quad \mathcal{C} := (ReduceTerm(t, E_1, \Pi)) \bigtriangledown (ReduceTerm(t, E_2, \Pi))$, where $A \bigtriangledown B$ is $A \vee B$,$A \wedge \neg B$, or $A \wedge B$ when $\odot$ is $+$, $-$, or $*$ respectively
5 **else if** E *is of the form* $f(s_1[X_1], \ldots, s_n[X_n]) : cond(X_1, \ldots, X_n)$
6 $\quad$ *where the condition* $cond(X_1, \ldots, X_n)$ *is optional* **then**
7 $\quad$ **if** t *is not a variable and is not formed by a record name* f **then**
8 $\quad\quad$ **return** false
9 $\quad$ Let $X'_1, \ldots X'_n$ be new variables
10 $\quad$ **if** t *is of the form* $f(t_1, \ldots t_n)$ **then**
11 $\quad\quad \mathcal{C} := (X'_1 = t_1) \wedge \cdots \wedge (X'_n = t_n))$
12 $\quad$ **else if** t *is a variable* **then**
13 $\quad\quad \mathcal{C} := (t = f(X'_1, \ldots X'_n))$
14 $\quad$ **if** *condition* $cond(X_1, \ldots, X_n)$ *is present in* E **then**
15 $\quad\quad \mathcal{C} := \mathcal{C} \wedge cond'(X'_1, \ldots, X'_n)$ where $cond'(X'_1, \ldots, X'_n)$ is obtained from $cond(X_1, \ldots, X_n)$ by replacing X_i with X'_i and $<, \leq$ with $\prec, \preceq$ respectively.
16 $\quad \mathcal{C} := \mathcal{C} \wedge (ReduceTerm(X'_1, s_1, \Pi)) \wedge \ldots \wedge (ReduceTerm(X'_n, s_n, \Pi))$
17 **else**
18 $\quad$ **if** t *is not ground term of the form* $f(t_1, \ldots t_n)$ **then**
19 $\quad\quad \mathcal{C} = \vee\{(t_1 = t'_1) \wedge \ldots \wedge (t_n = t'_n) | f(t'_1, \ldots, t'_n) \in \mathcal{D}(E)\}$[a]
20 $\quad$ **else**
21 $\quad\quad \mathcal{C} = t \in \mathcal{D}(E)$
22 **return** $\mathcal{C}$

[a] empty disjunction is interpreted as `false`

Note that the algorithm $ReduceTerm$ comes to line 17 when expression E is of the form $\{t_1, \ldots, t_n\}$ or is defined by statements of the form 2,3 or 5. In this case the corresponding $\mathcal{D}(E)$ is computed explicitly. The correctness of $ReduceTerm$ algorithm is guaranteed by the following claim:

Claim. Given a $\mathcal{SPARC}$ term t, a sort expression E of a program Π and a substitution θ, θ is a solution of the formula $ReduceTerm(t, E, \Pi)$ if and only if $t\theta \in \mathcal{D}(E)$.

Example 3 (Tracing the Algorithm).
Now let us trace our syntax checker on a rule `p(f(b),Y+1) :- p(f(b),Y)` of program Π_0 from Examples 1 and 2. To check the rule's syntax we use $IsAtom$ to establish that $p(f(b), Y+1)$ and $p(f(b), Y)$ are atoms of Π_0. $IsAtom(p(f(b), Y+1), \Pi_0)$ calls $ReduceTerm(f(b), s_1, \Pi_0)$ which returns true (see line 21). After that we have the following two calls: $IsTerm(Y+1, \Pi_0)$ and $ReduceTerm(1, s_2, \Pi_0)$. The latter, and hence the former, return true. Hence, the head of our rule is an atom of Π_0. Similarly for the body. Therefore, the rule `p(f(b),Y+1) :- p(f(b),Y)` is indeed a rule of Π_0.

Now let $\Pi_1 = \Pi_0 \cup \{p(X, f(b)).\}$. This time $IsAtom(p(X, f(b)), \Pi_1)$ will return false, because $f(b)$ is a ground term which is not an element of corresponding sort $s2$ (therefore, $ReduceTerm(f(b), s2, \Pi_1)$ returns false).

4 Empty Rule Checking

In this section we introduce an algorithm, *IsEmptyRule*, which checks if a rule r of Π is *empty*, i.e. has no ground instantiations. This is done by applying a standard constraint satisfaction algorithm to a constraint formula over finite domains[9] produced by function *ReduceRule*.

Algorithm 4: IsEmptyRule

Input: rule r of a program Π.
Output: *true* if r is a non-empty rule of Π and *false* otherwise.

1 $C = ReduceRule(r, \Pi)$
2 **return** $satisfiable(C)$

Algorithm 5: ReduceRule

Input: a rule r and a $\mathcal{SPARC}$ program Π.
Output: a formula $\mathcal{C}$, which is satisfiable if and only if r is not empty rule of Π.

1 $\mathcal{C} :=$ `true`
2 **for each** t_i *of each atom* $p(t_1, \ldots, t_n)$ *occurring in* r **do**
3 Let $(s_1, \ldots, s_n)$ be $\Sigma(p)$
4 $\mathcal{C} := \mathcal{C} \wedge ReduceTerm(t_i, s_i, \Pi)$
5 **return** $\mathcal{C}$

In $ReduceRule$, we extract constraints, using $ReduceTerm$, for every term of every atom of r and connect them by conjunctions. The function $ReduceTerm$ takes a term t and a sort expression E of a program Π and returns a formula which is satisfiable if and only if there is an instance of t which belongs to $\mathcal{D}(E)$.

Claim. Given a $\mathcal{SPARC}$ program Π and a program rule r of Π, $IsEmptyRule(\Pi, r)$ returns true if and only if r is not empty.

Example 4 (Empty rule).
Consider the rule `p(X,X)` of program Π_0. $ReduceRule(r, \Pi_0)$ returns formula $X \in \{f(b)\} \wedge X \in \{1, 2\}$ which is clearly unsatisfiable. Therefore, the rule is an empty rule.

5 Computing Answer Sets of a $\mathcal{SPARC}$ Program

Answer sets of a $\mathcal{SPARC}$ program Π_{sparc} will be computed by translating the program into a program in the language of DLV with weak constraints. First we need some notation: every cr-rule r of Π_{sparc} will be assigned a unique number i. An expression $rn(i, X_1, ..., X_n)$ where $X_1, ..., X_n$ is the list of distinct variables occurring in r will be referred to as the *name of* r. For instance, if rule $p(X, Y) \leftarrow q(Z, X, Y)$ is assigned number 1 then its name is $rn(1, X, Y, Z)$. We also need the following definition:

Definition 9 (DLV counterparts of $\mathcal{SPARC}$ programs). A DLV program Π_{dlv} is *a counterpart* of a $\mathcal{SPARC}$ program Π_{sparc} if

- the signature of Π_{dlv} is an extension of the signature of Π_{sparc}, and
- the answer sets of Π_{sparc} and Π_{dlv} coincide on literals from the language of Π_{sparc}.

The translation is performed by Algorithm 6. The basic idea is to explicitly add the necessary sorts in the bodies of the DLV rules (which will eliminate possible problems with the safety of variables) and to replace cr-rules by a collection of weak constraints. The latter requires introduction of some new predicate symbols which explains the first requirement in definition 9.

Algorithm 6: Translate

Input: a $\mathcal{SPARC}$ program Π_{sparc}
Output: a DLV counterpart Π_{dlv} of Π_{sparc}.

1 Set variable Π_{dlv} to directives of Π_{sparc}
2 Let `appl`/1 be a new predicate not occurring in Π_{sparc}
3 **for each** *rule* r *in* Π_{sparc} **do**
4 $S := \{s(t) \mid$ there exists $p(t_1, ..., t_n),$ occurring in $r,$ such that
5 $p(s_1, \ldots, s_n) \in \mathcal{P}(\Pi_{sparc});$ for some $i, t = t_i, s = s_i;$
6 and t is not ground$\}$
7 **for each** *distinct sort name* s *occurring in* S **do**
8 $\Pi_{dlv} := \Pi_{dlv} \cup \{s(t).|t \in \mathcal{D}(s)\}$
9 Let rule r' be the result of adding all elements of S to the body of r
10 **if** r' *is a regular rule* **then**
11 Add r' to Π_{dlv}
12 **if** r' *is a cr-rule of the form* $q \stackrel{+}{\leftarrow}$ *body* **then**
13 Add to Π_{dlv} the following rules
`appl(`$rn(i, X_1, ..., X_n)$`)` $\vee$ $\neg$`appl(`$rn(i, X_1, ..., X_n)$`)` `:-` *body*`.`
`:~` `appl(`$rn(i, X_1, ..., X_n)$`),` *body*`.`
q `:-` `appl(`$rn(i, X_1, ..., X_n)$`),` *body*`.`
14 where $rn(i, X_1, ..., X_n)$ is the name of r

The intuitive idea behind the rules added to Π_{dlv} for a cr-rule at line 12 is the following: `appl(`$rn(i, X_1, ..., X_n)$`)` holds if the cr-rule r is used to obtain an answer set of the $\mathcal{SPARC}$ program; the first of the added rules says that r is either used or not used; the second rule, a weak constraint, guarantees that r is not used if possible, and the last rule

allows the use of r when necessary. The correctness of the algorithm is guaranteed by the following theorem whose complete proof can be found in our technical report [10].

Theorem 1. *A DLV program P_{dlv} obtained from a $\mathcal{SPARC}$ program P_{sparc} by the algorithm $Translate$ is a DLV counterpart of P_{sparc}.*

The translation can be used to compute answer set of $\mathcal{SPARC}$ program Π_{sparc} by using the DLV solver to compute answer sets of Π_{sparc}'s DLV counterpart and removing from them all auxiliary literals introduced in $Translate$.

Example 5 (Computing answer sets of a $\mathcal{SPARC}$ program).
To illustrate the translation and the computation of an answer set of a $\mathcal{SPARC}$ program, consider the input program Π_1 obtained from Π_0 by changing the type of one of its rules to consistency restoring:

```
#const n = 1.
sorts
#s1 = f(b).
#s2 = [0..n].
predicates
p(#s1,#s2).
rules
p(f(b),0).
:- not p(f(b),1).
p(X,X).
p(f(b),Y+1) :+ p(f(b),Y).
```

After the execution of the loop at line 3 of algorithm $Translate$, the first three regular program rules will be translated into

```
p(f(b),0).
:- not p(f(b),1).
p(X,X):-s1(X),s2(X).
s2(0). s2(1). s1(f(b)).
```

Assuming the only cr-rule is numbered by 0, it is translated as[1]:

```
appl(rn(0, Y))| -appl(rn(0, Y)) :- p(f(b),Y),s2(Y),s2(Y+1).
:~ appl(rn(0, Y)),p(f(b),Y),s2(Y),s2(Y+1).
p(f(b),Y+1):-appl(rn(0, Y)),p(f(b),Y),s2(Y),s2(Y+1).
```

Given the program resulted from $Translate$, DLV solver returns an answer set

$$\{s2(0), s2(1), s1(f(b)), p(f(b),0), appl(rn(0,0)), p(f(b),1)\}.$$

After dropping $appl(rn(0,0)), s2(0), s2(1), s1(f(b))$ from this answer set, we obtain an answer set $\{p(f(b),0), p(f(b),1)\}$ for the original program.

[1] The actual output result of the implemented version may be different because of variable renaming, change of the order of rules and shifting arithmetic terms.

6 Conclusion

As ASP has been employed to solve more and more problems, we believe constructs are needed to improve the productivity of ASP programmers. Particularly, constructs are needed to allow a programmer to better structure the program, to automatically determine some syntax and semantic errors and to avoid thinking about safety of ASP rules — non-declarative conditions on rules required by existing ASP systems. We define the syntax and semantics of a knowledge representation language $\mathcal{SPARC}$ which offers explicit constructs to specify objects, relations, and their sorts. The new language expands CR-Prolog — an extension of ASP by consistency restoring rules. We introduce an implementation of $\mathcal{SPARC}$ based on its translation to DLV with weak constraints. A simple syntax checking algorithm helps a programmer to avoid errors related to misspelling the names of objects and predicates as well as simple type errors. Another type checking algorithm flags program rules which, due to type conflicts, have no ground instantiations. We hope that the sort related algorithms presented in this paper will be eventually used to make $\mathcal{SPARC}$ a front-end for other ASP based systems (including CR-Prolog system CR-models [11]).

7 Acknowledgements

This work was partially supported by NSF grant IIS-1018031.

References

1. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of ICLP-88. (1988) 1070–1080
2. Syrjänen, T.: Lparse 1.0 user's manual (2000)
3. Balduccini, M.: Modules and signature declarations for a-prolog: Progress report. In: Software Engineering for Answer Set Programming Workshop (SEA07). (2007)
4. Balduccini, M., Gelfond, M.: Logic programs with consistency-restoring rules. In: International Symposium on Logical Formalization of Commonsense Reasoning, AAAI 2003 Spring Symposium Series. Volume 102., The AAAI Press (2003)
5. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic (TOCL) **7**(3) (2006) 499–562
6. Buccafurri, F., Leone, N., Rullo, P.: Strong and weak constraints in disjunctive datalog. In: Logic Programming And Nonmonotonic Reasoning. Springer (1997) 2–17
7. Balai, E., Gelfond, M., Zhang, Y.: SPARC – sorted ASP with consistency restoring rules. In: Answer Set Programming and Other Computing Paradigms. (2012)
8. SPARC system, http://www.depts.ttu.edu/cs/research/krlab/#software
9. Dechter, R.: Constraint Processing. Morgan Kaufmann, San Francisco, CA (2003)
10. Balai, E., Gelfond, M., Zhang., Y.: SPARC – sorted ASP with consistency restoring rules. http://www.depts.ttu.edu/cs/research/krlab/#papers, Technical Report, Texas Tech University, USA (2012)
11. Balduccini, M.: CR-MODELS: An Inference Engine for CR-Prolog. In Baral, C., Brewka, G., Schlipf, J., eds.: Proceedings of the 9th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'07). Volume 3662 of Lecture Notes in Artificial Intelligence., Springer (2007) 18–30