Extending and implementing RASP

Stefania Costantini
Università di L'Aquila
Stefania.Costantini@univaq.it

Andrea Formisano
Università di Perugia
formis@dmi.unipg.it

Davide Petturiti
Università di Perugia
davide.petturiti@dmi.unipg.it

Abstract

In previous work we have proposed an extension to ASP (Answer Set Programming), called RASP, standing for ASP with Resources. RASP supports declarative reasoning on production and consumption of (amounts of) resources. The approach combines answer set semantics with quantitative reasoning and relies on an algebraic structure to support computations and comparisons of amounts. The RASP framework provides some form of preference reasoning on resources usage. In this paper, we go further in this direction by introducing expressive constructs for supporting complex preferences specification on aggregate resources. We present a refinement of the semantics of RASP so as to take into account the new constructs. For all the extensions proposed, we provide an encoding into plain ASP. We prove that the complexity of establishing the existence of an answer set, in such an enriched framework, remains NP-complete as in ASP. Finally, we report on *raspberry*, a prototypical implementation of RASP. This tool consists of a compiler that, given a ground RASP program, produces a pure ASP encoding suitable to be processed by commonly available ASP-solvers.

Key words: Answer set programming, quantitative reasoning, preferences, language extensions.

1 Introduction

Preferences are an important issue, arising in many applications of non-monotonic reasoning and computational logic (for a recent overview, the reader may refer, e.g., to [7]). In fact, in many situations preferences strongly influence how an agent makes decisions and chooses a course of action. Preferences are often related to resources, as an agent may prefer to consume (or, more generally, to "invest") some resources rather than others, and may also have preferences about what one should try to obtain with the available resources.

Answer Set Programming (ASP) [16, 20, 23, 3, 1, 19, 27, 15] is a logic programming paradigm strongly oriented to knowledge representation and non-monotonic reasoning, with many significant applications in relevant domains, among which planning and configuration.

RASP, standing for ASP with Resources, is an extension of the ASP framework obtained by explicitly introducing the notion of *resource*. RASP allows one to specify resources and resource usage, and supports quantitative reasoning on consumption and production of amounts of resources. Resources are modeled by *amount-atoms* of the form $q\#a$, where q represents a specific type of resource and a denotes the corresponding amount, and can be produced or consumed (or declared available from the beginning). The processes that transform some amount of resources into other resources are specified by *r-rules*, for instance, as in the following simple example:

$$computer\#1 \leftarrow cpu\#1, harddisk\#2, motherboard\#1, ram\_module\#2.$$

where we model the fact that an instance of the resource *computer* (occurring in the *head* of the rule) can be obtained by "consuming" some other resources, in the amounts indicated in the *body* of the rule itself.

In their most general form, r-rules may involve regular ASP literals together with amount-atoms. Semantics for RASP programs is provided by combining the traditional semantics for ASP, i.e., answer set (or equivalently stable model) semantics with a notion of *allocation*. While stable models are used to deal with usual ASP literals, allocations are exploited to take care of amounts and resources. Intuitively, an allocation assigns a (possibly null) quantity to each amount-atom. Quantities are interpreted in an auxiliary algebraic structure that supports comparisons and operations on amounts. Admissible allocations are those satisfying, for all resources, the requirement that one can consume only what has been produced. Alternative allocations might be possible, as they correspond to different ways of using the same resources.

The original RASP framework supports limited forms of preference specification on resource usage [9] as the following simple example illustrates.

Example 1 *Assembling different PCs requires different sets of components (motherboard, processor(s), ram modules, etc.), depending on the kind of PC. As concerns servers, one might prefer SCSI disks rather than EIDE disks and vice versa for normal PCs. Here are the r-rules specifying how to obtain a main_unit from different combinations of components:*

$cpu\#7.$ $\quad scsihd\#15.$ $\quad eidehd\#9.$ $\quad motherboard\#7.$ $\quad ram\_module\#25.$
$main\_unit(server)\#1 \leftarrow cpu\#2,\ (scsihd\#4 > eidehd\#2),\ motherboard\#1,\ ram\_module\#4.$
$main\_unit(desktop)\#1 \leftarrow cpu\#1,\ (eidehd\#2 > scsihd\#2),\ motherboard\#1,\ ram\_module\#2.$

Notice that some resources can be declared as available from the beginning. This is done by means of *r-facts*, like in the first line of the above program.

A full description of the RASP approach can be found in [9, 10], where the reader can also find a detailed comparison with previous approaches to preference reasoning, as well as a discussion on related work involving the notion of resource in logic programming frameworks. We can say however that, referring to the approaches reported in [7], the strong point of RASP consists in allowing preferences on resources in the body of rules, instead than only in the head like in other approaches. This is relevant because, as observed in [7], preferences may depend on the current context, as what is preferred under certain circumstances may not be preferred in other situations.

In this paper, we further enrich the RASP language by introducing more expressive constructs to support the specification of compound resources and complex preferences. There are concrete reasons why complex preferences are needed. As emphasized for instance in [5], complex preferences are useful in many realistic optimization settings. Preferences may even be purely qualitative, but flexible ways of expressing preferences are needed, and it must be possible to combine them using various combination strategies. Preferences can be interpreted also as priorities. In fact, the motivating example discussed in [5] concerns scheduling problems to be solved according to both preferences and priorities. [7] discusses more generally the multifaceted relationship between non-monotonic logics and preferences, arguing that the non-monotonicity of reasoning itself is closely tied to preferences reasoners have on models of the world.

To focus on specific applications, we mention for instance Legal Reasoning. In recent research work concerning Law Argumentation Theory, see, e.g., [25] and [4], it is argued that there is a quest for preferences and priorities in this field: in fact, the application of legal rules is governed by priorities and exceptions to priorities, where priorities could change in different contexts. Even in the field of Bio-Informatics (for a survey the reader may refer, e.g., to [22]) one comes across complex preferences: e.g., certain protein residues "preferably" interact with particular nucleotides and, also, interactions with base display some strong complex preferences.

In [7] it is observed that "...commonsense reasoning [is] based on our inherent preference to assume that things, given what we know, are normal or as expected. This assumption allows us to form preferred belief sets, base our reasoning exclusively upon them, and ignore all other belief sets that are consistent with our incomplete knowledge but represent situations that are abnormal or rare". More generally, expressing preferences can be viewed as an indirect way of expressing preferences on belief sets in terms of the elements these belief sets contain. One may also want to represent conditional preferences so that beliefs can be accepted based on other beliefs already accepted or rejected. In this paper, we make a step ahead in this direction by introducing preferences which are not static (i.e., either hold or not hold) but rather are evaluated

in the current instance of the knowledge base. In particular, we introduce ways to specify if and when (i.e., in which contexts and circumstances) preferences should be applied.

The paper is organized as follows. In Section 2 we introduce the language of the extended RASP, and we illustrate the novel features by means of examples. In Section 3 we provide a formal semantics, by extending the semantics presented in [9, 10] so as to incorporate complex preferences. In Section 4 we consider the computational complexity of extended RASP. Computational complexity of RASP without preferences has been assessed in [10], by showing that the problem of establishing the existence of an answer set for a RASP program is NP-complete. For all the proposed extensions we provide an encoding into plain ASP and show that the enriched framework retains the same computational complexity: this guarantees that practical application of the approach is computationally affordable. In fact, complex preferences are not just "syntactic sugar", as the encoding into ASP is fairly complicated: therefore, providing a versatile programming tool at no additional complexity is one merit of our approach. We report in Sect. 5 on a prototypical implementation of RASP, publicly available online. Finally, we draw our conclusions.

2 The Language of RASP

In this section, we introduce the syntax of RASP and provide an intuitive semantics of its basic constructs. We borrow the basic notion from [10]. Then, we propose a significant extension to the RASP language.

The RASP underlying language is partitioned into *P*rogram symbols and *R*esource symbols. Precisely, let $\langle \Pi, \mathcal{C}, \mathcal{V} \rangle$ be an alphabet where $\Pi = \Pi_P \cup \Pi_R$ is a set of predicate symbols such that $\Pi_P \cap \Pi_R = \emptyset$, $\mathcal{C} = \mathcal{C}_P \cup \mathcal{C}_R$ is a set of symbols of constant such that $\mathcal{C}_P \cap \mathcal{C}_R = \emptyset$, and $\mathcal{V}$ is a set of symbols of variable.[1] The elements of $\mathcal{C}_R$ are said *amount-symbols* (*a-symbols*, for short), while the elements of Π_R are said *resource-predicates* (*r-predicates*). A *program-term* (*p-term*) is either a variable or a constant symbol. An *a-term* is either a variable or an a-symbol.

Let $\mathcal{A}(X, Y)$ denote the collection of all atoms $p(t_1, \ldots, t_n)$, with $p \in X$ and $\{t_1, \ldots, t_n\} \subseteq Y$. Then, a *p-atom* is an element of $\mathcal{A}(\Pi_P, \mathcal{C} \cup \mathcal{V})$. An *r-term* is an element of $\Pi_R \cup \mathcal{A}(\Pi_R, \mathcal{C} \cup \mathcal{V})$. An *a-atom* is a writing of the form $q\#a$ where q is an r-term and a is an a-term. We call *resource-symbols* (*r-symbols*) the ground r-terms, i.e. the elements of $\tau_R = \Pi_R \cup \mathcal{A}(\Pi_R, \mathcal{C})$.

Some examples: in the two expressions $p\#5$ and $q(3)\#b$, p and $q(3)$ are r-symbols (with $p, q \in \Pi_R$ and $3 \in \mathcal{C}$) aimed at defining two resources which are available in quantity 5 and b, resp. (with $5, b \in \mathcal{C}_R$ a-symbols). As the set of variables is not partitioned, the same variable may occur both as a p-term and as an a-term. Hence, we admit expressions such as $p(X)\#V$ where V, X are variables. In such cases, quantities are derived through instantiation (see Example 2).

Ground a-atoms contain no variables. A *program-literal* (*p-literal*, for short) L is a p-atom A or the negation *not* A of a p-atom (intended as negation-as-failure).[2] If $L = A$ (resp., $L = not\ A$) then $\overline{L}$ denotes *not* A (resp., A). Notice that, we do not allow negation of a-atoms (cf., [10] for a discussion on this point). Let I be a set of p-literals. We denote by I^+ the set of p-atoms $\{A \mid A \in I \text{ and } A \text{ is a p-atom}\}$ and by I^- the set of p-atoms $\{A \mid not\ A \in I \text{ and } A \text{ is a p-atom}\}$.

A-atoms are intended to model elementary resources. In order to represent collections of (somehow related) resources, we introduce two notions of "*compound resource*".

Definition 1 *Let $q_1\#a_1, \ldots, q_k\#a_k$ be a collection of $k > 0$ pairwise distinct a-atoms. Then $\{q_1\#a_1, \ldots, q_k\#a_k\}$ denotes a* conjunctive compound resource *(briefly,* c-resource*). Similarly, $\{q_1\#a_1; \ldots; q_k\#a_k\}$ denotes a* disjunctive compound resource *(briefly,* d-resource*).*

In what follows, we will often identify the singleton $\{q\#a\}$ by the single a-atom $q\#a$. A c-resource $\{q_1\#a_1, \ldots, q_k\#a_k\}$ represents a unique resource composed by k "elementary" resources (i.e., k a-atoms),

[1] We will partially discharge the requirement that $\mathcal{C}_P \cap \mathcal{C}_R = \emptyset$ in Section 4.

[2] We will only deal with negation-as-failure. Though, classical negation of program literals could be used in RASP programs and treated as usually done in ASP.

in the indicated quantities. Hence, a c-resource specifies a set of a-atoms that has to be conceptually considered in its entirety. A d-resource $d = \{q_1\#a_1; \cdots ; q_k\#a_k\}$ denotes a collection of a-atoms to be intended disjunctively: each occurrence/use of the resource d, corresponds to an occurrence/use of exactly one of its a-atoms. The choice is non-deterministic.

We anticipate here the notion of rule, in which a-atoms might occur to denote the use of resources. Such notion will be refined in the sequel (see Def. 6) after the introduction of more complex forms of literals. We distinguish between *p-rules* (plain ASP program rules, including the case of ASP *constraints*, i.e., rules with empty head) and *r-rules* which differ from p-rules in that they may contain a-atoms. *R-literals* and *r-rules* are so defined:

Definition 2 *An* r-literal *is either a p-literal or a compound resource (or an a-atom).*

An r-rule γ *has the form*

$$Idx: \quad H \leftarrow B_1, \ldots, B_m. \tag{1}$$

where $B_1, \ldots, B_m$ are r-literals, H is either a p-atom or a compound resource, and at least one a-atom occurs in γ (possibly in a compound resource). Idx is of the form $[N_{1,1}\text{-}N_{1,2}, \ldots, N_{h,1}\text{-}N_{h,2}]$, with $h \geqslant 1$, and each $N_{j,\ell}$ is a variable or a positive integer number.

Intuitively, in (1), when all the $N_{j,\ell}$s are integers, Idx denotes a set of positive integers (the union of h, possibly void, intervals in $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$). It is intended to restrain the number of times the r-rule γ can be used, i.e., *fired*. Such a number must belong to Idx or the rule cannot be fired at all.

In general, in Def. 2, each $N_{j,\ell}$ can possibly be a variable. Then, after grounding (see below), each $N_{j,\ell}$ has to be instantiated to a positive integer.

An *r-fact* is an r-rule having the particular form $q\#a \leftarrow$, where $q\#a$ is a ground a-atom. It models a fixed amount of resource q that is available "from the beginning".

A *rule* is either a p-rule or an r-rule. An *r-program* is a finite set of rules.

The Example 2 shows how variables allow to define resources in a parametric manner. The grounding phase will then introduce the actual r-symbols, by instantiating the variables (see also Remark 1 and Example 7 which elaborates the next example).

Example 2 *Let us refine the Example 1 on assembling different PCs, by considering that to compose a complete PC we need also a mouse, a monitor, and a keyboard. As before, different main units might be used. We complete the RASP specification of Example 1 by adding these r-rules.*

$$\begin{array}{ll} [1\text{-}N]: & computer(T)\#1 \leftarrow main\_unit(T)\#1,\ monitor\#1,\ mouse\#1, keyboard\#1, \\ & \qquad\qquad pc\_type(T),\ max\_need(T,N). \\ & pc\_type(server). \qquad pc\_type(desktop). \\ & monitor\#3. \qquad mouse\#8. \qquad keyboard\#4. \\ & max\_need(T,N) \leftarrow \ldots. \end{array}$$

By means of the first r-rule we are specifying a set of ground r-rules, one for each possible instantiation of T (and N). Namely, once grounded, fact $pc\_type(T)$ determines the kind of PC we are assembling. Consequently, one of the r-rules in Example 2 can be used to produce a main unit of the right type. Moreover, for each ground instantiation, the maximum number of firings for the first r-rule is determined through a fragment of program (omitted) defining $max\_need(T,N)$. Notice that, $max\_need(T,N)$ can be grounded by instantiating N and T to any constant symbol in the program. Nevertheless, the only significant instances (i.e., those that might be satisfied in a model) have T instantiated either to server or desktop and N instantiated to a positive integer.

Preferences on resource usage. We introduce now the possibility of expressing preferences among compound resources. To this aim we generalize the notion of preference list we previously introduced in [9].

Definition 3 *Let $s_1, \ldots, s_k$ be $k > 0$ compound resources. Then a* preference-list *(*p-list*, for short) is a writing of the form $s_1 > \cdots > s_k$. Each compound resource s_i has* degree of preference *i in the p-list.*

Plainly, a-atoms can be seen as particular p-lists made of singleton compound resources. Intuitively, a p-list $s_1 > \cdots > s_k$ describes a linear preference in choosing to use exactly one of the s_i. For instance, consider an r-rule involving a p-list $s_1 > \cdots > s_k$ where each s_i is a c-resource $\{q_{i,1}\#a_{i,1}, \ldots, q_{i,k_i}\#a_{i,k_i}\}$. The intended meaning is that, for all j, ℓ, $j < \ell$, it is preferred to produce/consume (by firing the r-rule) all the resources in $\{q_{j,1}\#a_{j,1}, \ldots, q_{j,k_j}\#a_{j,k_j}\}$ than all those in $\{q_{\ell,1}\#a_{\ell,1}, \ldots, q_{\ell,k_\ell}\#a_{\ell,k_\ell}\}$. Note that, by imposing $k_i = 1$, for all $i \in \{1, \ldots, k\}$, we obtain the simplified notion of p-lists as defined in [9], which does not cope with compound resources.

Example 3 *Consider the following substitute for the first r-rule of Example 2.*

$$\begin{aligned} computer(T)\#1 \leftarrow\ & main\_unit(T)\#1,\ pc\_type(T), \\ & \{wired\_mouse\#1,\ wired\_keyboard\#1\} > \{wireless\_mouse\#1,\ wireless\_keyboard\#1\}, \\ & \{monitorCRT\#1\ ;\ monitorLCD\#1\}. \end{aligned}$$

In this case, to assemble a PC we have two equally preferable possibilities in choosing the kind of monitor and a preference for wired equipment on wireless mouse and keyboard.

It is often the case that a preference among resources should be applied only when some precondition is satisfied. To model situations of such kind, we introduce *conditional p-lists*.

Definition 4 *A* conditional p-list *(*cp-list*, for short) is a writing of one of the following forms*

- $(r\ \ pref\_when\ \ L_1, \ldots, L_n)$, *or*
- $(r\ \ only\_when\ \ L_1, \ldots, L_n)$,

where $r = s_1 > \cdots > s_k$ is a p-list and $L_1, \ldots, L_n$ are p-literals.

Intuitively, a cp-list $(r\ \ pref\_when\ \ L_1, \ldots, L_n)$ specifies that one of the s_is has to be used (i.e., consumed or produced). The choice of which s_i is used is determined by the literals $L_1, \ldots, L_n$. Namely, if all $L_1, \ldots, L_n$ are satisfied, then the choice is ruled by the preference expressed through r. Otherwise, if any of the L_i is not satisfied, a non-deterministic choice among the s_is is made. Note that one of the s_i's is always used, regardless the satisfiability of $L_1, \ldots, L_n$.

As regards the cp-lists of the second form, $(r\ \ only\_when\ \ L_1, \ldots, L_n)$, they involve a different criterion in selecting the s_i. More specifically, if the cp-lists is used in a situation that satisfies all of the literals $L_1, \ldots, L_m$, then, as before, one of the resources in r is used and the choice is determined by the preference expressed through r. On the contrary, if the precondition is not satisfied, none of the resources s_i's is used.

Example 4 *Consider again the r-rules of the Examples 2 and 3. Suppose that we want to choose the kind of monitor depending on the dimension of our desk: we can replace the r-rule that defines how to produce a computer by the following one (where the program atom little_space, defined elsewhere in the program, is true when there is little free space on the desk).*

$$\begin{aligned} computer(T)\#1 \leftarrow\ & main\_unit(T)\#1,\ mouse\#1,\ keyboard\#1,\ pc\_type(T), \\ & (monitorLCD\#1 > monitorCRT\#1\ pref\_when\ \ little\_space). \end{aligned}$$

By exploiting variables and cp-lists we can also replace the two r-rules specifying the production of main units with the following single r-rule:

$$\begin{aligned} main\_unit(T)\#1 \leftarrow\ & cpu\#1,\ motherboard\#1,\ ram\_module\#2, \\ & (\{scsihd\#4,\ cpu\#1,\ ram\_module\#2\} > \{eidehd\#2,\ cpu\#1,\ ram\_module\#2\} \\ & \qquad only\_when\ \ T = server), \\ & (eidehd\#2 > scsihd\#2\ only\_when\ \ T \neq server),\ pc\_type(T). \end{aligned}$$

After grounding, depending on the specific instantiation of T, the above r-rule surrogates the two r-rules of Example 1, because in any case exactly one among the cp-lists is active. For instance, if $T \neq server$, all the a-atoms in the first cp-list are ignored, while the second cp-list translates into the p-list eidehd#2>scsihd#2.

Expressing arbitrary preferences. In general, there might be cases in which useful (conditional) preferences are not expressible as a linear order on a set of alternatives. Moreover, preferences might depend on specific contextual conditions that are not foreseeable in advance.

A simple example: next summer Jack would like to visit a foreign country. It might be that his sister Jill joins, but only if she does not get a job for the summer. Jack would like to go either to Brazil, France, Spain, Norway, or Iceland. He prefers Brazil the most. In case going to Brazil is not possible, he considers equally interesting visiting either France or Spain. The least preferred options are Norway and Iceland, but no preference is expressed between them, except in August, when Norway is preferred. This simple case of preference order, being not linear, cannot be directly modeled by p-lists.

P-sets are a generalization of p-lists that allows one to use any binary relation (not necessarily a partial order) in expressing (collections of alternative) p-lists.

Definition 5 *Let $q_1 \# a_1, \ldots, q_k \# a_k$ be $k > 0$ a-atoms and let pred be a binary program predicate. A* p-set *is of the form*

$$\{q_1 \# a_1, \ldots, q_k \# a_k \mid pred\}.$$

A p-set may occur in any place where an r-literal does. The program predicate *pred* is supposed to be defined elsewhere in the program where the p-set occurs.

The intuitive semantics of a p-set can be grasped by considering a particular extension for the predicate *pred* (namely, a set of pairs $\langle a, b \rangle$ assumed to be true in a certain situation, i.e., a certain model of the program, see below). Let X be the set of r-symbols $\{q_1, \ldots, q_k\}$. Consider the binary relation $R \subseteq X^2$ obtained by restricting to X the extension of *pred*. R is interpreted as a preference relation over X: namely, for any $q_i, q_j \in X$ the fact that $\langle q_i, q_j \rangle \in R$ models a preference of q_i on q_j. The case of p-lists is a particular case of p-sets, obtained when R describes a total order.[3]

As mentioned, R does not need to be a partial order, e.g., for instance, it may imply cycles. In such cases, those resources that belong to the same cycle in R are considered equally preferable (e.g., France and Spain, in the above example). On the other hand, R might be a partial relation. So, there might exist elements on X that are incomparable (e.g., Norway and Iceland in July).

Because of the presence of incomparable resources and equivalent resources, R can be seen as a representation of a collection of p-lists, one for each possible total order on X compatible with R. In particular, in case of equally preferable options, a non-deterministic choice is made. Instead, in case of incomparable options one among the possible total ordering of these options is arbitrarily selected. In the above example, the extension of *pred* should include the pairs $\langle Brazil, France \rangle$, $\langle France, Spain \rangle$, $\langle France, Iceland \rangle$, $\langle France, Norway \rangle$, and $\langle Spain, France \rangle$ (plus $\langle Norway, Iceland \rangle$, if is it August). Consequently, the admissible linear orders, if it is not August, are:

$$Brazil{>}Spain{>}Norway{>}Iceland. \qquad Brazil{>}France{>}Norway{>}Iceland,$$
$$Brazil{>}Spain{>}Iceland{>}Norway, \qquad Brazil{>}France{>}Iceland{>}Norway,$$

while only the first two should be considered in August.

At this point, we are ready to revise the definitions of r-literal and r-rule (Def. 2), by considering p-lists, cp-lists, and p-set, as r-literals:

Definition 6 *An* a-literal *is either an a-atom, a compound resource, a p-list, a cp-list, or a p-set.*

An r-literal *is either a p-literal or an a-literal.*

An r-rule γ *has the form*

$$Idx: \quad H \leftarrow B_1, \ldots, B_m. \tag{2}$$

where $B_1, \ldots, B_m$ are r-literals, H is either a p-atom or a sequence of a-literals (provided that at least one a-atom occurs in γ). Idx is as before.

[3]Note that we are introducing a change in the syntax of RASP, by admitting a limited use of r-symbols as arguments of p-literals.

Example 5 *Let us consider the example mentioned above. This is a fragment of r-program describing Jack's preferences on buying plane tickets:*

$$
\begin{array}{l}
jack\_happy \leftarrow \{ticket(b)\#N, ticket(f)\#N, ticket(s)\#N, ticket(i)\#N, ticket(n)\#N \\
\qquad\qquad | \; jack\_pref\}, is\_summer, num\_of\_tickets(N). \\
jack\_pref(ticket(b), ticket(f)). \quad jack\_pref(ticket(f), ticket(i)). \\
jack\_pref(ticket(f), ticket(s)). \quad jack\_pref(ticket(s), ticket(f)). \\
jack\_pref(ticket(f), ticket(n)). \quad jack\_pref(ticket(n), ticket(i)) \leftarrow is\_august. \\
ticket(P)\#N \leftarrow money\#C, cost(P, C1), C = C1 * N, num\_of\_tickets(N). \\
cost(b, 9). \quad cost(f, 6). \quad cost(s, 5). \quad cost(i, 5). \quad cost(n, 6). \\
num\_of\_tickets(1) \leftarrow jill\_works. \qquad num\_of\_tickets(2) \leftarrow not\ jill\_works.
\end{array}
$$

Here is another example involving different forms of preference specification:

Example 6 *Assume that, in making a cake or some cookies, you might choose among different ingredients, and you have to consider some constraints due to possible allergy or diet. Note that the preference among chocolate, nuts and coconut, i.e., a particular p-list, is determined depending on the extension of the predicate less_caloric, which might be different for different answer sets and has to be established dynamically (through the predicate calory, defined elsewhere in the program).*

$$
\begin{array}{l}
cake\#1 > cookie\#15 \leftarrow egg\#2, flour\#2, raisin\#4, \\
\qquad (\{aspartame\#1, skim\_milk\#6\} > \{sugar\#4, whole\_milk\#6\}\ pref\_when\ \ diet), \\
\qquad (\{vanilla\#1; lemon\#2\} > cinnamon\#1\ \ only\_when\ \ not\ \ allergy), \\
\qquad \{chocolate\#2, nuts\#1, coconut\#1 \mid less\_caloric\}. \\
less\_caloric(X, Y) \leftarrow calory(X, A), calory(Y, B), A < B. \\
calory(X, Y) \leftarrow ...
\end{array}
$$

Remark 1 *The grounding of an r-program P is the set of all ground instances of rules (and facts) of P, obtained through ground substitutions over the constants occurring in P.*[4] *Notice that in any r-program only a finite number of a-symbols of $\mathcal{C}_\mathcal{R}$ occurs, also because all r-facts must be ground. Hence, as far as a-atoms are concerned, a finite number of ground instances can be generated by the grounding process. This is because all instances of a-terms are among the instances of the terms occurring in p-atoms. A "smart" grounder for RASP would avoid generating instances of r-rule where variables occurring as a-terms are instantiated to constants of $\mathcal{C}_\mathcal{P}$ instead of constants of $\mathcal{C}_\mathcal{R}$. Such "wrong" instances are however both semantically and practically irrelevant (apart from the waste of space).*

Example 7 *Let us consider the r-rules in Example 2. Assume that the predicates max_need and pc_type are defined by these facts:*

$$
\begin{array}{lll}
max\_need(server, 3). & max\_need(desktop, 5). & pc\_type(server). \\
max\_need(laptop, 2). & max\_need(netbook, none). & pc\_type(desktop).
\end{array}
$$

Then, a "smart" grounder would produce the following instances of the first r-rule of Example 2:

$$
\begin{array}{ll}
[1\text{-}3]: & computer(server)\#1 \leftarrow main\_unit(server)\#1, monitor\#1, mouse\#1, keyboard\#1, \\
 & \qquad pc\_type(server), max\_need(server, 3). \\
[1\text{-}5]: & computer(desktop)\#1 \leftarrow main\_unit(desktop)\#1, monitor\#1, mouse\#1, keyboard\#1, \\
 & \qquad pc\_type(desktop), max\_need(desktop, 5). \\
[1\text{-}2]: & computer(laptop)\#1 \leftarrow main\_unit(laptop)\#1, monitor\#1, mouse\#1, keyboard\#1, \\
 & \qquad pc\_type(laptop), max\_need(laptop, 2).
\end{array}
$$

This is so because the value none is not a positive integer, so the grounder can avoid generating the r-rule with $N = none$. Notice that also the last r-rule is useless because there is no model of the program satisfying the

[4] As it is well-known, at present, almost all ASP solvers perform a preliminary grounding step as they are able to find the answer sets of ground programs only. Work is under way to overcome at least partially this limitation (cf., [12], for instance).

fact pc_type(laptop). Then, the grounding of the program can be simplified by omitting such ground instance. (Actually, some of the currently available grounders [2] for ASP are able to apply this sort of simplifications, in order to obtain smaller ground programs.)

3 Semantics of RASP

In this section we define the formal semantics of r-programs reflecting the intuitive description outlined in the previous sections.

Semantics of a (ground) r-program is determined by interpreting p-literals as usually done in ASP (by exploiting stable model semantics) and a-atoms in an auxiliary algebraic structure Q. In principle, such a structure can be of full generality, provided that it supports operations and comparisons among amounts. The rationale behind the proposed semantic definition is the following. On the one hand, we translate r-rules into a fragment of a plain ASP program, so that we do not have to modify the definition of stability. This is of some importance in order to make the several theoretical and practical advances in ASP still available for RASP. On the other hand, an interpretation involves the allocation of actual quantities to a-atoms. In fact, this allocation is one of the components of an interpretation: an answer set of an r-program models an r-rule only if it is satisfied (in the usual way, relying on stable model semantics) as concerns its p-literals, and the correct amounts are allocated for the a-atoms. A last component of an interpretation copes with the repeated firing of a rule: in case of several firings, the resource allocation must be iterated accordingly.

In order to define semantics of r-programs, we have to fix an interpretation for a-symbols. This is done by choosing a collection Q of *quantities*, and the operations to combine and compare quantities. A natural choice is $Q = \mathbb{Z}$: thus, we consider given a mapping $\kappa : \mathcal{C}_R \to \mathbb{Z}$ that associates integers to a-symbols. Positive and negative integers will be used to model produced and consumed amounts, respectively. For the sake of simplicity, in what follows we adopt a simplification by identifying $\mathcal{C}_R$ with $\mathbb{Z}$ (and κ being the identity). This will not cause loss in the generality of the treatment.

Notation. Before going on, we introduce some useful notation. Given two sets X, Y, let $\mathcal{FM}(X)$ denote the collection of all finite multisets of elements of X.[5] Moreover, let Y^X denote the collection of all (total) functions having X and Y as domain and codomain, respectively. For any (multi)set Z of integers, $\sum(Z)$ denotes their sum. (E.g., $\sum(\{\!\{2, 5, 3, 3, 5\}\!\}) = 18$ and $\sum(\{\!\{\}\!\}) = 0$.) Given a collection S of (non-empty) sets, a *choice function* $c(\cdot)$ for S is a function having S as domain and such that for each s in S, $c(s)$ is an element of s. In other words, $c(\cdot)$ chooses exactly one element from each set in S.

Let p be a binary predicate symbol p and I a set of ground p-atoms. Consider all the atoms of the form $p(a, b)$ in I. Let $I_{|\,p}$ denote the transitive closure of the set $\{p(a, b) \mid p(a, b) \in I\}$ (namely, the smallest set such that $\big(p(a, b) \in I \vee (p(a, c) \in I_{|\,p} \wedge p(c, b) \in I_{|\,p})\big) \to p(a, b) \in I_{|\,p}$ for all a, b, c).

To deal with the disjunctive aspect of some of the a-literals (e.g., p-lists, d-resources, p-sets) and to model the degrees of preference, we mark a-atoms with integer indices. In presence of an a-literal ℓ involving choices and preferences, each allocation of resources might assign not null amounts to (some of) the a-atoms occurring in ℓ. We collect the indices of these a-atoms, for all the r-rules of an r-program, to record the degrees of preferences of all choices made by the allocation at hand. We remark that the introduction of such indices is not essential for defining of the notions of interpretation and answer set of an r-program (Def. 8, to be seen). Instead, as we will see, they are exploited to reflect preferences expressed by p-lists, so as to induce a (partial) order on the answer sets of an r-program. Then, the alternative allocations can be ranked by some preference criterion which exploits such collections of indices in order to determine which are the most preferred ones (cf., Remark 2).

The values of the indices assigned to a-atoms depend on the kind of language construct. For instance, in case of a p-list $q_1\#a_1 > \cdots > q_k\#a_k$, its composing a-atoms are associated, from left to right, with successive indices starting from 1, so that an index valued i corresponds to the selection of the i-th a-atom. For the

[5] A multiset (or bag) is a generalization of a set, where repeated elements are allowed. Then, a member of a multiset can have more than one membership.

sake of uniformity, indices are also assigned to those a-atoms that are not subject to preferences (as in the case of single a-atoms or c-resources not occurring in a p-list). In such cases a null index is used. In general, indices are assigned through two mappings (*setify* and *ch*, to be seen) in such a way that, for each r-rule γ, the number of not null indices corresponds to the number of choices made in allocating resources for a-atoms of γ. The values of the indices represent the degrees of preference.

In this frame, any a-atom will be interpreted as a pair in $\mathbb{N} \times Q$ that we call an *amount couple*. For example: an interpretation for *skim_milk*#2>*whole_milk*#2, occurring in the head of an r-rule, will involve one of the couples $\langle 1, 2\rangle$ and $\langle 2, 2\rangle$, where the first components of the couples reflect the degree of preference and the second elements are the quantities. For single a-atoms, such as *egg*#2, in a body of an r-rule, no preference is involved and a potential interpretation is $\langle 0, -2\rangle$.

Interpretation of RASP Programs. An interpretation for an r-program P must determine an allocation of amounts for all occurrences of such amount symbols in P. We represent produced quantities (i.e., a-atoms in heads) by positive values, while negative values model consumed amounts (i.e., a-atoms in bodies). For each r-symbol q, the overall sum of quantities allocated to (produced and consumed) a-atoms of the form $q\#a$ must not be negative. The collection $\mathbb{S}_P$ of all potential allocations (i.e., those having a non-negative global balance)—for any single r-symbol occurring in P (considered as a set of rules)—is the following collection of mappings:

$$\mathbb{S}_P = \left\{ F \in (\mathcal{FM}(\mathbb{N} \times Q))^P \;\middle|\; 0 \leqslant \sum \Big(\bigcup_{\gamma \in P} \{\!\!\{ x \mid \langle n, x\rangle \text{ in } F(\gamma) \}\!\!\} \Big) \right\} \tag{3}$$

The rationale behind the definition of $\mathbb{S}_P$ is as follows. Let q be a fixed r-symbol. Each element $F \in \mathbb{S}_P$ is a function that associates to every rule $\gamma \in P$ a (possibly empty) multiset $F(\gamma)$ of amount couples, assigning certain quantities to each occurrence of a-atoms of the form $q\#a$ in γ. All such Fs satisfy (by definition of $\mathbb{S}_P$) the requirement that, considering the entire P, the global sum of all the quantities F assigns is non-negative. As we will see later, only some of these allocations will actually be acceptable as a basis for a model.

An r-interpretation of the amount symbols in a ground r-program P is defined by providing a mapping $\mu : \tau_R \to \mathbb{S}_P$. Such a function determines, for each r-symbol $q \in \tau_R$, a mapping $\mu(q) \in \mathbb{S}_P$. In turn, this mapping $\mu(q)$ assigns to each rule $\gamma \in P$ a multiset $\mu(q)(\gamma)$ of quantities, as explained above. The use of multisets allows us to handle multiple copies of the same a-atom: each of them corresponds to a different amount of resource to be taken into account. We have the following definition of *r-interpretation*:

Definition 7 *An* r-interpretation *for a (ground) r-program P is a triple $\mathcal{I} = \langle I, \mu, \xi\rangle$, with $I \subseteq \mathcal{A}(\Pi_P, \mathcal{C})$, $\mu : \tau_R \to \mathbb{S}_P$, and ξ a mapping $\xi : P \to \mathbb{N}^+$.*

In Def. 7, I plays the role of a usual answer set assigning truth values to p-literals; μ describes an allocation of resources; ξ associates to each rule the number of times the rule is used. By little abuse of notation, we consider ξ to be defined also for p-rules and r-facts. For this kind of rules we assume the interval $Idx = [1\text{-}1]$ as implicitly specified in the rule definition.

The firing of an r-rule is enabled only if the truth values of the p-literals satisfy the rule. We reflect this fact by introducing a suitable fragment of ASP program $\widehat{\gamma}$. Let the r-rule γ have $L_1, \ldots, L_k$ as p-literals and $R_1, \ldots, R_h$ as a-literals. Then, $\widehat{\gamma}$ is so defined:

$$\widehat{\gamma} = \begin{cases} \{ \leftarrow \overline{L_1}, \ldots, \leftarrow \overline{L_k} \} & \text{if the head of } \gamma \text{ is made of a-literals} \\ \{ \leftarrow \overline{L_1}, \ldots, \leftarrow \overline{L_k}, & \text{if } \gamma \text{ has the p-atom } H \text{ as head} \\ \quad H \leftarrow L_1, \ldots, L_k \} & \quad \text{and } h > 0 \\ \{ \gamma \} & \text{otherwise (e.g., } \gamma \text{ is a p-rule).} \end{cases}$$

Def. 8, to be seen, states that in order to be a model, an r-interpretation that allocates non-void amounts to the r-symbols of γ, has to model the ASP-rules in $\widehat{\gamma}$. So far we have developed a semantic structure in which r-rules are interpretable by singling-out suitable collections of amount couples. Different ways of allocating amount of resources to an r-program are possible. Each allocation reflects, for each p-list r in P,

one of the admissible choices that r represents. In order to denote such admissible choices we need some further notation. Let j be an integer and let ℓ be a compound resource (recall that this includes the case of a-atoms). Let

$$setify(\ell, j) = \begin{cases} \{\{\langle j, q_1, a_1\rangle, \langle 0, q_1, a_1\rangle, \ldots, \langle 0, q_k, a_k\rangle\}\} & \text{if } \ell \text{ is } \{q_1\#a_1, \ldots, q_k\#a_k\} \\ \{\{\langle j, q_1, a_1\rangle\}, \ldots, \{\langle j, q_k, a_k\rangle\}\} & \text{if } \ell \text{ is } \{q_1\#a_1; \ldots; q_k\#a_k\} \end{cases} \quad (4)$$

We exploit $setify(\ell, j)$ to represent the collection of all possible choices in allocating amounts to the r-symbols occurring in an r-rule. Observe the difference between the two cases in (4). If ℓ is a c-resource then a unique choice is possible: all the k a-atoms have to be allocated. Otherwise, if ℓ is a d-resource, a single a-atom has to be chosen and there are k viable alternatives. *setify* assigns indices to all a-atoms, but for each choice at most one a-atom is assigned the given value j.[6] Such value is possibly not null and determined as explained below. Let us consider each different form of a-literal. In what follows I is a set of ground p-atoms:

- For compound resources and p-lists we define:[7]

$$ch(\ell, I) = \begin{cases} \{setify(\ell, 0)\} & \text{if } \ell \text{ is a compound resource} \\ \{setify(s_1, 1), \ldots, setify(s_k, k)\} & \text{if } \ell \text{ is } s_1 > \cdots > s_k \end{cases} \quad (5)$$

We use *ch* and *setify* to represent the a-atoms of r-rules as triples denoting: the position in each preference list where they occur; the r-symbol they contain; the amount that is required for this r-symbol. In case of a p-list, each degree of precedence is assigned, through *setify*, to one of the selectable a-atoms. Let us consider the remaining types of a-literal:

- Due to the presence of the preconditions, the representation of conditional p-lists is parameterized by a set I of p-atoms. Let $r = s_1 > \cdots > s_k$ *pref_when* $L_1, \ldots, L_n$, where each s_i is a compound resource. We put:

$$ch(r, I) = \begin{cases} ch(s_1 > \cdots > s_k, \emptyset) & \text{if } \{L_1, \ldots, L_n\}^+ \subseteq I \wedge \{L_1, \ldots, L_n\}^- \cap I = \emptyset \\ ch(\{s_1; \ldots; s_k\}, \emptyset) & \text{otherwise} \end{cases} \quad (6)$$

- Similarly, concerning cp-lists of the form $r = s_1 > \cdots > s_k$ *only_when* $L_1, \ldots, L_n$, we put:

$$ch(r, I) = \begin{cases} ch(s_1 > \cdots > s_k, \emptyset) & \text{if } \{L_1, \ldots, L_n\}^+ \subseteq I \wedge \{L_1, \ldots, L_n\}^- \cap I = \emptyset \\ \emptyset & \text{otherwise} \end{cases} \quad (7)$$

- Since a p-set represents a collection of alternative p-lists on the same set of a-atoms, the treatment is lightly more complex because all such alternatives are potentially selectable. Moreover, for each of these p-lists, a choice among its a-atoms must be made. Let $ps = \{q_1\#a_1, \ldots, q_k\#a_k \mid p\}$. Given a set I of ground p-atoms, let us denote the set of p-lists represented by ps as follows:

$$PLists(ps, I) = \{q_{i_1}\#a_{i_1} > \cdots > q_{i_n}\#a_{i_n} \mid \langle i_1, \ldots i_n\rangle \text{ is a maximal prefix of } \langle 1, \ldots k\rangle \text{ such that } \forall j, h\, (j < h \rightarrow p(q_{i_h}, q_{i_j}) \notin I_{|p})\}$$

  Then, we characterize the collection of possible choices in allocating resources for a p-set, as follows:

$$ch(ps, I) = \bigcup_{pl \in PLists(ps, I)} ch(pl, I) \quad (8)$$

[6] Incidentally, we observe that to make the definition of $setify(\ell, j)$ deterministic in selecting the a-atom to be assigned the index j, one can assume the set of all a-atoms to be totally ordered and select the minimal a-atom of the c-resource.

[7] Note that when ℓ is a compound resource or a p-list, $ch(\ell, I)$ does not depend on I. This is not the case for the other forms of a-literals.

The definition of ch is then extended to heads and bodies of rules by putting:

$$\begin{array}{rcl} ch_h(\gamma, I) & = & \{\!\{\, x \mid x = ch(\ell, I), x \neq \emptyset, \ell \text{ in } r\text{-}head(\gamma) \,\}\!\} \\ ch_b(\gamma, I) & = & \{\!\{\, x \mid x = ch(\ell, I), x \neq \emptyset, \ell \text{ in } r\text{-}body(\gamma) \,\}\!\} \end{array}$$

(Notice that we exclude the empty sets of choices possibly introduced by (7).)

Finally, we associate to each r-rule γ, the following set $\mathcal{R}(\gamma, I)$ of multisets. Each element of $\mathcal{R}(\gamma, I)$ represents a possible admissible selection of a-atoms for each of the a-literals in γ and an actual allocation of amounts (taken in Q via the function κ) to the a-symbols occurring in them. Notice that quantities associated to a-atoms occurring in the body of γ are negative, as these resources are consumed.[8] Vice versa, quantities associated to a-atoms of the head are positive, as these resources are produced. (Note that also this definition is parameterized by a set I of p-atoms.)

$$\begin{array}{rcl} \mathcal{R}(\gamma, I) & = & \Big\{ \{\!\{\, \langle i, q, \kappa(a)\rangle \mid \langle i, q, a\rangle \text{ in } c_1(S_1) \text{ and } S_1 \text{ in } ch_h(\gamma, I) \,\}\!\} \\ & & \quad \cup\ \{\!\{\, \langle i, q, -\kappa(a)\rangle \mid \langle i, q, a\rangle \text{ in } c_2(S_2) \text{ and } S_2 \text{ in } ch_b(\gamma, I) \,\}\!\} \\ & & \quad \mid \text{ for } c_1 \text{ and } c_2 \text{ choice functions for } ch_h(\gamma, I) \text{ and } ch_b(\gamma, I), \text{ resp.} \Big\} \end{array}$$

where c_1 (resp. c_2) ranges on all possible choice functions for $ch_h(\gamma)$ (resp. for $ch_b(\gamma)$).

To account for multiple firings, we need to be able to "iterate" the allocation of quantities for a number n of times: to this aim, for any $n \in \mathbb{N}^+$ and $q \in \tau_R$, let

$$\mathcal{R}^n(\gamma, I) \ = \ \Big\{ \bigcup \{\!\{ X_1, \ldots, X_n \}\!\} \mid \{\!\{ X_1, \ldots, X_n \}\!\} \in \mathcal{FM}\big(\mathcal{R}(\gamma, I)\big) \Big\}$$

and

$$\mathcal{R}^n(q, \gamma, I) \ = \ \Big\{ \{\!\{\, \langle i, v\rangle \mid \langle i, q, v\rangle \text{ is in } X \,\}\!\} \mid X \in \mathcal{R}^n(\gamma, I) \Big\}.$$

While, given I, $\mathcal{R}(\gamma, I)$ represents all the different choices for allocating resources to γ, the collection $\mathcal{R}^n(\gamma, I)$ represents all the possible ways of making n times these choices (possibly, in different manners). Fixed I and an r-symbol q, the set $\mathcal{R}^n(q, \gamma, I)$ extracts from each alternative in $\mathcal{R}^n(\gamma, I)$ the multiset of amount couples relative to q. We have the following definition:

Definition 8 *Let $\mathcal{I} = \langle I, \mu, \xi\rangle$ be an r-interpretation for a (ground) r-program P. $\mathcal{I}$ is an* answer set *for P if the following conditions hold:*

- *for all rules $\gamma \in P$*

$$\Big(\forall\, q \in \tau_R \ \big(\mu(q)(\gamma) = \emptyset\big)\Big) \vee \Big(\forall\, q \in \tau_R \ \big(\mu(q)(\gamma) \in \mathcal{R}^{\xi(\gamma)}(q, \gamma, I)\big) \wedge \big(N_{1,1} \leqslant \xi(\gamma) \leqslant N_{1,2}\big)\Big) \tag{9}$$

- *I is a stable model for the ASP-program $\widehat{P}$, so defined*

$$\widehat{P} = \bigcup \left\{ \ \widehat{\gamma} \ \middle| \ \begin{array}{l} \gamma \text{ is a p-rule in } P, \ \text{ or} \\ \gamma \text{ is a r-rule in } P \text{ and } \exists\, q \in \tau_R \ \big(\mu(q)(\gamma) \neq \emptyset\big) \end{array} \right\}$$

The two disjuncts in Def. 8 correspond to the two cases: a) the rule γ is not fired, so null amounts are allocated to all its a-symbols; b) the rule γ is actually fired $\xi(\gamma)$ times and all needed amounts are allocated (by definition this happens if and only if $\exists\, q \in \tau_R \ \big(\mu(q)(\gamma) \neq \emptyset\big)$ holds). Note that case b) imposes that the amount couples assigned by μ to a resource q in a rule γ reflect one of the possible choices in $\mathcal{R}^{\xi(\gamma)}(q, \gamma, I)$. Moreover, in Def. 8 we use the set $\mathcal{R}^n(q, \gamma, I)$ to impose restrictions on the global resource balance, for each q. Observe that the specific stable model I at hand is used as parameter in $\mathcal{R}^n(q, \gamma, I)$ to determine the correct collection of choices of the cp-lists and the p-sets occurring in P.

By summarizing the above discussion, we can say that $\mathcal{I}$ is an *answer set* of P if:

[8] To be precise, the assigned quantity corresponds to the negation, in Q, of the amount occurring in an a-atom of the body. One may also specify negative *byproducts* in the body, which are produced and not consumed: in such a case, the assigned quantity will be positive.

- it satisfies all the p-rules in P and all the fired r-rules (in the usual way) as concerns their p-literals;
- for each a-literal occurring in a fired r-rule, $\mathcal{I}$ chooses one of the possible allocations of amounts; the correct amounts are assigned to the chosen a-atoms, while null amounts are assigned to the others;
- the global balance of each r-symbol is not negative (i.e., all consumed amounts have been also produced by rule firings or available from r-facts).

Finally, we say that $\mathcal{I}$ is an answer set of an r-program P if it is an answer set for the grounding of P.

Remark 2 *Different allocations originate different answer sets. To impose a preference order on such answer sets, any* preference criterion *can be used. Such a criterion should order the collection of answer sets by reflecting the (preference degrees in the) p-lists. Any criterion has to take into account that each rule determines a (partial) preference ordering on answer sets, and it should aggregate/combine all such "local" partial orders to obtain a global one. This yields the notion of* most preferred answer set *of an r-program. Simple criteria to rank the collection of answer sets are described in [9, 6] (to which the reader is referred for more details on this issue). We recall here the simplest one, to emphasize the use of the indices and the amount couples that constitute the r-interpretation. In this criterion we directly exploit the degrees of preference of a-atoms. For any multiset m in $\mathcal{FM}(\mathbb{N}\times Q)$ and $i \in \mathbb{N}$, let be $\beta_i(m) = |\{\!\{ \langle i,v\rangle \mid \langle i,v\rangle \text{ is in } m \}\!\}|$. A partial order on answer sets can be defined as follows. Given two answer sets $\mathcal{I}_1 = \langle I_1, \mu_1, \xi_1\rangle$ and $\mathcal{I}_2 = \langle I_2, \mu_2, \xi_2\rangle$ for an r-program P, with $\mu_1 \neq \mu_2$, let m_i be the multiset $m_i = \bigcup_{\gamma\in P,\, q\in\tau_R} \mu_i(q)(\gamma)$, for $i \in \{1,2\}$, and let j be the minimum natural number such that $\beta_j(m_1) \neq \beta_j(m_2)$. We put $\mathcal{I}_1 \prec_1 \mathcal{I}_2$ if and only if $\beta_j(m_1) > \beta_j(m_2)$.*

The preference criterion $\mathcal{PC}_1$ states that $\mathcal{I}_1$ is preferred to $\mathcal{I}_2$ if it holds that $\mathcal{I}_1 \prec_1 \mathcal{I}_2$. The preferred answer sets *with respect to $\mathcal{PC}_1$ are those answer sets that are $\prec_1$-minimal. In a sense, the criterion $\mathcal{PC}_1$ has a "positional flavor": the answer sets that selects the highest possible number of leftmost elements (in the p-lists) are preferred.*

Remark 3 *We conclude this section by observing that the semantics described here is a proper generalization of the one defined in [9, 10] which does not deal with compound resources, p-lists, cp-lists, and p-sets. It is immediate to see that the two semantics coincide on basic-RASP programs, i.e., those not involving these forms of a-literals. Consequently, for basic-RASP programs we can plainly inherit the results on computational complexity obtained in [10]. In the following sections we exploit this observation for assessing the complexity of RASP extended with the new constructs introduced so far.*

4 An ASP encoding of RASP

In this section, we provide an abstract implementation of RASP by devising an encoding of (ground) r-programs into pure ASP programs. We start by recalling the encoding for basic RASP described in [10]. Then, we show that such an encoding can be refined in order to deal with all the forms of literal introduced in Section 2.

Let $\Pi_{\mathcal{T}}$ be a set of fresh p-symbols with $\{$*notfired*, *fired*, *use*, *r_rule*, *a_atom*, *res_symb*, *counter*, *notcounter*$\} \subseteq \Pi_{\mathcal{T}}$. Given a ground r-program S, let $S_{\mathcal{R}} \subseteq S$ be the set of r-rules in S and let $S_{\mathcal{P}} = S \setminus S_{\mathcal{R}}$ (i.e., the p-rules). For any set of ASP rules X, let $atoms(X)$ denote the set of all atoms occurring in X. For any ASP rule γ, let $lits^+(\gamma)$ (resp., $lits^-(\gamma)$) be the set of atoms occurring positively (resp., negatively) in the body of γ. Moreover, let $head(\gamma)$ be the set of atoms occurring in the head of γ (actually, it is a singleton). A translation $\mathcal{T}$ from RASP into ASP is defined as follows.

Let us start by treating the case of plain r-rules, i.e., those involving simple a-atoms only and devoid of p-lists.

4.1 Treatment of plain r-rules

Each r-rule γ in $S_{\mathcal{R}}$ is univocally named by introducing a fresh constant symbol r_γ and the fact:

$$r\_rule(r_\gamma). \tag{10}$$

Let the r-rule γ be of the form

$$Idx: \quad q_0\#a_0, \ldots, q_h\#a_h \leftarrow q_{h+1}\#a_{h+1}, \ldots, q_k\#a_k,\ L_1, \ldots, L_n.$$

for some $0 \leqslant h \leqslant k$, $n \geqslant 0$, and $(k-h)+n > 0$, with $L_1, \ldots, L_n$ p-literals. For each a-atom $q_i\#a_i$ in γ we introduce the fact:

$$a\_atom(r_\gamma, i, q_i, \hat{a_i}). \tag{11}$$

where $\hat{a_i} = a_i$ if $0 \leqslant i \leqslant h$ and $\hat{a_i} = -a_i$ if $h < i \leqslant k$. These facts represent, in the ASP translation, the a-atoms occurring in $S_\mathcal{R}$. The second argument of *a_atom* is needed in the ASP translation to distinguish among different occurrences of identical a-atoms of the r-rule because all multiple copies of the same a-atom must be considered, since they correspond to distinct amounts of resource.[9]

As mentioned, the two disjoints of the formula (9) in Def. 8 discriminate between the two situations where an r-rule γ is fired or not. The number $\xi(\gamma)$ of firings of γ has to be taken into account. The following fragment of ASP code "declares" a counter used to denote such number. Its value is restrained, considering the admitted values *Idx*, by introducing a fact $firings(r_\gamma, i).$ for each integer i in *Idx*.[10]

$$\begin{array}{l} counter(Rule, C) \leftarrow firings(Rule, C),\ fired(Rule),\ not\ \ notcounter(Rule, C). \\ notcounter(Rule, C) \leftarrow counter(Rule, D),\ C \neq D,\ r\_rule(Rule). \\ \leftarrow not\ fired(Rule),\ counter(Rule, C). \end{array} \tag{12}$$

In this manner, the predicate *counter* encodes the mapping ξ introduced as part of the r-interpretation, cf., Def. 7. The third rule in (12), that is actually an ASP constraint, imposes that no counter selection is done for the rules that are not fired. The situations in which an r-rule is fired or not are modeled through the rules:

$$\begin{array}{l} fired(Rule) \leftarrow not\ \ notfired(Rule),\ r\_rule(Rule). \\ notfired(Rule) \leftarrow not\ \ fired(Rule),\ r\_rule(Rule). \end{array} \tag{13}$$

For each a-atom *Res*#*Amount* in an r-rule *Rule*, we represent the fact that $Count > 0$ firings of *Rule* use $Count * Amount$ of a resource *Res*, through the ASP rules:

$$\begin{array}{l} use(Rule, I, Res, Count * Amount) \leftarrow fired(Rule),\ counter(Rule, Count), \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad a\_atom(Rule, I, Res, Amount). \\ fired(Rule) \leftarrow use(Rule, I, Res, CA). \\ notfired(Rule) \leftarrow not\ \ use(Rule, I, Res, CA). \end{array} \tag{14}$$

Finally, we impose that the firing of γ has to be enabled by the truth of the literals $L_1, \ldots, L_n$, through the ASP rules:

$$\leftarrow \overline{L_i},\ fired(r_\gamma). \qquad\qquad \text{for each } i \in \{1, \ldots, n\} \tag{15}$$

The translation $\mathcal{T}(\gamma)$ of γ consists of the rules (10)–(15). Such a translation can be slightly modified to treat r-rules having a p-atom as head (see [10] for the details). r-facts are treated as r-rules that are supposed to be always fired once. Hence, if γ is the r-fact $q\#$ then $\mathcal{T}(\gamma)$ is as follows:

$$r\_rule(r_\gamma). \qquad a\_atom(r_\gamma, 0, q, a). \qquad fired(r_\gamma). \qquad use(r_\gamma, 0, q, a). \qquad counter(r_\gamma, 1). \tag{16}$$

By defining $\mathcal{T}(\gamma) = \{\gamma\}$ for all p-rules, the translation of an r-program S is defined as the ASP program $\mathcal{T}(S) = \bigcup_{\gamma \in S} \mathcal{T}(\gamma)$.

Let M be a model of the program $\mathcal{T}(S)$. For some r-symbols q, some of the atoms $use(r_\gamma, i, q, a)$, occurring in $\mathcal{T}(S)$, are true in M. These atoms are intended to represent the amounts of resources involved in fired r-rules. To take into account the constraints on global balance of the allocated amounts, we introduce a condition $pos(M)$ so defined:

$$pos(M) \quad = \quad \forall q \in \tau_\mathcal{R}\ \big(\textstyle\sum \{\!\!\{ a \mid use(r_\gamma, i, q, a) \in M \}\!\!\} \geqslant 0\big) \tag{17}$$

[9] Such iterated copies of the same a-atom may result from the grounding process.

[10] Clearly, in encoding an r-program it suffices to introduce the fragments (12) once, and similarly for (13), (14), and (18)–(19) to be seen.

This condition reflects the positivity constraint imposed in the definition (3) of $\mathbb{S}_P$. We encode this condition by the following ASP fragment, which involves the aggregate function *sum*:

$$res\_symb(Res) \leftarrow a\_atom(Rule, I, Res, Amount). \tag{18}$$

$$\leftarrow sum\{A : use(Rule, I, Q, A)\} < 0,\ res\_symb(Q). \tag{19}$$

A detailed description of aggregate functions (such as *sum*, *min*, *max*) and *aggregate literals* can be found, for instance, in [18, 26, 13]. A simple form of aggregate literal, sufficient for our purposes, is the following: $n_1\ op_1\ Agg\{\,Var : Conj\}\ op_2\ n_2$, where *Var* is a variable, *Conj* is a conjunction of literals, each op_i is a relational operator (e.g., $<$,$\leqslant$,$\geqslant$, etc.), n_1 and n_2 are terms (called *guards*, usually numbers), and *Agg* is an aggregate function. (One among the guards might be absent.) The intended meaning is as follows: given an interpretation, the aggregate function *Agg* is applied to the multiset of all the values for *Var* that satisfy the conjunction of literals *Conj*. Variables in *Conj*, different from *Var*, are considered as being existentially quantified. The result of the function is then compared, through op_i, with n_1 and n_2 to determine the truth value of the aggregate literal. Considering (19), the variables *Rule* and I are existentially quantified, whereas the value of Q is determined by the atom $res\_symb(Q)$. Hence, for each q such that $res\_symb(q)$ holds, the aggregate function *sum* is applied to the multiset of those values A occurring in the facts $use(Rule, I, q, A)$ that are true in the interpretation at hand.

Observe that we are introducing an aggregate literal in a constraint. Hence, no literal in other rules of $\mathcal{T}(S)$ is defined depending on such aggregate. This ensures that the resulting program is aggregate stratified. Stable model semantics can be smoothly extended to such class of programs by generalizing the notion of reduct of a program.

4.2 Treatment of plain p-lists

We describe here an ASP rendering of p-lists that do not involve compound resources. In doing this, we focus on the treatment of a single p-list in the body of a ground r-rule γ:

$$Idx: \quad H \leftarrow q_1\#a_1 > \cdots > q_k\#a_k,\ B_1, \ldots, B_m. \tag{20}$$

where $B_1, \ldots, B_m$ ($m \geqslant 0$) are r-literals, $k > 1$, and, as before, *Idx* is $[N_{1,1}\text{-}N_{1,2}, \ldots, N_{h,1}\text{-}N_{h,2}]$. The case of a p-list in the head is similar and in case of multiple p-lists the translation proceeds by processing each p-list one at a time. From (20) the following two r-rules are obtained:

$$Idx: \quad H \leftarrow pl\#1,\ B_1, \ldots, B_m. \tag{21}$$

$$pl\#N_{h,2}. \tag{22}$$

where pl is a new auxiliary r-symbol whose availability, in quantity equal to the maximum number of firings admitted for rule (20), is stated by (22). Both (21) and (22) are further translated into ASP as explained earlier. The following fragment of ASP program establishes a dependence between (unitary amounts of) resource pl, each single firing of rule (21), and the a-atoms in $q_1\#a_1 > \cdots > q_k\#a_k$ (here, we are assuming $\{auxres,\ res\_pl,\ use\_pl,\ sum\_use\_pl\} \subseteq \Pi_{\mathcal{T}}$):

$$\begin{array}{l} auxres(r_\gamma, pl). \\ res\_pl(r_\gamma, pl, q_1, 1). \quad \cdots \quad res\_pl(r_\gamma, pl, q_k, k). \\ 1\{use\_pl(r_\gamma, q_1, I, a_1, pl), \ldots, use\_pl(r_\gamma, q_k, I, a_k, pl)\}1 \leftarrow counter(r_\gamma, NumFirings), \\ \qquad\qquad\qquad\qquad use(r_\gamma, 1, pl, -1{*}NumFirings), I \leqslant NumFirings, iter(I). \end{array} \tag{23}$$

The facts in the first two lines associate each q_i with the resource pl. The rule in the third line implements a correspondence between firings of (21) (i.e., uses of pl) and firings of γ (i.e., uses of q_i). Being *NumFirings* the number of times (21) is fired, this rule imposes that each time one instance of pl is used (namely, once for each I, $1 \leqslant I \leqslant NumFirings$, through the predicate $iter(I)$ that acts as *domain predicate* restraining the admitted values for I), exactly one instance of one resource among the q_is is used (i.e., one of the facts

$use\_pl(r_\gamma, q_i, I, a_i, pl)$ is true). Notice the use of a cardinality constraint to force the truth of exactly one atom use_pl.

In general, a cardinality constraint has the form $k\{L_1, \ldots, L_n\}m$. It is satisfied in an interpretation M, if M satisfies $k \leq h \leq m$ literals among $L_1, \ldots, L_n$. The use of such a constraint could be avoided, but it allows a more succinct encoding. See, for instance, [3] for details on cardinality constraints and on how to surrogate them through usual ASP rules.

Finally, the encoding developed so far is extended with the following ASP fragment that reflects consumption of (each unit of) auxiliary resource into consumption of real resources.

$$\begin{array}{l} res\_symb(Pl) \leftarrow auxres(Rule, Pl), r\_rule(Rule). \\ use(Rule, Idx, R, U{*}N) \leftarrow sum\_use\_pl(Rule, R, N, Pl), res\_pl(Rule, Pl, R, Grade), \\ \qquad\qquad\qquad\qquad r\_rule(Rule), a\_atom(Rule, Idx, Pl, U), N \neq 0. \\ sum\_use\_pl(Rule, R, N, Pl) \leftarrow res\_pl(Rule, Pl, R, Grade), \\ \qquad\qquad\qquad\qquad N = sum\{Q : use\_pl(Rule, R, Iter, Q, Pl), firings(Rule, Iter)\}, \\ \qquad\qquad\qquad\qquad r\_rule(Rule), auxres(Rule, Pl), iter(Iter). \end{array} \tag{24}$$

These rules evaluate the balance of each resource occurring in the initial p-list, by means of an aggregate literal that sums up the amounts of real resources (i.e., the q_is) corresponding to units of the auxiliary resource (i.e., the pl). In particular, notice that, the first rule allows us to exploit the same encoding described earlier, to evaluate consumptions of auxiliary resources.

4.3 Completeness, soundness, and complexity issues for RASP with preferences

Some notions (adapted from [21]) are needed. Consider two sets of atoms Z and X and an ASP program G. For a rule $\gamma \in G$ let γ' denote the rule obtained by removing those atoms belonging to Z from the body of γ. More precisely, γ' is such that $head(\gamma') = head(\gamma)$, $lits^+(\gamma') = lits^+(\gamma) \backslash Z$, and $lits^-(\gamma') = lits^-(\gamma) \backslash Z$. Then, given G, the program $e_Z(G, X)$ is defined as the following set of rules $e_Z(G, X) = \{\gamma' \mid \gamma \in G,\ lits^+(\gamma) \cap Z \subseteq X,\ (lits^-(\gamma) \cap Z) \cap X = \emptyset\}$.

A *splitting set* Z for an ASP program G is a set of atoms such that for each rule $\gamma \in G$, if $head(\gamma) \cap Z \neq \emptyset$ then $lits^+(\gamma) \cup lits^-(\gamma) \subseteq Z$. The set of rules $b_Z(G) = \{\gamma \in G \mid atoms(\gamma) \subseteq Z\}$ is called the *bottom* of G w.r.t. Z.

Let Z be a splitting set for a program G. A *solution* to G w.r.t. Z is a pair $\langle X, Y \rangle$ of sets of atoms such that X is an answer set for $b_Z(G)$ and Y is an answer set for $e_Z(G \setminus b_Z(G), X)$.

Let $S = S_\mathcal{P} \cup S_\mathcal{R}$ be a ground r-program. Observe that $atoms(S_\mathcal{P}) \subseteq \mathcal{B}(\Pi_\mathcal{P}, \mathcal{C})$.[11]

Consider the ground ASP program $\mathcal{T}(S)$ so defined:

$$\mathcal{T}(S) = S_\mathcal{P} \cup \check{S} \cup S_\mathcal{T}$$

where $S_\mathcal{T}$ is the set of ground instances of all rules of the forms (10), (11), (12), (13), (14), (16), (18), (23), and (24), while $\check{S}$ is the set of all ground instances of rules (15) originating from the translation. (Notice that, for the time being, we are excluding the rules (19) from $\mathcal{T}(S)$. We will recover them later, by imposing the condition (17).)

To simplify the treatment, without loss of generality, we make some assumptions on S (and $\mathcal{T}(S)$). In particular, in place of each of the ground instances in $\mathcal{T}(S)$ of constraints (15), we consider the equivalent instance of the form

$$aux_{L_i,\gamma} \leftarrow not\ \ aux_{L_i,\gamma}, \overline{L_i}, fired(r_\gamma).$$

where $aux_{L_i,\gamma}$ is a fresh auxiliary atom. Let Π_{aux} be the set of all such predicate symbols $aux_{L_i,\gamma}$.

We perform an analogous replacement for all ASP constraint in $S_\mathcal{P}$. Hence, we proceed by considering all rules in $S_\mathcal{P}$ having the form $H \leftarrow L_1, \ldots, L_m, not\ \ L_1, \ldots, not\ \ L_n$.

Moreover, we assume all "built-in" predicates (such as $<, \leqslant, =, \neq, \ldots$) and functions (such as $+, *, -, \ldots$) used in the encoding, as defined by suitable ASP fragments, included in $S_\mathcal{P}$. (Notice that the number of

[11] Recall that $\mathcal{B}(X, Y)$ denotes the collection of all ground atoms built up from predicate symbols in X and terms in Y.

constants occurring in a program is finite, hence such fragments of ASP code might consist of simple lists of ground facts.)

Finally, we surrogate each aggregate literal occurring in (instances of) the third rule of (24), by a fragment of ASP code, to evaluate the sum (N, in (24)) of the amounts of each resource (R, in (24)).[12] Similarly, we replace each cardinality constraint (occurring in instances of (23)) by their standard equivalent encodings in pure ASP rules (cf., [3], for instance).

Consequently, we simplify $\mathcal{T}(S)$ so that it does not contain any aggregate or cardinality literal, or built-in predicate/relator. Clearly, these simplifications of $\mathcal{T}(S)$ do not affect its answer sets.

We can now plainly proceed by adapting the proof of completeness and soundness of basic RASP provided in [10, Sect. 5.2], to the more complex case in which both plain p-lists and iterated firings of rules are allowed. The proof relies on the *Splitting Set Theorem* [21] to show that each answer set of $\mathcal{T}(S)$ can be split in two parts, one of which being the component I of an answer set $\mathcal{I} = \langle I, \mu, \xi\rangle$ for S. The other part encodes the two mappings μ and ξ.

Let us start by observing that $atoms(\check{S}) \subseteq \mathcal{B}(\Pi_{\mathcal{P}}, \mathcal{C}) \cup \mathcal{B}(\{fired\} \cup \Pi_{aux}, \{r_\gamma \mid \gamma \in S\})$ and $atoms(S_{\mathcal{T}}) \subseteq \mathcal{B}(\Pi_{\mathcal{T}}, \{r_\gamma \mid \gamma \in S\} \cup \mathbb{Z} \cup \tau_{\mathcal{R}})$. The set $U_S = \mathcal{B}(\Pi_{\mathcal{T}}, \{r_\gamma \mid \gamma \in S\} \cup \mathbb{Z} \cup \tau_{\mathcal{R}})$ is a *splitting set* for $\mathcal{T}(S)$ and the *bottom* of $\mathcal{T}(S)$ relative to U_S is

$$b_{U_S}(\mathcal{T}(S)) = \{\gamma \in \mathcal{T}(S) \mid atoms(\gamma) \subseteq U_S\} = S_{\mathcal{T}}.$$

This is so because no atom in U_S occurs as head in rules of $S_{\mathcal{P}} \cup \check{S}$. At this point, we can establish a relation between the answer sets of an r-program and those of its translation.

Let M be an answer set for $\mathcal{T}(S)$. By the *Splitting Set Theorem* [21], $M = X \cup Y$ for X and Y such that $\langle X, Y\rangle$ is a solution to $\mathcal{T}(S)$ with respect to U_S. Hence, by the definition of a solution, we have that X is an answer set for $b_{U_S}(\mathcal{T}(S)) = S_{\mathcal{T}}$, and Y is an answer set for $e_{U_S}(S_{\mathcal{P}} \cup \check{S}, X)$. We have that $Y \subseteq atoms(\mathcal{T}(S)) \setminus U_S \subseteq \mathcal{B}(\Pi_{\mathcal{P}}, \mathcal{C}) \cup \mathcal{B}(\{fired\} \cup \Pi_{aux}, \{r_\gamma \mid \gamma \in S\})$ and that $X \subseteq atoms(\mathcal{T}(S)) \cap U_S$ encodes the information about which r-rules are fired in M.

Consider the r-interpretation $\mathcal{I}_M = \langle Y, \mu_M, \xi_M\rangle$ where $\xi_M(\gamma) = n$ for each $\gamma \in S$ such that $counter(r_\gamma, n)$ is in M (notice that, by (14), for each γ which is fired, exactly one fact $counter(r_\gamma, n)$ is in M); The mapping μ_M is such that for all $q \in \tau_{\mathcal{R}}$ and $\gamma \in S$, $\mu_M(q)(\gamma)$ is the multiset containing

- $\xi(\gamma)$ copies of $\langle idx, v\rangle$ for each occurrence of $q\#v$ in γ as a plain a-atom (i.e., not in a p-list) and $use(r_\gamma, idx, q, v * \xi(\gamma))$ in M.
- k copies of $\langle idx, v\rangle$ for each occurrence of $q\#v$ as j-th element of a p-list of γ and the ground atoms $use(r_\gamma, idx, q, v * k)$, $use\_pl(r_\gamma, q, |v|, pl)$, and $a\_atom(r_\gamma, idx, pl, u)$ are in M.

The following result holds (where $pos(M)$ is the condition defined by (17)).

Theorem 1 *If M is an answer set for $\mathcal{T}(S)$ such that $pos(M)$ holds, then $\mathcal{I}_M$ is an answer set the r-program S.*

Proof. We have to show that $\mathcal{I}_M = \langle Y, \mu_M, \xi_M\rangle$ fulfills the requirements expressed in the two items in Def. 8. Recall that, by the Splitting Set Theorem, $M = X \cup Y$, with $\langle X, Y\rangle$ solution to $\mathcal{T}(S)$ with respect to U_S.

- As a preliminary step we show that rules (23) and (24) establish an one-to-one correspondence between the amounts of real resources in a p-list and those of the auxiliary resource pl.

  Consider the instances of (23) and (24) in $\mathcal{T}(S)$ that encode a p-list $q_1\#a_1 > \cdots > q_m\#a_m$ in the body of a rule γ. (Observe that none of the facts of the form $use(\cdot, \cdot, pl, \cdot)$ depends (neither transitively) on facts of the form $use(\cdot, \cdot, q_i, \cdot)$ (for any i).) In every answer set M of $\mathcal{T}(S)$ such that γ is fired n times, by (13), n units of pl are consumed. Then, because of (23), n facts of the form $use\_pl(r_\gamma, q_i, h, a_i, pl)$ are in M. Therefore, by (24), facts $use(r_\gamma, idx, q_i, b_i)$ in M model the use of the real resources q_i. Conversely, each fact $use(r_\gamma, idx, q_i, -b_i)$ in M must be supported through (24) by the presence of a set of atoms $sum\_use\_pl(r_\gamma, q_i, a_i, pl)$ and $use\_pl(r_\gamma, q_i, h, a_i, pl)$ in M, so that, for each i, $b_i = n * a_i$,

[12] Actually, given a set of ground facts of the form $p(x, y, z)$ in an answer set, it is a programming exercise to write a set of ASP rules that evaluates the sum of all xs. In $\mathcal{T}$ we exploited aggregates to enable a more concise and readable encoding.

being n the number of times γ is fired in M. P-lists occurring in the head of γ are handled in the very same way.

Consider a rule $\gamma \in S$. Let $\ell_1, \ldots, \ell_k$ be all the a-literals in γ, each of them being either an a-atom or a p-list. Let, moreover, *Idx* be the set of admissible number of iterations in firing γ, as specified in its definition. Two cases are possible: γ is fired or not; i.e., $\mathit{fired}(r_\gamma)$ belongs to M or not. If $\mathit{fired}(r_\gamma) \in M$, because M models the instances of (12) in $\mathcal{T}(S)$, exactly one fact of the form $\mathit{counter}(r_\gamma, n)$ is in M, for an admissible integer value $n \in \mathit{Idx}$. Let us first consider the case in which each ℓ_i is an a-atom, say $q_i \# a_i$. By the definition of $\mathcal{T}$, facts of the form $\mathit{a\_atom}(r_\gamma, i, q_i, a_i)$ belong to X. Then, because of (14), facts of the form $\mathit{use}(r_\gamma, i, q_i, b_i)$, for $b_i = n * a_i$, are in X too and n copies of $\langle i, a_i \rangle$ are in $\mu(q_i)(\gamma)$, for each $i = 1, \ldots, k$. Hence, the second disjoint of (9) is satisfied by the definition of *ch* and $\mathcal{R}^n$, and because $\mathit{pos}(M)$ holds by hypothesis. (Recall that the constraint specifying, in (9), the admissible values in *Idx* for $\xi(\gamma)$ are rendered, by introducing facts of the form $\mathit{firings}(r_\gamma, i)$ in the translation (cf., Section 4, page 13).)

If it is the case that some ℓ_i is a p-list, then suitable instances of the rules (23) are in $\mathcal{T}(S)$. An auxiliary resource symbol pl_j is introduced for each j such that ℓ_j is a p-list, say $q_1^j \# a_1^j > \cdots > q_{m_j}^j \# a_{m_j}^j$. Since X models these instances of (23), for each $h \in \{1, \ldots, n\}$, one among the atoms $\mathit{use\_pl}(r_\gamma, q_1^j, h, a_1^j, pl_j), \ldots, \mathit{use\_pl}(r_\gamma, q_{m_j}^j, h, a_{m_j}^j, pl_j)$ belongs to X (for each j). Each of these atoms corresponds to one of the alternative choices represented by the set $\mathit{ch}(q_1^j \# a_1^j > \cdots > q_{m_j}^j \# a_{m_j}^j, \emptyset)$, as defined in (5) and facts of the form $\mathit{use}(r_\gamma, \mathit{idx}, q_s^j, k * a_s^j)$ are in M. Consequently, for each q the multiset $\mu(q)(\gamma)$ is a member of $\mathcal{R}^n(q, \gamma, Y)$. This ensures that the second disjunct of (9) is satisfied.

If γ is not fired, since M is a model of all instances of (12) in $\mathcal{T}(S)$, no atom of the forms $\mathit{fired}(r_\gamma)$ or $\mathit{counter}(r_\gamma, n)$ is in X. Hence, by (14), for each i there is no atom of the form $\mathit{use}(n_\gamma, i, q_i, a_i)$ in X and $\mu_M(q)(\gamma) = \emptyset$ for all $\in \tau_\mathcal{R}$. Condition (9) of Def. 8 is satisfied.

- Recall that Y is an answer set for $e_{U_S}(S_\mathcal{P} \cup \check{S}, X)$. Observe that the predicate symbols occurring in X do not occur in $S_\mathcal{P}$. Then $e_{U_S}(S_\mathcal{P} \cup \check{S}, X) = S_\mathcal{P} \cup e_{U_S}(\check{S}, X)$. Moreover, an atom of the form $\mathit{fired}(r_\gamma)$ belongs to X if and only if the r-rule γ produces/consumes the resources described by its a-literals (i.e., if and only if γ is fired, and by effect of rules (14), for each firing, all needed atoms $\mathit{use}(n_\gamma, i, q, v)$ belong to M). On the other hand, U_S contains all atoms of the form $\mathit{fired}(r_\gamma)$. Let ρ be a rule $\check{S}$. By definition, ρ has the form (15) and originates from the translation of an r-rule γ. If such γ is fired, then $\mathit{fired}(r_\gamma)$ belongs to X and occurs in ρ. Let ρ' be obtained by removing the atom $\mathit{fired}(r_\gamma)$ from ρ. The rule ρ' is in $e_{U_S}(\check{S}, X)$ only if $\mathit{lits}^+(\rho) \cap U_S = \{\mathit{fired}(r_\gamma)\} \subseteq X$. This shows (because of the way μ_M has been defined, see above), that $S_\mathcal{P} \cup e_{U_S}(\check{S}, X) = \hat{S}$ and the requirement specified by the second item of Def. 8 is satisfied.

□

Thm. 1 states the soundness of the ASP embedding of r-programs. We now proceed by showing its completeness. Let $\mathcal{I} = \langle I, \mu, \xi \rangle$ be an answer set of an r-program S. (Recall that, given $\mathcal{I}$, for each $\gamma \in S_\mathcal{R}$ one and only one of the disjuncts in condition (9) of Def. 8 is satisfied.) We introduce these sets of atoms (where Idx_γ denoted the set of admitted firings relatively to the rule γ):

$$
\begin{array}{lcl}
S_{(10)} & = & \{\mathit{r\_rule}(r_\gamma) \mid \gamma \text{ r-rule or r-fact in } S_\mathcal{R}\} \\
S_{(11)} & = & \{\mathit{a\_atom}(r_\gamma, i, q_i, \hat{a}_i) \mid q_i \# a_i \text{ is an a-atom in } \gamma \in S_\mathcal{R}\} \\
S_{(12)} & = & \{\mathit{counter}(r_\gamma, \xi(\gamma)) \mid \gamma \in S_\mathcal{R} \text{ and } \xi(\gamma) > 0\} \cup \{\mathit{firings}(r_\gamma, i) \mid \gamma \in S_\mathcal{R}, i \in \mathit{Idx}_\gamma\} \cup \\
 & & \{\mathit{notcounter}(r_\gamma, i) \mid \gamma \in S_\mathcal{R}, i \in \mathit{Idx}_\gamma, i \neq \xi(\gamma)\} \cup \{\mathit{iter}(i) \mid i \in S_\mathcal{R}, \in \{1, \ldots, \xi(\gamma)\}\} \\
S_{(13)} & = & \{\mathit{fired}(r_\gamma) \mid \gamma \in S_\mathcal{R} \text{ and } \gamma \text{ is fired in } \mathcal{I}\} \cup \{\mathit{notfired}(r_\gamma) \mid \gamma \in S_\mathcal{R} \text{ and } \gamma \text{ is not fired in } \mathcal{I}\} \\
S_{(14)} & = & \{\mathit{use}(r_\gamma, i, q_i, v) \mid q_i \# a_i \text{ is an a-atom in } \gamma \in S_\mathcal{R}, v = \xi(\gamma) * a_i, \text{ and } \gamma \text{ is fired in } \mathcal{I}\} \\
S_{(16)} & = & \{\mathit{fired}(r_\gamma) \mid \gamma \text{ r-fact in } S_\mathcal{R}\} \cup \{\mathit{a\_atom}(r_\gamma, 0, q, \hat{a}) \mid q \# a \text{ r-fact in } S_\mathcal{R}\} \cup \\
 & & \{\mathit{use}(r_\gamma, 0, q, \hat{a}) \mid q \# a \text{ r-fact in } S_\mathcal{R}\} \cup \{\mathit{counter}(r_\gamma, 1) \mid q \# a \text{ r-fact in } S_\mathcal{R}\} \\
S_{(18)} & = & \{\mathit{res\_symb}(q) \mid \ q \text{ r-symbol in } S_\mathcal{R}\}
\end{array}
$$

Moreover, for each fired r-rule γ and for each p-list $r = q_1^j \#a_1^j > \cdots > q_{m_j}^j \#a_{m_j}^j$ occurring as j-th a-literal in γ, let the set $S_{(24)}$ contain the following atoms:

- $r\_rule(r_{\gamma_j})$, $fired(r_{\gamma_j})$, $counter(r_{\gamma_j}, 1)$, $a\_atom(r_{\gamma_j}, 0, pl_j, n_\gamma)$, and $use(r_{\gamma_j}, 0, pl_j, n_\gamma)$, where pl_j is a fresh r-symbol, r_{γ_j} is a fresh constant symbol;
- $a\_atom(r_\gamma, 1, pl_j, 1)$, $res\_symb(pl_j)$, $auxres(r_\gamma, pl_j)$, and $use(r_\gamma, 1, pl_j, -\xi(\gamma))$, where n_γ is the maximum number of admitted firings for γ (i.e., the maximum integer in Idx_γ);
- the atoms $res\_pl(r_\gamma, pl_j, q_1^j, 1), \ldots, res\_pl(r_\gamma, pl_j, q_{m_j}^j, m_j)$;
- $\xi(\gamma)$ atoms of the form $use\_pl(r_\gamma, q_s^j, idx_h, a_s^j, pl_j)$ for $q_s^j \in \{q_1^j, \ldots, q_{m_j}^j\}$ and such that either $\langle j, a_s^j \rangle$ or $\langle 0, a_s^j \rangle$ is an amount couple in $\mu(q_s^j)(\gamma)$ not associated to a plain a-atom of γ, and arbitrarily choosing the $\xi(\gamma)$ distinct indices idx_hs so that $\{idx_1, \ldots, idx_{\xi(\gamma)}\} = \{1, 2, \ldots, \xi(\gamma)\}$;
- an atom $sum\_use\_pl(r_\gamma, q_s^j, v, pl_j)$, for each s such that facts of the form $use\_pl(r_\gamma, q_s^j, idx_h, a_s^j, pl_j)$ are introduced by the preceding item, so that v is the sum of the amounts a_s^j occurring in such facts.
- an atom $use(r_\gamma, j, q_s^j, x)$ for each fact $sum\_use\_pl(r_\gamma, q_s^j, v, pl_j)$ introduced by the preceding item and either $x = v$ or $x = -v$, depending on the fact that the p-list at hand occurs either in the head or in the body of γ.

Finally, let $X_S = S_{(10)} \cup S_{(11)} \cup S_{(12)} \cup S_{(13)} \cup S_{(14)} \cup S_{(16)} \cup S_{(18)} \cup S_{(24)}$. The following result states the completeness of the transformation.

Theorem 2 *Let $\mathcal{I} = \langle I, \mu, \xi \rangle$ be an answer set of an r-program S and X_S defined as described above. Then, $M = X_S \cup I$ is an answer set for $\mathcal{T}(S)$ and $pos(M)$ holds.*

Proof. Observe that I is an answer set for $\hat{S}$ and that X_S is an answer set for $b_{U_S}(\mathcal{T}(S))$. By an argument similar to the one applied in the proof of Thm. 1, we obtain that $e_{U_S}(\mathcal{T}(S) \setminus b_{U_S}(\mathcal{T}(S)), X_S) = e_{U_S}(S_\mathcal{P} \cup \check{S}, X_S) = \check{S}$. Hence, $\langle X_S, I \rangle$ is a solution of $\mathcal{T}(S)$ w.r.t. U_S. It follows, by the Splitting Set Theorem, that $X_S \cup I$ is an answer set for $\mathcal{T}(S)$. We can conclude the proof by observing that, by the definition of $\mathbb{S}_P$ (cf., (3)), the Def. 8, and by the facts of the form $use(r_\gamma, j, q, v)$ included, by definition, in $S_{(14)}$, $S_{(16)}$ and $S_{(24)}$, $pos(M)$ holds. □

Theorems 1 and 2 show that the translation $\mathcal{T}$ induces a correspondence between the answer sets of a (ground) r-program and those of its ASP encoding.

Computational complexity has been assessed for basic RASP (cf., [10, Sect. 6]). In particular, the NP-completeness of deciding whether an r-program has an answer set is obtained by verifying that the translation $\mathcal{T}$ is polynomial. An analogous result holds when p-lists are taken into account:

Proposition 1 *The problem of deciding whether a ground r-program involving p-lists has an answer set is NP-complete.*

Proof. The result is immediately obtained by observing that the encoding of a p-list through instances of (23) and (24) can be applied to treat each p-list in a program, independently from the treatment of the other p-lists. The resulting encoding involves the introduction of a number of ASP rules and literals which is polynomially bounded w.r.t. the length of the given r-program. We can conclude by the NP-completeness of basic RASP. □

Hence, adding plain p-lists to RASP does not affect its complexity. A similar result was stated in [9] by relating r-programs to LPOD [6]. Theorem 1 avoids referring to LPOD and establishes a direct connection to ASP.

As regards the complexity of the problem of determining if a given literal is true in a most preferred answer set, different results are obtained depending on the specific preference criterion which is adopted for ranking the answer sets. The complexity of credulous reasoning for RASP with preferences is in line with that of LPOD. For instance, if a *Pareto*-like criterion is adopted, Σ_P^2-completeness of both RASP and LPOD is obtained (see [6, 9]).

4.4 Compound resources and preferences on compound resources

Let us first consider the completeness and soundness of the encodings of compound resources not occurring in p-lists. The case of c-resources is immediate. In fact, a c-resource not occurring in a p-list can be seen

as syntactic sugar for the list of its a-atoms. The results follow from completeness and soundness of basic RASP.

As regards d-resources not occurring in p-lists, observe that an occurrence of a d-resource $\{q_1\#a_1; \cdots ; q_k\#a_k\}$ can be replaced by the cp-list $(q_1\#a_1 > \cdots > q_k\#a_k \;\; pref\_when \;\; p, not \;\; p)$, where p is any p-atom. Hence, the results for d-resources can be obtained as consequences of those for cp-lists, Propositions 4 and 5, to be seen.

Let us now describe how p-lists that involve compound resources can be polynomially encoded into ASP. Again, let us focus on the case of a single p-list occurring in the body of a ground r-rule γ of P:

$$Idx: \quad H \leftarrow s_1 > \cdots > s_k,\, B_1, \ldots, B_m.$$

where, for each $i \in \{1, \ldots, k\}$, $s_i = \{q_{i,1}\#a_{i,1}, \ldots, q_{i,k_i}\#a_{i,k_i}\}$ and Idx is $[N_{1,1}\text{-}N_{1,2}, \ldots, N_{h,1}\text{-}N_{h,2}]$. (The cases of p-lists occurring in the head and/or involving disjunctive compound resources can be treated similarly.) For each $i \in \{1, \ldots, k\}$ let aux_i be a fresh r-symbol not occurring elsewhere in the program. Then, we introduce the following r-rules:

$$\begin{array}{lll}
(\gamma') & Idx: & H \leftarrow B_1, \ldots, B_m,\, pl\#1\,, z\#\text{-}1. \\
(\gamma'') & & pl\#N_{h,2}. \\
(\gamma_i) & [1\text{-}N_{h,2}]: & aux_i\#1 \leftarrow q_{i,1}\#a_{i,1}, \ldots, q_{i,k_i}\#a_{i,k_i}, z\#1. \qquad \text{for each } i \in \{1, \ldots, k\}
\end{array}$$

where pl and z are fresh r-symbols (note that an a-atom with negative amount in the body of an r-rule, actually denotes resource production). The rationale is as follows. The auxiliary resource z acts as a counter of the number of firings of γ': each time γ' is fired, an instance of z is produced and one instance of pl is consumed. Conversely, each resource aux_i is produced only if γ_i is fired and this can happen only by consuming one instance of z. Hence, each firing of γ' corresponds to one firing of one of the γ_is. The r-fact γ'' ensures that γ' cannot be fired more than $N_{h,2}$ times.

Notice that γ'', γ'', and all of the γ_i do not involve preferences. Hence, their ASP encoding can be obtained through the translation described earlier. To complete the translation we impose a direct dependency between the uses of the resource pl and the resources $aux_1, \ldots, aux_k$. This is done by means of this fragment of ASP program:

$$\begin{array}{l}
auxres(r_{\gamma'}, pl). \\
res_pl(r_{\gamma'}, pl, aux_1, 1). \quad \cdots \quad res_pl(r_{\gamma'}, pl, aux_k, k). \\
1\{use_pl(r_{\gamma'}, aux_1, I, 1, pl), \ldots, use_pl(r_{\gamma'}, aux_k, I, 1, pl)\}1 \;\leftarrow\; counter(r_{\gamma'}, NumFirings), \\
\qquad\qquad use(r_{\gamma'}, 0, pl, -1{*}NumFirings), I \leqslant NumFirings, iter(I).
\end{array} \tag{25}$$

Notice that these rules surrogate the ASP encoding of an r-rule that involves a p-list of the form $aux_1\#1 > \cdots > aux_k\#1$ as explained in page 14 (consider an encoding obtained from (23) by replacing each q_i by aux_i and by instantiating each a_i to 1).

Let us refine the translation $\mathcal{T}$ so that all p-lists involving compound resources are translated as described above (subsequently, the resulting r-program is translated as explained in the previous subsection). We have the following result:

Proposition 2 *Let P be a ground r-program involving p-lists and compound resources and let P' be the grounding of the program $\mathcal{T}(P)$. Then, each answer set of P determines an answer set for P', and vice versa.*

Proof. The proof develops in strict analogy to what done for Theorems 1 and 2. In fact, the treatment of compound resources in p-lists relies on the previously described treatment of simple p-lists (observe that the fragment of ASP program (25) is a variant (23)). In particular, notice that the completeness/soundness of the encoding for basic RASP ensures that each use of a unit of pl corresponds to a use of one unit of z and, in turn, to one use of aux_i, for some i. □

The next proposition states that the complexity of RASP is not affected by the presence of preferences on compound resources.

Proposition 3 *The problem of deciding whether a ground r-program involving preferences on compound resources has an answer set is NP-complete.*

PROOF. The above-described translation can be applied to treat each p-list involving compound resources in a program, independently from the treatment of the other p-lists. The resulting encoding introduces a number of ASP rules and literals which is polynomially bounded w.r.t. the length of the given r-program. We conclude, by Proposition 1. □

4.5 Conditional p-lists

As before, let us focus on the case of a single cp-list occurring in the body of a ground r-rule ρ of P of the form:

$$Idx: \quad H \leftarrow B_1, \ldots, B_m, (s_1 > \cdots > s_k \;\; pref\_when \;\; L_1, \ldots, L_n)$$

where each s_i and Idx are as before. The rule ρ can be replaced by these r-rules (where p and np are fresh p-atoms, and pl is a fresh r-symbol):

(ρ')	$Idx:$	$H \leftarrow B_1, \ldots, B_m,\ pl\#1\,, z\#{-}1.$	
(ρ'')	$Idx:$	$pl\#1 \leftarrow p,\ s_1 > \cdots > s_k,\ z\#1$	
(ρ_i)	$[1\text{-}N_{h,2}]:$	$pl\#1 \leftarrow np,\ s_i,\ z\#1.$	for each $i \in \{1, \ldots, k\}$
		$p \leftarrow not\ \ np. \qquad np \leftarrow not\ \ p.$	
		$\leftarrow np, L_1, \ldots, L_n.$	
		$\leftarrow p, \overline{L_j}.$	for each $j \in \{1, \ldots, n\}$

Atoms p and np characterize the situations in which $L_1, \ldots, L_n$ are all satisfied or not, respectively. The truth of these two atoms reflects the two alternative cases of the definition of ch given by (6). As before, z acts as a counter of the number of times ρ' is fired. Depending on the truth of p (resp., np), each firing of ρ' is forced to correspond to one firing of ρ'' (resp., of one among the ρ_is). All the above rules, except ρ'', do not involve p-lists. Hence, we are left with the translation of ρ'' which involves preferences on compound resources. The translation can be completed as outlined earlier in this section. As before, the ASP encoding can be generalized to treat r-rules involving more than one cp-list (even with different kinds of compound resources).

The second form of cp-list (namely, $(r \;\; only\_when \;\; L_1, \ldots, L_n)$), can be treated similarly. The ASP encoding for such form of cp-list largely coincides with the one described above. It is made of the r-rules ρ, ρ', together with the following one (in place of all ρ_is), to reflect the definition of ch given by (7):

$$(\rho''') \qquad [1\text{-}N_{h,2}]: \qquad pl\#1 \leftarrow np,\ z\#1.$$

Let us refine the translation $\mathcal{T}$ so that all cp-lists are translated as described above (the resulting r-program is then translated as explained earlier in the previous subsections). Completeness, soundness, and complexity bounds for such a refined $\mathcal{T}$ are easily obtained by proceeding in analogy to what done for the other encodings seen so far. We have the following two statements:

Proposition 4 *Let P be a ground r-program involving cp-lists and let P' be the grounding of the program $\mathcal{T}(P)$. Then, each answer set of P determines an answer set for P', and vice versa.*

Proposition 5 *The problem of deciding whether a ground r-program involving conditional preferences on compound resources has an answer set is NP-complete.*

4.6 Treatment of p-sets

Let us consider the following r-rule η (where Idx is as before):

$$Idx: \quad H \leftarrow B_1, \ldots, B_m, \{q_1\#a_1, \ldots, q_k\#a_k \mid pred\}.$$

$$
\begin{array}{ll}
(31) & trcl(PS,X,Y) \leftarrow dom(PS,X), dom(PS,Y), dom(PS,Z), \\
& \qquad trcl(PS,X,Z), trcl(PS,Z,Y), X \neq Y, pset\_name(PS). \\
(32) & equiv(PS,T_1,T_2) \leftarrow trcl(PS,T_1,T_2), trcl(PS,T_2,T_1), \\
& \qquad dom(PS,T_1), dom(PS,T_2), pset\_name(PS). \\
(33) & 1\{ord\_idx(PS,T,N) : number(PS,N)\}1 \leftarrow dom(PS,T), pset\_name(PS). \\
(34) & used\_idx(PS,N) \leftarrow dom(PS,T), ord\_idx(PS,T,N), \\
& \qquad number(PS,N), pset\_name(PS). \\
(35) & \leftarrow dom(PS,T), ord\_idx(PS,T,N), number(PS,N), \\
& \quad N > 1, N_1 = N-1, not\ \ used\_idx(PS,N_1), pset\_name(PS). \\
(36) & \leftarrow dom(PS,T_1), dom(PS,T_2), ord\_idx(PS,T_1,N_1), ord\_idx(PS,T_2,N_2), \\
& \quad number(PS,N_1), number(PS,N_2), N_2 \leqslant N_1, T_1 \neq T_2, \\
& \quad trcl(PS,T_1,T_2), not\ \ trcl(PS,T_2,T_1), pset\_name(PS). \\
(37) & \leftarrow not\ \ equiv(PS,T_1,T_2), dom(PS,T_1), dom(PS,T_2), ord\_idx(PS,T_1,N), \\
& \quad ord\_idx(PS,T_2,N), number(PS,N), T_1 \neq T_2, pset\_name(PS). \\
(38) & ord\_idx(PS,T_2,N) \leftarrow equiv(PS,T_1,T_2), dom(PS,T_1), dom(PS,T_2), \\
& \qquad ord\_idx(PS,T_1,N), number(PS,N), pset\_name(PS). \\
(39) & order(PS,T,N) \leftarrow ord\_idx(PS,T,N), not\ \ other(PS,T,N), dom(PS,T), \\
& \qquad number(PS,N), pset\_name(PS). \\
(40) & other(PS,T,N) \leftarrow order(PS,T_2,N), T \neq T_2, dom(PS,T), dom(PS,T_2), \\
& \qquad number(PS,N), pset\_name(PS).
\end{array}
$$

Figure 1: Fragment of RASP's inference engine dealing with p-sets

By introducing a fresh symbol ps (that is univocally associated with the p-set at hand), the r-rule η can be replaced by these r-rules:

$$
\begin{array}{lll}
(\eta') & Idx: & H \leftarrow B_1, \ldots, B_m, ps\#1. \\
(\eta'') & & ps\#N_{h,2}.
\end{array}
$$

These r-rules do not involve p-lists, hence they are treated as described earlier in this section. To complete the translation we have to take into account each possible total order implicitly represented by the extension of *pred* (i.e., its interpretation in the answer set at hand). To this aim, the following ASP fragment is introduced to handle the specific p-set:

$$
\begin{array}{ll}
(26) & dom(ps,q_i). \quad number(ps,i). \qquad \text{for } i \in \{1,\ldots,k\} \\
(27) & pset\_name(ps) \leftarrow fired(r_\eta). \\
(28) & trcl(ps,X,Y) \leftarrow pred(X,Y), X \neq Y, fired(r_\eta). \\
(29) & 1\{use\_pl(r_\eta,q_1,I,a_1,ps),\ldots,use\_pl(r_\eta,q_k,I,a_k,ps)\}1 \leftarrow \ counter(r_\eta,NumFirings), \\
& \qquad use(r_\eta,0,ps,-1{*}NumFirings),\ I \leqslant NumFirings, iter(I). \\
(30) & res\_pl(r_\eta,ps,T,N) \leftarrow order(ps,T,N).
\end{array}
$$

In the above code, line (26) defines two domain predicates that enumerate the (relevant) arguments of *pred* (let X be the set of these elements) and a collection of possible indices (representing different preference degrees), respectively. The ASP rules in Figure 1 are independent of the specific p-sets. In particular, the rule at line (31), together with the one in line (28), evaluates the transitive closure of the relation R defined by *pred*. Line (32) determines the equivalences between r-symbols. Rules at lines (33)–(35) index the elements of X with consecutive (possibly repeated) integers. Lines (36)–(38) restrict the possible indexes to those that do not violate the (closure of) the relation R. In particular, elements are assigned equal indices if and only if they are equally preferred (i.e., equivalent in R). Higher indices are assigned to less preferred r-symbols. Finally, rules (39)–(40) generate (compatibly with the extension of *pred*) all admissible orders for X. Each of these orders determines a corresponding p-list. Such indexing of r-symbols is used (in the context of a specific answer set) through the rules (29)–(30), seen before, analogously to what done for the translation of plain p-lists (cf. rules (23) in page 14).

As done before, let us further refine the translation $\mathcal{T}$ so that all p-sets are treated as described above. Completeness, soundness, and complexity bounds for such a refined $\mathcal{T}$ follow from the analogous results holding for RASP without p-sets.

In particular, completeness and soundness results for RASP enriched with all the forms of a-literals seen so far, can be stated as follows.

Theorem 3 *Let P be a ground r-program involving compound resources, p-lists, cp-lists and p-sets, and P' the grounding of $\mathcal{T}(P)$. Then, each answer set of P determines an answer set for P', and vice versa.*

Proof. (Sketch) The result follows from previously stated propositions, because each construct of RASP can be treated independently. Concerning p-sets, let us focus on a specific p-set *ps* and suppose the corresponding r-rule η is fired in an answer set for P. For simplicity, suppose also that *pred* is defined in P by means of a collection S of ground facts. Let Q' be the instances of (31)–(40) introduced in P' to translate *ps*. Observe that no rule/predicate in $S \cup Q'$ depend on literals occurring in $P \setminus S$. Conversely, no predicate in $P \setminus S$ depend on literals defined through rules in $S \cup Q'$, except for *order*, whose extension determines through rule (30) a specific p-list in P', and possibly *pred*. Thus, two strata of P can be identified, the bottom one containing the instances of (31)–(40). Consequently, an answer set for P' can be split into two parts, each one corresponding to the two strata of the program P'. One of such parts determines an answer set for the "upper" stratum, which includes (η''), (η'), and (26)–(30). Since we are assuming *fired*(r_η) true, the r-rules (26)–(30) can be simplified. This allows us to proceed in the proof as done for the case of plain p-lists. In fact, observe that (being fixed the extension of *order*), the treatment of p-sets mirrors the one devised for p-lists. To be convinced of this, it suffices to consider that (35), which reflects the preference degrees, surrogates the facts *res_pl* listed in (23), and to compare (η')–(η'') with (21)–(22), and (34) with the last r-rule in (23). □

We conclude with the main result on the complexity of RASP.

Theorem 4 *The problem of deciding whether a ground r-program involving all forms of a-literals has an answer set is NP-complete.*

Proof. The result is obtained by Proposition 5 and by observing that the translation obtained through rules (26)–(40) introduces a polynomially bounded number of ASP rules and literals (w.r.t. the length of the given r-program). □

5 Raspberry: a concrete implementation

The translation from RASP into ASP described earlier proceeds by generating a fragment of ASP code for each construct of the RASP language. Several of these fragments (namely (12), (13), (14), (18), (19), (24), and (31)–(40)), do not depend on the specific r-rule being translated. These ASP rules can be factorized in order to compose the core of an ASP-based inference engine that implements the crucial resource-oriented reasoning. This engine is joined to those fragments of the translation that are specific to the r-program at hand.

This is the approach we adopted in mechanizing RASP. The translation from (ground) RASP programs into ASP has been implemented in a stand alone tool, named *raspberry*, which is available in [28].[13]

ASP programs generated by raspberry can be processed by commonly available ASP-solvers (in particular, the input syntactic formats for gringo and lparse are both supported [2]). The answer sets found by the ASP solver encode the solutions of the RASP problem.

More specifically, raspberry has been developed in C++. The application consists of three main modules: a lexical analyzer, a parser and a translator. In developing the lexical analyzer and the parser we have adopted the same design choices adopted by the developers of the grounder gringo [14]. In particular, we made use of `re2c` to automatically generate the string tokenizer and `lemon` to generate the parser for the RASP language.

[13] Actually, non-ground programs are also correctly translated, provided that all a-atoms are ground (i.e., variables occur only in p-literals). The treatment of generic non-ground r-rules is subject for future work.

One reason to support the choice of these tools is that they allow for rapid development of readable, easily modifiable and extensible class-oriented code, in view of future extensions of RASP and its integration with a grounder. The translator is itself organized in a modular architecture consisting of a sequence of filters that act, in turn, on each r-rule. Each filter translates a specific RASP construct. Such an organization enables future extensions of the RASP language to be handled by adding further filters.

The translation is carried on via a single scan of the input r-program. For each rule in the input file raspberry acts depending on the kind of rule: p-rules, r-rules, and r-facts. If the rule at hand is a p-rule, it is left unchanged and added to the output. r-facts are translated by producing the corresponding ASP facts (16). The treatment of r-rules is more complex and exploits the sequence of filters. Each of these filters is in charge of generating the encodings of cp-lists, p-sets, p-lists and a-atoms, respectively.

During the scanning phase, some information on the r-program is gathered. For instance, data are collected about the kind of a-literals, the predicates, the magnitude of numeric constants appearing in the r-program. Such information is used for optimizing both the translation process (e.g., by disabling useless filters) and the output program (e.g., by adding suitable domain predicates to speed-up the subsequent grounding phase or to avoid the generation of unnecessary portions of the inference engine).

As regards preference criteria (cf., Remark 2 and [9]), the search for most preferred solutions is implemented by adding a fragment of ASP code that exploits the optimization features offered by smodels and clasp (see [2], for the details on these features).

The prototype currently encompasses all features described in this paper. Improvements are planned, mainly to achieve the generation of better ASP code, for instance by extracting more semantic information from the input r-program, in order to gain better efficiency in the grounding and solving phases.

Some command line options can be used to guide and tune the translation process. The most relevant of them allow one to:

- enable or disable the generation of some parts of the inference engine. For example, the addition of the ASP code needed to impose preference criteria can be inhibited (command line option `-m`). In this manner, some ASP solvers (such as clasp and smodels) are enabled to find all answer sets of the output ASP program, instead of focusing the search on the most preferred solutions;
- select some features of the target language. This might be useful to drive different (versions of) grounders, such as gringo and lparse, that accept slightly different syntax (command line options `-l3`, `-l2`, etc.);
- control input and output streams;
- get help on raspberry usage (command line option `-h`).

(See [28] or use the command line option `-h` to get an exhaustive description of all the options.)

In implementing RASP we have fixed the algebraic structure Q (modeling amounts) to be the set $\mathbb{Z}$ of integer numbers. We made this choice because commonly available ASP solvers offer operations on integers as built-in features. Refinements of the tool so as to deal with other groups are a theme for future work.

Observe that the concrete implementation strictly reflects the abstract implementation described in Section 4. Hence, its correctness and completeness directly follow.

The following examples provide more details on the translation and describe some of the experimentation we run by using raspberry, together with gringo 2.0.5 as grounder and clasp 1.3.3 as solver. (All experiments have been executed on an Intel Pentium 1.86GHz Linux machine.)

Example 8 *Let us consider the r-program of Example 6, completed by adding the definition of the predicate calory. The syntax accepted by raspberry largely coincides with the one we adopted in this paper. This is the r-program P processable by the translator:*

```
[1-1]: cake#1>cookies#15 :- egg#2, flour#2, raisin#4,
            ({aspartame#1, skimmilk#6}>{sugar#4, wholemilk#6} pref_when diet),
            ({vanilla#1; lemon#2}>cinnamon#1 only_when not allergy),
            {chocolate#2, nuts#1, coconut#1 | lesscaloric}.

lesscaloric(X, Y) :- calory(X, A), calory(Y, B), A < B.

calory(chocolate,5).        calory(coconut,5).        calory(nuts,10).
allergy.          diet.
egg#3.            flour#3.          raisin#5.
sugar#4.          aspartame#2.      skimmilk#6.       wholemilk#8.
vanilla#2.        lemon#2.          cinnamon#1.
chocolate#2.      nuts#2.           coconut#3.
```

As mentioned, the translation generated by raspberry reflects the encoding described in the previous sections. In order to make the output program processable by commonly available grounders (such as gringo or lparse), raspberry introduces a number of domain predicates to bound the admitted values for the variables appearing in the output program. (This is also needed to enable the evaluation of optimization statements and aggregate literals). Moreover, all predicate and constant symbols introduced by the translation of an r-program P, e.g., those in $\Pi_{\mathcal{T}}$ or in Π_{aux}, are conventionally prefixed by the string "`rasp_`". This makes them easily identifiable in $\mathcal{T}(P)$. Consequently, the generated ASP code appears less concise than the one described in this paper. We omit here the complete translation of the above program (the interested reader can retrieve it from the site [28], or simply run raspberry on the above r-rules).

By disabling the search for most preferred answer sets (by using the command line option `-m` of raspberry), 25 answer sets for the grounding of $\mathcal{T}(P)$ were found by clasp within 0.75 seconds of solving time. These 25 solutions correspond to: a solution that does not fire the r-rule and 24 different firings of it. In turn, these firings are obtained by combining 2 choices for the p-list in the head, 2 choices for the first cp-list, (whereas the second one has no effect, since allergy is true), and 6 different choices for the p-set.

The same r-program has been used in order to search for the most preferred solution. In particular, a most preferred answer set, w.r.t. the preference criterion $\mathcal{PC}_1$ of Remark 2 (actually, this is the only criterion currently implemented in the prototype), has been determined by clasp within 0.23 seconds of solving time. Such answer set of $\mathcal{T}(P)$ encodes an answer set $\mathcal{I} = \langle I, \mu, \xi \rangle$ for P such that $I = \{calory(chocolate, 5),\ calory(nuts, 10),\ calory(coconut, 5),\ less\_caloric(chocolate, nuts),\ less\_caloric(coconut, nuts),\ diet,\ allergy\}$ and $\xi(\gamma) = 1$ for all r-rules. The selected p-list, among those encoded by the p-set is coconut#1>chocolate#2>nuts#1. The allocation μ for the first r-rule γ is: $\mu(cake)(\gamma) = \{\!\{ \langle 1, 1\rangle \}\!\}$, $\mu(egg)(\gamma) = \{\!\{ \langle 0, -2\rangle \}\!\}$, $\mu(flour)(\gamma) = \{\!\{ \langle 0, -2\rangle \}\!\}$, $\mu(raisin)(\gamma) = \{\!\{ \langle 0, -4\rangle \}\!\}$, $\mu(aspartame)(\gamma) = \{\!\{ \langle 1, -1\rangle \}\!\}$, $\mu(skim\_milk)(\gamma) = \{\!\{ \langle 0, -6\rangle \}\!\}$, $\mu(coconut)(\gamma) = \{\!\{ \langle 1, -1\rangle \}\!\}$, and $\mu(x)(\gamma) = \{\!\{ \}\!\}$ for any other r-symbol x. The resulting resource balance is as follows: chocolate#2, nuts#2, coconut#2, cake#1, cookie#0, egg#1, egg#1, flour#1, raisin#1, vanilla#2, lemon#2, cinnamon#1, aspartame#1, skim_milk#0, sugar#4, whole_milk#8.

As a variation on P, consider the r-program P' obtained by replacing the two a-atoms aspartame#1 and coconut#2 in the first r-rule of P, by aspartame#3 and coconut#5, respectively.

In this case, only 9 possible answer sets have been found for the grounding of $\mathcal{T}(P')$ (within 0.35 seconds of solving time). This is so because the solutions that consume aspartame or coconut are no more admissible since not enough resources are available. A most preferred answer set, w.r.t. $\mathcal{PC}_1$, has been determined by clasp within 0.25 seconds of solving time. In this answer set the selected p-list, among those encoded by the p-set is chocolate#2>coconut#1>nuts#1. The allocation for the first r-rule γ is $\mu(cake)(\gamma) = \{\!\{ \langle 1, 1\rangle \}\!\}$, $\mu(egg)(\gamma) = \{\!\{ \langle 0, -2\rangle \}\!\}$, $\mu(flour)(\gamma) = \{\!\{ \langle 0, -2\rangle \}\!\}$, $\mu(raisin)(\gamma) = \{\!\{ \langle 0, -4\rangle \}\!\}$, $\mu(sugar)(\gamma) = \{\!\{ \langle 2, -4\rangle \}\!\}$, $\mu(whole\_milk)(\gamma) = \{\!\{ \langle 0, -6\rangle \}\!\}$, $\mu(chocolate)(\gamma) = \{\!\{ \langle 1, -2\rangle \}\!\}$, and $\mu(x)(\gamma) = \{\!\{ \}\!\}$ for any other r-symbol x.

The following example shows how biochemical reactions and pathways can be encoded in RASP.

Example 9 (Calvin-Benson-Bassham cycle) *The Calvin-Benson-Bassham cycle is the major fixation pathway for carbon dioxide (CO_2). It is found in green plants and many autotrophic bacteria [8]. In fact*

it is the basic reaction through which a hexose sugar (e.g., glucose) is produced, during the "dark phase" of photosynthesis, from carbon dioxide and water.

We consider here a simplified description of the cycle. Intuitively, the process captures carbon dioxide and makes the precursors of glucose by using chemical energy stored in specific molecules (i.e., c_NADPH and c_ATP, see below). This carbon fixation *process can be divided into three phases: fixation, reduction, and regeneration (more details can be found in [24, 11], among others).*

For the sake of simplicity, in what follows we denote chemical compounds and enzymes by using the acronyms listed in the following table (Note that, the same acronyms will serve as r-symbols and p-atoms in the RASP encoding):

COMPOUNDS:			
adenosine-triphosphate	*c_ATP*	*adenosine-diphosphate*	*c_ADP*
D-sedoheptulose-1,7-bisphosphate	*c_SBP*	*D-sedoheptulose-7-phosphate*	*c_S7P*
phosphate	*c_Pi*	*dihydroxyacetone phosphate*	*c_DHAP*
D-erythrose-4-phosphate	*c_E4P*	*D-glyceraldehyde-3-phosphate*	*c_G3P*
3-phospho-D-glycerate	*c_3PG*	*1,3-diphosphateglycerate*	*c_PGAP*
nicotinamide adenine dinucleotide phosphate	*c_NADP*$^+$	*reduced c_NADP*$^+$	*c_NADPH*
D-ribose-5-phosphate	*c_R5P*	*D-xylulose-5-phosphate*	*c_X5P*
D-ribulose-5-phosphate	*c_Ru5P*	*D-ribulose-1,5-bisphosphate*	*c_RuBP*
D-fructose-6-phosphate	*c_F6P*	*fructose-1,6-bisphosphate*	*c_FBP*
ENZYMES:			
ribulose bisphosphate carboxylase	*e_RubisCO*	*triose-phosphate isomerase*	*e_TIM*
ribose 5-phosphate isomerase	*e_R5IM*	*ribulose-phosphate-3-epimerase*	*e_REP*

The three phases of the overall cycle can be described by the following reactions:[14]

Fixation phase:

$c\_RuBP + CO_2 + H_2O \quad \Rightarrow \quad 2\ c\_3PG + 2\ H^+$ (*catalyzed by the enzyme: e_RubisCO*)

Reduction phase:

$c\_3PG + c\_ATP \quad \Rightarrow \quad c\_PGAP + c\_ADP + H^+$

$c\_PGAP + c\_NADPH + H^+ \quad \Rightarrow \quad c\_G3P + c\_NADP^+ + c\_Pi$

Regeneration phase:

$c\_G3P \quad \Rightarrow \quad c\_DHAP$ (*catalyzed by the enzyme: e_TIM*)

$c\_DHAP + c\_G3P \quad \Rightarrow \quad c\_FBP$

$c\_FBP + H_2O \quad \Rightarrow \quad c\_F6P + c\_Pi$

$c\_F6P + c\_G3P \quad \Rightarrow \quad c\_E4P + c\_X5P$

$c\_DHAP + c\_E4P \quad \Rightarrow \quad c\_SBP$

$c\_SBP + H_2O \quad \Rightarrow \quad c\_Pi + c\_S7P$

$c\_S7P + c\_G3P \quad \Rightarrow \quad c\_R5P + c\_X5P$

$c\_R5P \quad \Rightarrow \quad c\_Ru5P$ (*catalyzed by the enzyme: e_R5IM*)

$c\_X5P \quad \Rightarrow \quad c\_Ru5P$ (*catalyzed by the enzyme: e_REP*)

$c\_Ru5P + c\_ATP \quad \Rightarrow \quad c\_RuBP + c\_ADP + 2\ H^+$

In the first phase, the enzyme e_RubisCO catalyzes the carboxylation (by carbon dioxide) of one 5-carbon compound (c_RuBP) obtaining two molecules of a 3-carbon compound (c_3PG), and two protons. By using the energy stored in molecules of c_NADPH and c_ATP, the second phase converts c_3PG into molecules of c_G3P. In six iterations of the cycle six molecules of CO_2 are fixed and 12 molecules of c_G3P are produced. Two of them are converted into 2 molecules of c_F6P, which is the precursor of hexose sugar (namely, the final product of the overall cycle). The other 10 molecules of c_G3P are used in the remaining chains of reactions in order to re-generate 6 molecules of c_RuBP. These steps reconstitute the initial amount of CO_2-acceptor molecules. The cycle is regulated by the presence of several enzymes (we indicated just a few of them).

Each of the reactions can be modeled by using r-rule that involve multiple firings. For instance, the fixation phase is encoded as follows:

$$[1\text{-}N]: \quad c\_3PG\#2,\ proton\#2 \ \leftarrow \ c\_RuBP\#1,\ carbDioxide\#1,\ water\#1,\ e\_RubisCO.$$

[14] Notice that, w.r.t. the completely specified cycle [8, 24], in this example we applied some simplification. For instance, we indicated only a few of the enzymes that catalyze the reactions.

where we used the p-atom e_RubisCO to model the availability of the enzyme ribulose bisphosphate carboxylase *and r-atoms to model the compound elements that are produced/consumed. N stands for an integer number that, in order to reflect the above described production of c_G3P, has to be a multiple of 6. The other reactions are encoded analogously. The complete encoding can be found in [28]. We experimented with different values of N, to allow multiple iterations of the entire cycle, in correspondence of different requirements on the amount of c_G3P to be produced. The results are shown in Table 1.*

Our last example concerns conditional preferences in allocating resources.

Example 10 *Let us consider the following variant of the r-program of Example 4, where the atom $max\_need(T, M)$ denotes the maximum number of computers of kind T that we are allowed to assemble.*

$$[1\text{-}M]: \quad main\_unit(T)\#1 \leftarrow cpu\#1,\ motherboard\#1,\ ram\_module\#2,\ (\{scsihd\#4,\ cpu\#1,\ ram\_module\#2\} > \{eidehd\#2,\ cpu\#1,\ ram\_module\#2\}\ only\_when\ \ T = server),\ (eidehd\#2 > scsihd\#2\ only\_when\ \ T \neq server),\ max\_need(T, M).$$

$$[1\text{-}M]: \quad computer(T)\#1 \leftarrow main\_unit(T)\#1,\ monitor\#1,\ mouse\#1, keyboard\#1,\ pc\_type(T),\ max\_need(T, M).$$

$monitor\#25.$ $mouse\#25.$ $keyboard\#25.$ $motherboard\#30.$
$cpu\#29.$ $scsihd\#30.$ $eidehd\#29.$ $ram\_module\#30.$

Assume, moreover, that the predicates max_need and pc_type are defined as follows:

$max\_need(server, 2).$ $max\_need(desktop, N).$ $pc\_type(server).$ $pc\_type(desktop).$

with N replaced by a positive integer. Table 1 reports the result of our experimentation with r-programs obtained by performing the grounding after substituting a specific integers for N (cf., Example 7). In this case we report the time spent to find the optimal solution with respect to the expressed preferences and in case the maximum number of computers is assembled.

Instance	Solving time for the first solution	Instance	Solving time for the first solution	Solving time for the optimal solution
calvin-1-2	0.15	computers-1	0.61	12.20
calvin-2-2	15.06	computers-3	0.68	13.41
calvin-2-4	3.80	computers-5	0.77	14.24
calvin-3-4	83.85	computers-7	1.94	16.59
calvin-3-5	309.37	computers-9	2.09	17.46
calvin-3-6	312.44	computers-11	1.23	17.64

Table 1: Some experimental results. All timings are in seconds. In the instance calvin-N-K we require the production of K molecules of c_G3P and provide initial chemical compounds for at most N iteration of the carbon fixation cycle. (cf., Example 9). In the instance computers-N we search for the most-preferred assembling of $N + 2$ computers (cf., Example 10).

Concluding remarks

In this work we have presented some improvements to RASP, which is an extension of ASP that supports reasoning about resources and their amounts as well as the specification, in form of rules, of those processes that produce and consume resources. In fact, we have proposed several additions to the RASP language in order to allow for specification of complex forms of preferences on resource allocation. To accommodate these improvements, we have extended the RASP semantics and have evaluated the resulting complexity. Under this respect it turns out that new language constructs, though far from being trivial syntactic sugar,

do not imply any increase in the computational complexity of the framework. In particular, the problem of establishing the existence of an answer set for an r-program is still NP-complete. Consequently, the complexity of credulous reasoning for RASP, when such form of complex preferences are involved, is in line with that of LPOD [6]. The proposed extensions have been fully implemented.

The extension of (R)ASP that we have illustrated is, to the best of our knowledge, an original proposal. With respect to our previous work we may remark that, albeit cp-lists have been introduced in [9], only preferences between single a-atoms were considered and, as for p-sets, no complexity result was assessed. With respect to related work RASP proposes a different (and, even better, complementary) viewpoint upon preferences concerning, at least, two aspects.

First, the kind of preferences admitted in RASP have a *local scope*: each p-list (cp-list, or p-set) is meaningful in the context of a particular rule (which models a specific process in manipulating some resources). Consider, for instance, the r-program of Example 1, where completely antithetic orders are expressed in the two r-rules. Notice that both r-rules might be fired at the same time, as sufficient resources are available. However, such a local aspect is strictly related to the constraints on global resource balance and resource availability. Consequently, preferences stated locally in different rules might/should be expected to interact indirectly with those expressed in other rules. There is a clear difference w.r.t. the approach usually adopted in extending (logic) programming frameworks with preferences, where preference orders are *globally* applied (either on program rules or on the set of all ground atoms). A detailed comparison between RASP and previous approaches to preference reasoning can be found in [9].

A second novelty consists in the possibility of dealing with non-linear preference orders in resource usage. In this sense, a similar proposal recently appeared in literature is DLPOD [17]. In this case, pure ASP is considered and preferences are imposed on the truth of program literals occurring in heads of rules in analogy to what done in LPOD [6]: then, also in this proposal preferences have a global flavor. These preferences, in turn, determine an ordering of the answer sets of a DLPOD program. In [17] non-linear preferences are introduced by combining ordered and unordered disjunction in the same framework.

RASP and DLPOD have different purposes and tend to solve different problems: consequently, they have different expressive power. In particular, notice that DLPOD has LPOD as a special case. Hence, it is reasonable that RASP and DLPOD also differ as regards the computational complexity, for instance, of credulous reasoning. In fact, [17] conjectures Σ_P^3-completeness for DLPOD.

As future work, on the one hand we intend to experiment the implementation on realistic case-studies. On the other hand, we mean to further extend the approach towards more general non-linear preferences, and by introducing meta-level preferences (i.e., preferences about what one should prefer) that can be seen as an extension of the "budget policies" we introduced in [10].

Acknowledgements. The authors would like to thank Torsten Schaub and Sven Thiele for their support and advice about the gringo grounder. The parser of raspberry plainly extends the one included in the distribution of gringo. This research has been partially supported by GNCS-INdAM project "*Tecniche innovative per la programmazione con vincoli in applicazioni strategiche*", *project cod.2009.010.0336* (ricerca di base 2009), and MIUR-PRIN project "*Innovative and multi-disciplinary approaches for constraint and preference reasoning*".

References

[1] C. Anger, T. Schaub, and M. Truszczyński. ASPARAGUS – The Dagstuhl Initiative. *ALP Newsletter*, 17(3), 2004. See `http://asparagus.cs.uni-potsdam.de`.

[2] Web references for some ASP solvers. ASSAT: `http://assat.cs.ust.hk`; Ccalc: `http://www.cs.utexas.edu/users/tag/ccalc`; Clasp: `http://http://potassco.sourceforge.net`; Cmodels: `http://www.cs.utexas.edu/users/tag/cmodels`; DeReS and aspps: `http://www.cs.uky.edu/ai/`; DLV: `http://www.dbai.tuwien.ac.at/proj/dlv`; Smodels: `http://www.tcs.hut.fi/Software/smodels`.

[3] C. Baral. *Knowledge representation, reasoning and declarative problem solving.* Cambridge University Press, 2003.

[4] S. M. Brasil, Jr. Rules and principles in legal reasoning: A study of vagueness and collisions in artificial intelligence and law. *Information & Communications Technology Law*, 10(1), 2001.

[5] G. Brewka. Complex preferences for answer set optimization. In D. Dubois, C. A. Welty, and M.-A. Williams, editors, *Proceedings of 9th International Conference on the Principles of Knowledge Representation and Reasoning (KR'09)*, pages 213–223. AAAI Press, 2004.

[6] G. Brewka, I. Niemelä, and T. Syrjänen. Logic programs with ordered disjunction. *Computational Intelligence*, 20(2):335–357, 2004.

[7] G. Brewka, I. Niemelä, and M. Truszczyński. Preferences and nonmonotonic reasoning. *AI Magazine*, 29(4), 2010.

[8] R. Caspi, T. Altman, J. M. Dale, K. Dreher, C. A. Fulcher, F. Gilham, P. Kaipa, A. S. Karthikeyan, A. Kothari, M. Krummenacker, M. Latendresse, L. A. Mueller, S. Paley, L. Popescu, A. Pujar, A. G. Shearer, P. Zhang, and P. D. Karp. The MetaCyc database of metabolic pathways and enzymes and the BioCyc collection of pathway/genome databases. *Nucleic Acids Research*, 38, 2010.

[9] S. Costantini and A. Formisano. Modeling preferences and conditional preferences on resource consumption and production in ASP. *Journal of of Algorithms in Cognition, Informatics and Logic*, 64(1), 2009.

[10] S. Costantini and A. Formisano. Answer set programming with resources. *Journal of Logic and Computation*, 20(2):533–571, 2010.

[11] M. Cox and D. Nelson. *Lehninger Principles of Biochemistry*. Freeman & Co., 2004.

[12] A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. GASP: Answer set programming with lazy grounding. *Fundamenta Informaticae*, 96, 2009.

[13] W. Faber, G. Pfeifer, N. Leone, T. Dell'Armi, and G. Ielpa. Design and implementation of aggregate functions in the DLV system. *Theory and Practice of Logic Programming*, 8(5-6):545–580, 2008.

[14] M. Gebser, T. Schaub, and S. Thiele. GrinGo: A new grounder for answer set programming. In C. Baral, G. Brewka, and J. Schlipf, editors, *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 of *Lecture Notes in Computer Science*, pages 266–271. Springer, 2007.

[15] M. Gelfond. Answer sets. In *Handbook of Knowledge Representation*, chapter 7. Elsevier, 2007.

[16] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proceedings of the 5th International Conference and Symposium on Logic Programming (ICLP/SLP'88)*, pages 1070–1080. The MIT Press, 1988.

[17] P. Kärger, N. Lopes, A. Polleres, and D. Olmedilla. Towards logic programs with ordered and unordered disjunction. In *Proceedings of ASPOCP'08 Workshop of ICLP'08*, 2008.

[18] D. B. Kemp and P. J. Stuckey. Semantics of logic programs with aggregates. In V. A. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Logic Programming Symposium (ILPS'91)*, pages 387–401. The MIT Press, 1991.

[19] N. Leone. Logic programming and nonmonotonic reasoning: From theory to systems and applications. In C. Baral, G. Brewka, and J. Schlipf, editors, *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 of *Lecture Notes in Computer Science*, page 1. Springer, 2007.

[20] V. Lifschitz. Answer set planning. In D. De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 23–37. The MIT Press, 1999.

[21] V. Lifschitz and H. Turner. Splitting a logic program. In *Proceedings of the 11th International Conference on Logic Programming (ICLP'94)*, pages 23–37. The MIT Press, 1994.

[22] N. M. Luscombe, D. Greenbaum, and M. Gerstein. What is bioinformatics? A proposed definition and overview of the field. *Methods of Information in Medicine*, 40(4):346–358, 2001.

[23] V. W. Marek and M. Truszczyński. Stable logic programming - an alternative logic programming paradigm. In *25 years of Logic Programming Paradigm*, pages 375–398. Springer, 1999.

[24] C. K. Mathews, K. E. van Holde, and K. G. Ahern. *Biochemistry*. Prentice Hall, 1999.

[25] H. Prakken and G. Sartor. Argument-based logic programming with defeasible priorities. *Journal of Applied Non-classical Logics*, 7, 1997. Special issue on 'Handling inconsistency in knowledge systems'.

[26] T. Son and E. Pontelli. A constructive semantic characterization of aggregates in answer set programming. *Theory and Practice of Logic Programming*, 7(3), 2007.

[27] M. Truszczyński. Logic programming for knowledge representation. In V. Dahl and I. Niemelä, editors, *Proceedings of the 23rd International Conference on Logic Programming (ICLP'07)*, volume 4670 of *Lecture Notes in Computer Science*, pages 76–88. Springer, 2007.

[28] Web reference for Raspberry: an implementation of RASP. `http://www.dmi.unipg.it/formis/raspberry/`.