Introducing ASP recipes and ASP Chef

Mario Alviano*, Davide Cirimele and Luis Angel Rodriguez Reiners

DEMACS, University of Calabria, Via Bucci 30/B, 87036 Rende (CS), Italy

Abstract

Answer Set Programming (ASP) is gaining popularity in the Knowledge Representation and Reasoning community thanks to its high-level modeling capabilities that ease the fast prototyping of systems addressing complex tasks. On the other hand, ASP is not a general purpose programming language and therefore it is usually employed for specific tasks of possibly long and articulated pipelines. This article introduces the notion of ASP recipe, a chain of ingredients combining computational tasks typical of ASP with other operations of data manipulation and visualization. ASP recipes are at the core of ASP Chef, a simple, intuitive web app for analysing answer sets that is designed to ease the creation of pipelines of operations over sequences of interpretations.

Keywords

answer set programming, modular programming, knowledge visualization, UX design

1. Introduction

Knowledge Representation and Reasoning applications are characterized by complex domains whose features of interest are properly encoded in a computer-readable format so that automatic manipulation, synthesis and sophisticated reasoning tasks can be addressed by efficient, specialized engines [1]. In this context, Answer Set Programming (ASP) [2] is gaining popularity among researchers and practitioners thanks to its declarative approach to problem solving that combines linguistic high-level constructs typical of logic-based programming languages as well as advanced solving algorithms for combinatorial search and optimization [3].

Despite its theoretical unrestricted computational capabilities, allowing to simulate any Turing machine when uninterpreted function symbols are used [4], ASP is not designed or intended to be a general programming language. As such, ASP must be employed to address specific tasks of a broader pipeline, and therefore tasks addressed by ASP engines are expected to receive their input from other modules of the pipeline, possibly implemented in a different paradigm, and are as well expected to produce output that will be consumed by subsequent modules of the pipeline, again possibly implemented in different paradigms. (For example, [5] reports a framework for controlling articulated robots involving several modules, among them several that are powered by ASP.)

ICLP Workshops 2023

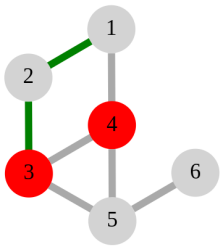
*Corresponding author.

mario.alviano@unical.it (M. Alviano); crmdvd98c05g975u@studenti.unical.it (D. Cirimele); luis.reiners@unical.it (L. A. Rodriguez Reiners)

https://alviano.net/ (M. Alviano)

0000-0002-2052-2063 (M. Alviano); 0009-0000-1808-9910 (L. A. Rodriguez Reiners)

CEUR Workshop Proceedings (CEUR-WS.org)




```
connected(1,2).   town(1..6).
connected(1,4).   townsintravel(3).
connected(2,3).
connected(3,4).
connected(3,5).
connected(4,5).
connected(5,6).
```

Figure 1: An instance of *Fighting with the gang of Billy the Kid* encoded by facts (right) and shown as a graph (left). All paths of 3 towns include at least one red node. An example path is shown in green.

```
1 connected(X,Y) :- connected(Y,X).  % symmetric closure
2 {select(T)} :- town(T).  % guess solution
3 :∼ select(T). [1@1, T]   % minimize selected towns
4 % no path of length n not including any selected town
5 :- townsintravel(N), path_length(_,N).
6 path_length((T,nil), 1) :- town(T), not select(T).
7 path_length((T',(T,X)), L+1) :- path_length((T,X), L), townsintravel(N), L < N, connected(T,T'),
      not select(T'), not in_path(T',X).
8 in_path(T, (T,X)) :- path_length((T,X), _).
9 in_path(T',(T,X)) :- path_length((T,X), _), in_path(T',X).
```

Figure 2: A monolithic ASP solution for *Fighting with the gang of Billy the Kid*. Note that paths of distinct towns not including those selected are materialized in the form $(t_1, (t_2, (\ldots, \texttt{nil})))$ by lines 6–9.

As a matter of fact, experienced ASP practitioners are used to map data from one format to the format accepted by ASP engines, and to map the output produced by ASP engines to some other format that is suitable to be presented to the end-user or to be further processed in the implemented pipeline. The problem of implementing such mapping procedures occurs at each ASP module of the pipeline and therefore it represents a not negligible task for development teams. On the one hand, the problem almost disappears if the pipeline is actually composed by a single ASP module, essentially either in the unrealistic hypothesis that the input and output format of ASP engines is suitable for the application to develop, or because there are only two mapping procedures to implement. On the other hand, even in such cases pretending to use ASP only in this way might severely limit its scope of application. In fact, without proper mechanisms enabling the exchange of data between ASP modules and modules of different nature, ASP practitioners, especially those at an early stage, may be tempted to use ASP as a single entity and as such find it convenient only to address computationally complex combinatorial problems, rather than as a solid formalism to be employed in broader pipelines.

This article introduces the notion of *ASP recipe* as a chain of ingredients that are essentially the instantiation of different operations. An operation can be one of the typical computational tasks for which ASP engines find a natural application, such as combinatorial search and optimization, some data manipulation procedure, or also some data visualization procedure. The idea is to provide a formalism in which the user can employ ASP directly by specifying a set of logic rules, but also indirectly via the operations that are implemented using an ASP engine.

```
1 connected(X,Y) :- connected(Y,X).
2 {path(1,T) : town(T)} = 1.
3 {path(I+1,T') : connected(T,T')} = 1 :-
        path(I,T), townsintravel(N), I < N.
4 :- path(I,T), path(J,T), I < J.
5 in_path(T) :- path(_,T).
```

```
1 {select(T)} :- town(T).
2 :~ select(T). [1@1, T]
3 ok(P) :- in_path(P,T), select(T).
4 :- in_path(P,_), not ok(P).
```

Figure 3: An ASP program to enumerate paths (left) and an ASP program to select towns to monitor (right). Note that the instances of `in_path` produced by the program on the left (line 5) must be enriched with a path identifier before being processed by the program on the right (lines 3–4).

For example, consider *Fighting with the gang of Billy the Kid*, a problem from the *LP/CP Programming Contest* series (https://github.com/lpcp-contest/). The input comprises a positive integer n, and a graph representing towns and streets. The goal is to select a minimum number of towns such that each path visiting n distinct towns includes at least one of the selected towns. Intuitively, the selected towns are those to monitor in order to catch bandits moving among n towns for their illicit activities. An instance and its solution are shown in Figure 1. The problem can be solved by a monolithic ASP encoding guessing (and minimizing) the selected towns, and enforcing that no path of n non-selected towns does exist; such a requirements can be enforced relying on uninterpreted function symbols, as shown in Figure 2. Approaching the problem with an ASP recipe may first rely on an ASP program whose answer sets are paths of n towns, enumerate all of them, then pack all paths as the input for another ASP program addressing the required combinatorial optimization task. Figure 3 reports such ASP programs, and Section 6 further elaborates on this solution to the problem.

One of the difficulties to face in order to enable the composition of linear chains of ingredients is the format to adopt for the input and output of each operation. Adopting different formats leads to increased complexity in terms of design of the notion of ASP recipe as well as in their composition, as essentially compatibility of formats must be taken into account at each chain junction. Such a difficulty is circumvented by adopting a uniform format for input and output of all operations, that is, *sequences of interpretations*. Additionally, operations can have parameters to customize their behavior, and also have side output to enable inspection and visualization of intermediate states of the evaluation of ASP recipes.

This work also introduces ASP Chef, a web app designed to implement possibly long pipelines in which ASP is often used as a core engine to address several computational tasks and putting in practice the notion of ASP recipe. We remark here that the main enabling mechanism adopted by ASP Chef to ease the application of ASP in possibly long pipelines is a uniform format for the input and output of each operation in the pipeline. Thanks to such a uniformity, the several operations implemented in ASP Chef can be combined in any order, and new operations can be accommodated easily in the future. Actually, among the operations already implemented in ASP Chef, there are operations to ease the embedding of structured and semi-structured data, so that the restriction to the format of sequences of interpretations becomes almost immaterial. In fact, the restriction is bypassed by operations relying on *Base64* encoding as Base64 strings are valid terms for ASP engines, and they are conceived to encode any binary data. For example, the *ParseCSV* operation maps a *comma-separated values* (CSV) content into a sequence of facts

that is added to each interpretation in input to produce an output enabling the processing of the CSV data (see Section 4.6). The ASP encoding itself can be composed and manipulated by means of the *Encode* operation, which essentially encodes in Base64 an arbitrary content and extends each interpretation in input with a new fact carrying such data (see Section 4.5). In the next sections, after introducing some background (Section 2), we introduce the notions of operation, ingredient and recipe (Section 3), and formalize some of the around 50 operations currently implemented in ASP Chef (Section 4). Finally, a couple of examples are provided in Section 6, related work is discussed in Section 7, and concluding remarks and future works are given in Section 8.

2. Background

We assume that the reader is familiar with the language of ASP [6], and only introduce the background notions to which ASP practitioners may be less accustomed.

Let $Base64$ be the function associating every binary string with a longer binary string that can be interpreted as printable ASCII characters; the output string is obtained according to RFC 4648 §4 (https://datatracker.ietf.org/doc/html/rfc4648#section-4). Let $Base64^{-1}$ be the inverse function of $Base64$.

Example 1. Let t be the following two-lines text:

```
hello world
1 2 3
```

$Base64(t)$ is (the binary string associated with) the ASCII string `aGVsbG8gd29ybGQKMSAyIDM=`. The two-lines text t can be obtained by applying $Base64^{-1}$ to `aGVsbG8gd29ybGQKMSAyIDM=`.

In the following the term *object* refers to any finite element of a fixed universe (comprising, among other elements, strings, numbers, graphs, atoms, programs, and sequences). The notation $(x_1, \ldots, x_n)$ is used to refer to a (finite) *sequence* of $n \geq 0$ objects, where each object x_i is associated with index i. Moreover, by *space* we refer to a (possibly infinite) set of objects. Finally, the Boolean values are denoted by $\mathbf{T}$ and $\mathbf{F}$.

3. ASP Recipes

Terms are defined as usual by inductively combining constants and functions of positive arity; similarly, atoms combine predicates and terms. An *interpretation* is a (finite) sequence of distinct atoms, that is, an interpretation is a set of atoms also associating each atom with an index (starting from 1); among the interpretations we include the inconsistent interpretation $\perp$ to represent errors. Let $\mathcal{I}$ be the set of all *sequences of interpretations*.

Example 2. $\mathcal{I}$ includes, among others, the sequence $((hello), (hello), (hello, world))$ of interpretations. Note that atoms within the same interpretation must be distinct, while the same interpretation may occur multiple times. Here, the last interpretation has index 3, and in that interpretation atom $hello$ has index 1 and atom $world$ has index 2.

An *operation* O is a function with signature $O : \mathcal{I} \longrightarrow \mathcal{I}$, that is, a function receiving in input a sequence of interpretations and producing in output a sequence of interpretations.

Example 3. Let Idx be the operation associating every atom in input with its index in the interpretation it occurs. More formally, if the input contains an interpretation with index i and atom α occurs at index j in that interpretation, then the output contains an interpretation with index i and atom $index(j, \alpha)$ at index j in this interpretation.

An *operation O with side output space $\mathcal{S}$* is a function with signature $O : \mathcal{I} \longrightarrow \mathcal{I} \times \mathcal{S}$. No particular restriction is imposed to the side output space; common cases include the set of strings, the set of graphs, and the set containing the empty set (to essentially have operations with no side output as a special case); to lighten the notation, if the side output space is $\{\emptyset\}$, we simply omit it. Let $O|_{\mathcal{I}}$ denote the operation obtained from O by discarding the side output.

Example 4. Let $Table$ be the operation reproducing in output its input, and additionally having as side output a sequence of tables, one for each interpretation in input, each one containing a row for each atom in the interpretation and columns for predicates and arguments of atoms. Note that $Table|_{\mathcal{I}}$ is essentially a *NOP* (an operation that does nothing).

A *parameterized operation O with parameter space $\mathcal{P}$ and side output space $\mathcal{S}$* is a function with signature $\mathcal{P} \longrightarrow (\mathcal{I} \longrightarrow \mathcal{I} \times \mathcal{S})$, that is, $O\langle P \rangle$ is an operation with side output space $\mathcal{S}$ for each parameter value $P \in \mathcal{P}$ (note that angle brackets are used to denote a *parameterized operation instantiation*). No particular restriction is imposed to the parameter space; common cases include the set of integers, the set of strings, sets of tuples, and the set containing the empty set (to have non-parameterized operations as a special case); to lighten the notation, if the parameter space is $\{\emptyset\}$, we simply omit it.

Example 5. Let $Index$ be the function associating every predicate name p with the following operation: if the input contains an interpretation with index i and atom α occurs at index j in that interpretation, then the output contains an interpretation with index i and atom $p(j, \alpha)$ at index j in this interpretation. Note that $Index\langle index \rangle$ is the operation Idx from Example 3.

An *ingredient* is an instantiation of a parameterized operation with side output, that is, if O is a parameterized operation with parameter space $\mathcal{P}$ and side output space $\mathcal{S}$, and $P \in \mathcal{P}$ is a parameter value, then $O\langle P \rangle$ is an ingredient. A *recipe* is a tuple of the form $(encode, Ingredients, decode)$, where $Ingredients$ is a (finite) sequence of ingredients, and $encode$ and $decode$ are Boolean values. Intuitively, the input of a recipe is either (the string representation of) a sequence of interpretations in $\mathcal{I}$, or a string to be $Base64$-encoded. The input is processed by the pipeline of ingredients, possibly producing some side output along the way. Finally, some encoded content is possibly decoded. Such an intuition is formalized below.

Let $Ingredients$ comprise $n \geq 0$ ingredients $O_1\langle P_1 \rangle, \ldots, O_n\langle P_n \rangle$, and let s_{in} be the string in input. The output and side output of the recipe $(encode, Ingredients, decode)$ given s_{in} are respectively s_{out} and $S_1, \ldots, S_n$ defined as follows:

- I_0 is one of (i) ((`__base64__("s")`)), if $encode$ is true, where $s = Base64(s_{in})$; (ii) the sequence of interpretations represented by s_{in}, where interpretations are separated by the reserved character § and atoms are represented as facts, if s_{in} conforms to this specification; (iii) $(\bot)$ otherwise, to report an error.

- For each $i = 1, \ldots, n$, let I_i be $O_i\langle P_i\rangle|_{\mathcal{I}}(I_{i-1})$, and S_i be the side output of $O_i\langle P_i\rangle(I_{i-1})$. Essentially, each ingredient of the recipe receives a sequence of interpretations from the previous computational step, and produces in output a sequence of interpretations for the next computational step. Possibly, a side output is also produced.
- Let s_{out} be the string obtained by concatenating the atoms of I_n represented as facts, one per line, and using § as the separator for interpretations. If *decode* is true, s_{out} is further processed by replacing every occurrence of `__base64__("`s`")` with (the ASCII string associated with) $Base64^{-1}(s)$; in this case, if $Base64^{-1}(s)$ produces an error, then s_{out} is simply the string representation of $(\bot)$.

Let $R(s_{in}) = (s_{out}, S_1, \ldots, S_n)$ denote the fact that s_{out} and $S_1, \ldots, S_n$ are the output and side output of a recipe R given an input string s_{in}.

Example 6. Recall t from Example 1, its Base64-encoding `aGVsbG8gd29ybGQKMSAyIDM=`, and the operations *Table* and *Index* defined in Examples 4–5. Let R_1, R_2 be the recipes $(\mathbf{T}, (Table), \mathbf{T})$ and $(\mathbf{F}, (Table, Index\langle idx\rangle), \mathbf{T})$. Hence, $R_1(t) = (t, (table_1))$, where $table_1$ has one row with two cells, namely `__base64__` and `"aGVsbG8gd29ybGQKMSAyIDM="`. In fact, t is first encoded, produced as $table_1$, and finally decoded back to t. Similarly, for t', t'' being the strings `__base64__("aGVsbG8gd29ybGQKMSAyIDM=").` and `idx(1,__base64__("aGVsbG8gd29ybGQKMSAyIDM="))`., it can be checked that $R_2(t') = (t'', (table_1), \emptyset)$.

4. Core Operations

This section introduces some operations by specifying how they process input. Parameter spaces are usually intuitive and parameter values that do not influence the output are possibly denoted with an underscore. Regarding side output spaces, if not stated otherwise, they are assumed to be $\{\emptyset\}$; empty side output is also usually omitted.

4.1. Searching

Let $SearchModels\langle decode\_predicate, echo, rules, number\_of\_models\rangle$ be the operation mapping $(I_1, \ldots, I_n)$ to $(I_{1,1}, \ldots, I_{1,n_1}, I_{2,1}, \ldots, I_{n,n_n})$, where each $I_{i,1}, \ldots, I_{i,n_i}$ is obtained by enumerating up to $number\_of\_models$ answer sets of the program $rules \cup \{Base64^{-1}(c) \mid decode\_predicate(\text{“}c\text{”}) \in I_i\} \cup \{p(\overline{c}). \mid p(\overline{c}) \in I_i, p \neq decode\_predicate$ or $echo = \mathbf{T}\}$. As a special case, $number\_of\_models = 0$ is used to enumerate all answer sets.

Example 7. Let s_{in} be the fact

```
hello.
```

and R be the recipe $(\mathbf{F}, (SearchModels(\_, \_, rules_1, 0),\ (SearchModels(\_, \_, rules_2, 0)), \mathbf{F})$, where

$rules_1$: `{world}.`
$rules_2$: `wonderful. world.`

Hence, $R(s_{in})$ is the following text:

```
hello.
wonderful.
world.
§
hello.
world.
wonderful.
```

Note that after the application of the first ingredient there are essentially two answer sets, namely `{hello}` and `{hello, world}`. Each of them is processed by the second ingredient, which adds the facts `wonderful` and `world`.

Let *Optimize* be an operation with the same parameter space and behavior of *SearchModels*, but enumerating optimal answer sets (according to the objective function given in terms of weak constraints, as usual in ASP).

4.2. Sorting and Removing Duplicates

Example 7 highlights that the order of interpretations and of atoms within them is not in general under the control of all operations. In fact, the enumeration of answer sets does not follow a specific order, and atoms within an answer set have no fixed order. Control over the order of interpretations can be imposed by adding ingredients conceived for this purpose.

Let *SortCanonical* be the operation sorting atoms within each interpretation according to their lexicographical ordering (predicate first, then the first argument, and so on). The ordering of terms gives priority to number, then to constants, finally to functions. Let $SortByPredicateOrArgument\langle index, desc\rangle$ be the operation sorting atoms in each interpretation according to their predicates (if $index = 0$) or their $index$-th arguments (if $index \neq 0$). The ordering is descending if $desc = \mathbf{T}$, and ascending otherwise. Let $SortModelsCanonically\langle excluded\_predicates\rangle$ be the operation sorting the interpretations according to their string representation, obtained after removing all instances of predicates in $excluded\_predicates$. Let $Unique\langle excluded\_predicates\rangle$ be the operation producing in output the first interpretation in input, and every other interpretation that is not immediately preceded by a copy of itself, once all instances of predicates in $excluded\_predicates$ are removed; often this operation is preceded by a sort operation to remove all duplicates from the input.

Example 8. If R in Example 7 is extended with *SortCanonical*, the output will contain two copies of the first interpretation. If R is further extended with $SortModelsCanonically\langle\emptyset\rangle$ and $Unique\langle\emptyset\rangle$, the output will contain only one copy of the first interpretation.

4.3. Filtering and Selecting

Let $Filter\langle regex\rangle$ be the operation preserving only atoms whose string representation matches the given regular expression $regex$, in each interpretation in input. Let $SelectPredicates\langle predicates\rangle$ be the operation preserving only instances of predicates in the set $predicates$, for each interpretation in input. Let $SelectModel\langle index\rangle$ be the operation preserving only the $index$-th interpreation in input.

4.4. Merging and Splitting

Let $Merge\langle predicate\rangle$ be the operation mapping $(I_1, \ldots, I_n)$ to the interpretation containing an atom $predicate(i)$ for each $i = 1, \ldots, n$ and an atom $predicate(i, \alpha)$ for each $\alpha \in I_i$. Let $Split\langle predicate\rangle$ be the operation replacing every interpretation in input with the interpretations encoded by instances of $predicate$, according to the schema defined by $Merge$; that is, in each model I in input, each $predicate(i) \in I$ leads to one model in output comprising α for each $predicate(i, \alpha) \in I$.

4.5. Closures and Encoded Content

Let $SymmetricClosure\langle input\_predicate, closure\_predicate, encode\_predicate\rangle$ be the operation extending each interpretation in input with the rules

```
closure_predicate(X,Y) :- input_predicate(X,Y).
closure_predicate(X,Y) :- input_predicate(Y,X).
```

Actually, interpretations in input are extended with the atom $encode\_predicate(c)$, where c is the Base64-encoding of the above rules.

Example 9. Let R be the recipe $(\mathbf{F}, (SymmetricClosure\langle \texttt{link}, \texttt{link}, \texttt{b64}\rangle, SearchModels\langle \texttt{b64}, \mathbf{F}, \emptyset, 1\rangle), \mathbf{F})$. Let s_{in} be `link(a,b)`. Hence, $R(s_{in})$ produces in output the interpretation consisting of `link(a,b)` and `link(b,a)`.

Similarly, *TransitiveClosure* extends interpretations in input with the rules

```
closure_predicate(X,Y) :- input_predicate(X,Y).
closure_predicate(X,Z) :- closure_predicate(X,Y), input_predicate(Y,Z).
```

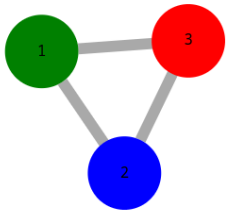
Finally, an arbitrary content can be Base64-encoded by $Base64\langle encode\_predicate, content\rangle$.

4.6. Working with Comma Separated Values

Let $ParseCSV\langle decode\_predicate, echo, separator, output\_predicate\rangle$ be the operation extending each interpretation I_i in input with atoms representing a CSV content $Base64^{-1}(c)$, for each $decode\_predicate(\text{“}c\text{”}) \in I_i$. Each $value$ occurring in cell (row, col) in the CSV content is represented by an atom of the form $output\_predicate(row, col, value)$. The instances of $decode\_predicate$ are removed from the output if $echo = \mathbf{F}$, and the separator used by CSV can be specified (e.g., TAB, SPACE, or a string). Let $GenerateCSV\langle input\_predicate, echo, encode\_predicate, separator\rangle$ be the operation producing a CSV content from instances of $input\_predicate$ in input, following the schema used by $ParseCSV$. The CSV content is Base64-encoded and stored as an instance of $encode\_predicate$. Instances of $input\_predicate$ are removed from the output if $echo = \mathbf{F}$, and the separator to use in the CSV content can be specified.

Example 10. Let t be the two-lines text from Example 1. Let R be the recipe $(\mathbf{T}, (ParseCSV\langle \texttt{\_\_base64\_\_}, \mathbf{F}, \text{SPACE}, \texttt{cell}\rangle), \mathbf{F})$. Hence, $R(t)$ produces in output the interpretation consisting of the following facts:

```
cell(1, 1, "hello").  cell(1, 2, "world").
cell(2, 1, 1).        cell(2, 2, 2).        cell(2, 3, 3).
```




```
g(node(1),color(green),label(1)).
g(node(2),color(blue),label(2)).
g(node(3),color(red),label(3)).
g(link(1,2)).
g(link(1,3)).
g(link(2,3)).
g(defaults,undirected).
```

Figure 4: A graph and its 3-coloring (left), as well as its fact representation (right).

4.7. Visualizing Side Output

Let *Table* be the operation forwarding its input and having as side output a sequence of tables, as described in Example 4. Let *Output* be the operation forwarding its input and having as side output its input. Let $OutputEncodedContent\langle predicate, echo\rangle$ be the operation forwarding its input, possibly after removing instances of *predicate* if $echo = \mathbf{F}$, and having as side output the Base64-decoded content of all instances of *predicate*.

Let $Graph\langle predicate, echo\rangle$ be the operation forwarding its input, possibly after removing instances of *predicate* if $echo = \mathbf{F}$, and having as side output a graph described by instances of *predicate*. The first argument of such instances is one of `node(ID)`, `link(SOURCE,TARGET)` and `defaults`. The other arguments have the form $property(VALUE)$. Node properties include `color`, `label`, `fx`, `fy`, `radius`, and `shape`. Link properties include `color`, `label`, `directed`, and `undirected`. Default properties include `node_color`, `node_radius`, `node_shape`, `link_color`, `directed`, and `undirected`.

Example 11. For an undirected graph encoded by predicate `link` and one of its k-coloring encoded by predicate `assign`, the following program can be used to obtain the facts shown in Figure 4 (right):

```
g(defaults, undirected).
g(node(X), color(C), label(X)) :- assign(X,C).
g(link(X,Y)) :- link(X,Y).
```

The above program can be used as a parameter for *SearchModels*, and the graph shown in Figure 4 (left) can be obtained by adding the ingredient $Graph\langle \mathrm{g}, \mathbf{F}\rangle$.

5. Implementation

The notion of ASP recipe is put in practice by ASP Chef, a web app powered by the JavaScript framework Svelte (https://svelte.dev/) and by the ASP engine clingo-wasm (https://github.com/domoritz/clingo-wasm), a WebAssembly version of clingo [7]. ASP Chef is freely available at https://asp-chef.alviano.net/, and its source code can be downloaded from https://github.com/alviano/asp-chef. The only requirement is a modern browser (Firefox is recommended).

Among the most interesting features of ASP Chef there is the representation of the recipe and its input in the URL as a zipped JSON object. Such an object is placed in the hash fragment (the

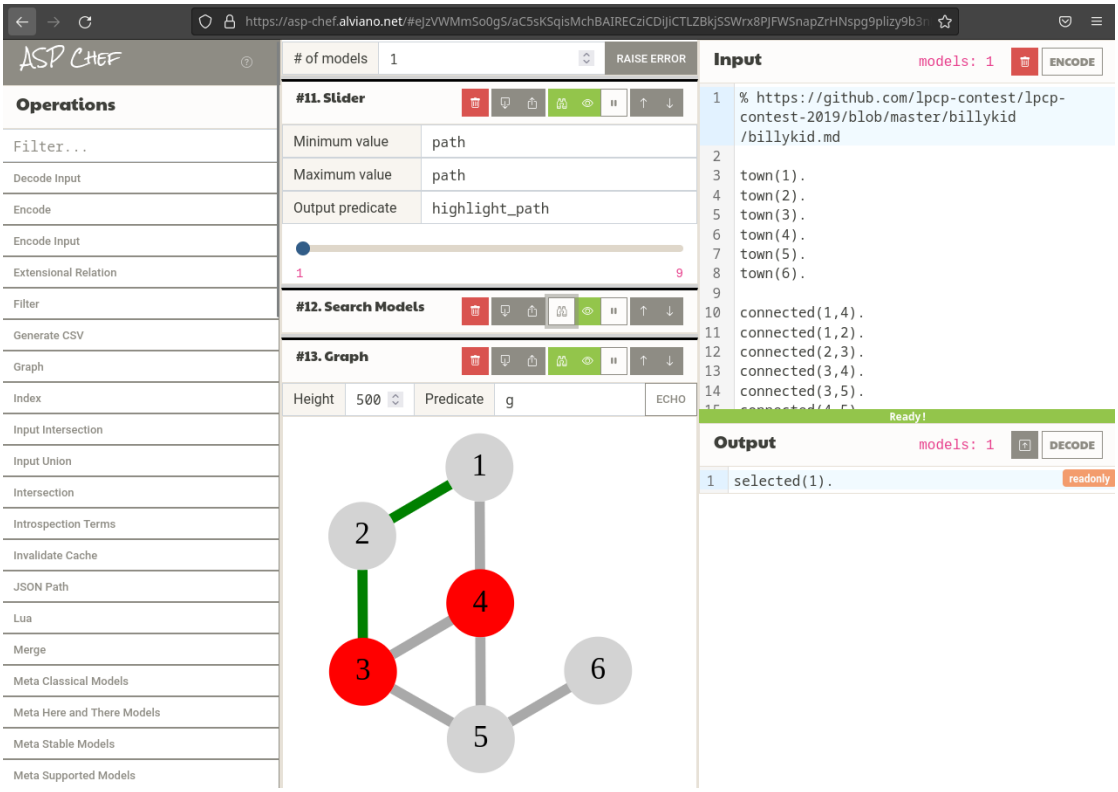


Figure 5: A screenshot of ASP Chef solving the instance of *Fighting with the gang of Billy the Kid* given in the introduction.

portion following the character #) so that it is not transmitted to the server. In fact, once ASP Chef is loaded in the browser, there is no further interaction with the server, which guarantees that the server cannot essentially be overloaded, and also ensures that *any sensitive data used by programmers remains confidential.*

Regarding the user interface of ASP Chef, it is inspired by CyberChef (https://github.com/gchq/CyberChef), a web app for encryption, encoding, compression and data analysis. As such, it splits the window in three columns, placing all operations on the left pane, input and output on the right pane, and the recipe in the middle pane; see Figure 5. Operations can be added to the recipe with their default parameter values, and different parameter values can be given in the recipe itself. Ingredients can be hidden, skipped, set as a termination point, as well as moved up and down in the recipe. A brief description of each operation is shown as a tool-tip to smooth out the learning curve.

Another interesting feature of ASP Chef is its caching mechanism. Ingredients are applied whenever the programmer changes something, taking advantage of the reactivity of SvelteKit. However, ASP Chef is designed so that information flows only in one direction, namely from the input to the first ingredient, and then from each ingredient to the next until the final output is produced. Such a design choice enables a simple, yet powerful caching strategy: if an ingredient in the recipe is changed, the output and side output of all ingredients occurring before it are

```
in_path(M,T) :- model(M, in_path(T)).
town(T) :- model(M, town(T)).
connected(X,Y) :- model(_, connected(X,Y)).

g(node(T), label(T)) :- town(T).
g(node(T), color(red)) :- select(T).
g(link(X,Y), undirected) :- connected(X,Y), X < Y.
```

Figure 6: ASP programs for the use case *Fighting with the gang of Billy the Kid* reported in Section 6.

```
size(S) :- c(3,1,S).
colors(C) :- c(3,2,C).
btn(X-3,Y,C) :- c(X,Y,C), X > 3.

{step(1..S*S/2)} :- size(S).
:- step(S), S > 1, not step(S-1).

btn(X,Y,C,0) :- btn(X,Y,C).

selector(0,1). selector(1,0). selector(1,1). selector(1,-1).
{pair((X,Y), (X+(D*SX),Y+(D*SY)), S)} :- selector(SX,SY), step(S);
    btn(X,Y,C,S-1); size(SIZE), D = 1..SIZE-1, btn(X+(D*SX),Y+(D*SY),C,S-1);
    #count{D' : D' = 1..D-1, btn(X+(D'*SX),Y+(D'*SY),C',S-1), C' != C} = 0.
:- step(S), #count{FROM, TO : pair(FROM, TO, S)} != 1.

cut(X,Y..Y',S) :- pair((X,Y), (X,Y'), S), Y < Y'.
cut(X..X',Y,S) :- pair((X,Y), (X',Y), S), X < X'.
cut(X+L,Y+L,S) :- pair((X,Y), (X',Y'), S), X < X', Y < Y', L=0..X'-X.
cut(X+L,Y-L,S) :- pair((X,Y), (X',Y'), S), X < X', Y > Y', L=0..X'-X.

btn(X,Y,C,S) :- btn(X,Y,C,S-1), step(S), not cut(X,Y,S).

:- btn(_,_,_,S), not step(S+1).

cell(1,1,S) :- S = #count{A,B,T : pair(A,B,T)}.
cell(T+1,1,X ) :- pair((X,Y), (X',Y'), T).
cell(T+1,2,Y ) :- pair((X,Y), (X',Y'), T).
cell(T+1,3,X') :- pair((X,Y), (X',Y'), T).
cell(T+1,4,Y') :- pair((X,Y), (X',Y'), T).
```

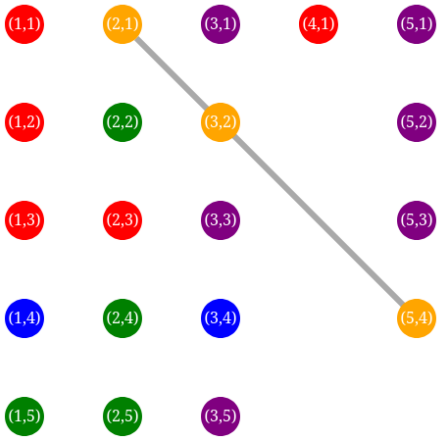
Figure 7: ASP programs for the use case *Buttons and Scissors* reported in Section 6.

those previously computed and recoverable from the cache.

Finally, ASP Chef takes inspiration from *initial learning environments* like Scratch [8] and *low-code development platforms* [9] like Appian (https://appian.com/), which are essentially easy to use visual environments for defining possibly complex procedures in terms of components and their interactions. Within this respect, ASP Chef provides ingredients as the analogous of components, and their interactions are simplified to the extreme in a single-direction linear flow as done by CyberChef. All in all, a programmer is expected to write less code to define an ASP recipe than an equivalent pipeline in a general purpose programming language, even if the amount of ASP code to write in ASP recipes is not necessarily low.

6. Use Cases

Let us consider *Fighting with the gang of Billy the Kid* from Section 1, and the input s_{in} shown in Figure 1 (right). Below are the ingredients of an ASP recipe addressing this problem.



Input

1	5 6
2	1 1 1 2 3
3	4 3 1 3 3
4	5 4 5 2 5
5	1 6 6 2 6
6	5 5 5 4 2

Output

1	11
2	4 4 5 5
3	4 2 4 5
4	2 1 5 4
5	1 5 2 4
6	3 1 3 3
7	1 1 1 3
8	2 3 4 1
9	2 2 2 5
10	3 5 5 3
11	1 4 3 4
12	5 1 5 2

Figure 8: Graph visualization of a specific cut in the solution of *Button and Scissors* (left) and CSV encoding of the input and output, a problem from *LP/CP Programming Contest*.

1. *SymmetricClosure*⟨`connected`, `connected`, `b64`⟩, to introduce the rules enforcing the symmetric closure of `connected`, since the graph is undirected.
2. *SearchModels*⟨`b64`, **F**, Figure 3 (left), 0⟩, to enumerate all paths of interest.
3. *SortCanonical*, *SortModelsCanonically*⟨{`path`}⟩, and *Unique*⟨{`path`}⟩, to remove paths associated with the same set of towns.
4. *Merge*⟨`model`⟩, and *SearchModels*⟨_, _, Figure 6 (lines 1–3), 1⟩, to pack all paths in a single interpretation. Note that paths are assigned an identifier, which is the index of the interpretation they occur in the input sequence of *Merge*.
5. *Optimize*⟨_, _, Figure 3 (right), 1⟩, to solve the combinatorial optimization subtask.
6. *SearchModels*⟨_, _, Figure 6 (lines 4–6), 1⟩, and *Graph*⟨g, **F**⟩, to show a graph.

A slightly extended recipe was used to obtain the screenshot shown in Figure 5.

Let us consider *Buttons and Scissors* from *LP/CP Programming Contest*: a grid of buttons of different colors is given, and the goal is to detach all buttons with a sequence of straight cuts involving buttons of the same color. The input is given in CSV, and the output is expected in CSV, so both *encode* and *decode* flags are set to true. The ingredients are given below.

1. *ParseCSV*⟨`__base64__`, **F**, SPACE, c⟩, and *SearchModels*⟨_, _, Figure 7 (lines 1–3), 1⟩, to accommodate the input in a format suitable for ASP.
2. *SearchModels*⟨_, _, Figure 7 (lines 4–17), 1⟩, to address the combinatorial search subtask of the problem. Without going into too much details, the sequence of cuts is guessed and the grid is updated after each cut, requiring to have no buttons after the last cut.
3. *SearchModels*⟨_, _, Figure 7 (lines 18–22), 1⟩, to represent a CSV output: the first line is the number of cuts, and the other lines give the coordinates of the starting and ending point of each cut.

4. *GenerateCSV*⟨cell, **F**, __base64__, SPACE⟩, and *SelectPredicates*⟨__base64__⟩, to leave only the CSV content in the output.

The above recipe can be extended to produce the graph shown in Figure 8, where nodes are given fixed positions by using the properties `fx` and `fy`. Figure 8 also shows a graph visualization for another problem from *LP/CP Programming Contest*, where node labels use emoji to represent walls (red squares) around points of interest (integers) and attacked cells (swords).

7. Related Work

Tools supporting the development of ASP programs were presented by many other works in the literature. Many of them focused on the design of Integrated Development Environment (IDE), like ASPIDE [10], SeaLion [11], and LoIDE [12]. While ASPIDE and SeaLion are desktop applications, LoIDE is designed as a web application. Comparing to ASP Chef, *which is not intended to be an IDE*, LoIDE requires a backend server to complete ASP computational tasks, while ASP Chef can run everything in the browser thanks to clingo-wasm. Other works dealt with simplifying the development of ASP modules and microservices [13, 14, 15, 16, 17], and share with ASP Chef the goal of employing ASP in a non-monolithic way.

Visualization of ASP output was approached by many other works in the literature. Among them, there are ASPViz [18], IDPD3 [19] and Kara [20], which are all based on the use of several predicates to describe how to draw a graphical representation of the solution provided by an answer set solver. The idea of easing the declarative specification of a graphical representation for ASP was revived in an interesting teaching experiment [21], which however is restricted to problems dealing with grids like Sudoku. On the same spirit, the more recent clingraph [22] can produce high-quality graphs and images that can be exported in LaTeX.

These works are related to the *Graph* operation introduced in Section 4.7, which is also capable of producing a graphical representation by interpreting instances of one user-specified predicate. At its current state, the *Graph* operation does not reach the richness of other visualization tools for ASP, as for example it does not support the creation of animations. On the other hand, ASP Chef is ready to use in the browser, without any additional software, and the *Graph* operation produces interactive graphs with the possibility to search and highlight node and link labels. Moreover, being a side output, the *Graph* operation can be introduced in all points of interest within the same recipe, showing intermediate states of the pipeline.

8. Conclusion

ASP recipes aim at easing the development of broad pipelines involving ASP. Uniformity of input and output format enables arbitrary combinations of operations, actually without limiting too much the data that can be manipulated in the pipeline thanks to the support for Base64 content. ASP Chef provides a ready-to-use framework to practice with ASP recipes, and employs the ASP engine clingo-wasm not only to address combinatorial search and optimization, but also to implement several operations.

The extensible design of ASP Chef leaves space for future work. First of all, there are several other operations that are already implemented in ASP Chef and have not being formalized here.

Some of them possibly require a more sophisticated notion of recipe, as for example *Store* and *Restore* operations, which require a notion of state. Moreover, the need for new operations is expected to arise from different domains, as for example we are experiencing in the definition of cyber ranges for digital twins defined in the *Digital Twin Definition Language* [23].

Finally, even if efficiency of computation is not the main goal of ASP Chef, an empirical assessment is expected in the near future, to better understand the overhead due to running the ASP engine in the browser. We do not expect a too severe overhead, but anyhow we already implemented a *Server* operation that can lighten the computation addressed by the browser.

Acknowledgments

This work was partially supported by Italian Ministry of Research (MUR) under PNRR project FAIR "Future AI Research", CUP H23C22000860006, under PNRR project Tech4You "Technologies for climate change adaptation and quality of life improvement", CUP H23C22000370006, and under PNRR project SERICS "SEcurity and RIghts in the CyberSpace", CUP H73C22000880001; by Italian Ministry of Health (MSAL) under POS project RADIOAMICA, CUP H53C22000650006; by the LAIA lab (part of the SILA labs) and by GNCS-INdAM.

References

[1] M. Balduccini, M. Barborak, D. A. Ferrucci, Pushing the limits of clingo's incremental grounding and solving capabilities in practical applications, Algorithms 16 (2023) 169.
[2] V. Lifschitz, Answer Set Programming, Springer, 2019.
[3] E. Erdem, M. Gelfond, N. Leone, Applications of answer set programming, AI Mag. 37 (2016) 53–68.
[4] M. Alviano, W. Faber, N. Leone, Disjunctive ASP with functions: Decidable queries and effective computation, Theory Pract. Log. Program. 10 (2010) 497–512.
[5] R. Bertolucci, A. Capitanelli, C. Dodaro, N. Leone, M. Maratea, F. Mastrogiovanni, M. Vallati, Manipulation of articulated objects using dual-arm robots via answer set programming, Theory Pract. Log. Program. 21 (2021) 372–401.
[6] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, T. Schaub, ASP-Core-2 input language format, Theory Pract. Log. Program. 20 (2020) 294–309.
[7] M. Gebser, B. Kaufmann, T. Schaub, Conflict-driven answer set solving: From theory to practice, Artif. Intell. 187 (2012) 52–89.
[8] J. H. Maloney, M. Resnick, N. Rusk, B. Silverman, E. Eastmond, The scratch programming language and environment, ACM Trans. Comput. Educ. 10 (2010) 16:1–16:15.
[9] A. Sahay, A. Indamutsa, D. D. Ruscio, A. Pierantonio, Supporting the understanding and comparison of low-code development platforms, in: SEAA, IEEE, 2020, pp. 171–178.
[10] O. Febbraro, K. Reale, F. Ricca, ASPIDE: integrated development environment for answer set programming, in: LPNMR, volume 6645 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 317–330.

[11] P. Busoniu, J. Oetsch, J. Pührer, P. Skocovsky, H. Tompits, Sealion: An eclipse-based IDE for answer-set programming with advanced debugging support, Theory Pract. Log. Program. 13 (2013) 657–673.
[12] F. Calimeri, S. Germano, E. Palermiti, K. Reale, F. Ricca, Developing ASP programs with ASPIDE and LoIDE, Künstliche Intell. 32 (2018) 185–186.
[13] F. Calimeri, G. Ianni, Template programs for disjunctive logic programming: An operational semantics, AI Commun. 19 (2006) 193–206.
[14] S. Costantini, G. D. Gasperis, Dynamic goal decomposition and planning in MAS for highly changing environments, in: CILC, volume 2214 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2018, pp. 40–54.
[15] P. Cabalar, J. Fandinno, Y. Lierler, Modular answer set programming as a formal specification language, Theory Pract. Log. Program. 20 (2020) 767–782.
[16] S. Costantini, G. D. Gasperis, L. D. Lauretis, An application of declarative languages in distributed architectures: ASP and DALI microservices, Int. J. Interact. Multim. Artif. Intell. 6 (2021) 66–78.
[17] P. Cabalar, J. Fandinno, T. Schaub, P. Wanko, On the semantics of hybrid ASP systems based on clingo, Algorithms 16 (2023) 185.
[18] O. Cliffe, M. D. Vos, M. Brain, J. A. Padget, ASPVIZ: declarative visualisation and animation using answer set programming, in: ICLP, volume 5366 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 724–728.
[19] R. Lapauw, I. Dasseville, M. Denecker, Visualising interactive inferences with IDPD3, CoRR abs/1511.00928 (2015).
[20] C. Kloimüllner, J. Oetsch, J. Pührer, H. Tompits, Kara: A system for visualising and visual editing of interpretations for answer-set programs, in: INAP/WLP, volume 7773 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 325–344.
[21] A. Dovier, P. Benoli, M. C. Brocato, L. Dereani, F. Tabacco, Reasoning in high schools: Do it with ASP!, in: CILC, volume 1645 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2016, pp. 205–213.
[22] S. Hahn, O. Sabuncu, T. Schaub, T. Stolzmann, Clingraph: ASP-based visualization, in: LPNMR, volume 13416 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 401–414.
[23] M. Platenius-Mohr, S. Malakuti, S. Grüner, J. Schmitt, T. Goldschmidt, File- and API-based interoperability of digital twins by model transformation: An IIoT case study using asset administration shell, Future Gener. Comput. Syst. 113 (2020) 94–105.