Using py-aspio

January 17, 2017

Outline

Motivation

Answer Set Programming

- Answer Set Programming (ASP) is a declarative programming paradigm [Gelfond and Lifschitz, 1991].
- Applications: *workforce management* [Ricca et al., 2012], *generating holiday plans for tourists* [Ielpa et al., 2009], for many others cf. [Erdem et al., 2016].

Limitations

- Typical end-user applications contain components which cannot be (easily) solved in ASP:
  - graphical user interfaces
  - presentation of results
  - interfaces to data sources
  - etc.
- Realizing such components is in the domain of traditional object-oriented (OOP) languages.

Motivation

Typical approach

- Use ASP programs as components of a larger application.
- The ASP program solves the core computational problem, while other components are implemented in an object-oriented language.
- To this end, object-oriented code
  1. adds input as facts,
  2. evaluates the ASP program, and
  3. interprets the answer sets.
- But: An implementation from scratch is similar for most applications $\Rightarrow$ repetitive work.

Evaluating ASP Programs from Object-Oriented Code

Overview

- We want to use ASP programs similarly to subprocedures:
  an ASP program P performs a computation over input parameters $v_1, \ldots, v_n$,
  and each answer set should correspond to a solution object.

- Approach: the ASP program is annotated with input/output specifications.
  Annotations are added as special comments of form `%!` to the ASP code.

- py-aspio is a Python library for evaluating ("calling") such an annotated program from Python code:
  - it takes an annotated ASP program and a list of input parameters (objects) as input, and
  - returns a set of objects (corresponding the results of the ASP program).

3-Colorability I

To give an overview over the usage of py-aspio, we show an example implementation of the 3-colorability problem:

- Given a graph as input
- assign to each node a color (red, green, or blue), such that
- any two adjacent nodes have different colors.

The graph in the Python code is represented by sets of nodes and edges:

- Nodes are represented by the class Node, where the attribute label is a unique string identifiying the node, and
- edges are represented by the class Edge, where the attributes first and second are the nodes at both ends of the edge.

3-Colorability II

coloring.dl (ASP code and input/output specification)

```
%! INPUT (Set<Node> nodes, Set<Edge> edges) {
%!     node(n.label) for n in nodes;
%!     edge(e.first.label, e.second.label) for e in edges; }
%! OUTPUT {
%!     colored_nodes = set {
%!         query: color(X, C);
%!         content: ColoredNode(X, C); }; }
color(X, red) v color(X, green) v color(X, blue) :- node(X).
:- edge(X, Y), color(X, C), color(Y, C).
```

This program can be called with two parameters of types Set<Node> and Set<Edge> (i.e., any set-like collection containing instances of the classes Node and Edge, respectively).
Its output is of type Set<ColoredNode> (by default, a frozenset containing instances of ColoredNode).

3-Colorability III

The Python program first defines the data-holding classes and registers them with py-aspio.

coloring.py (executable Python program)

```
from collections import namedtuple
import aspio

# Define classes and create sample data
Node = namedtuple('Node', ['label'])
ColoredNode = namedtuple('ColoredNode', ['label', 'color'])
Edge = namedtuple('Edge', ['first', 'second'])
a, b, c = Node('a'), Node('b'), Node('c')
nodes = {a, b, c}
edges = {Edge(a, b), Edge(a, c), Edge(b, c)}

# Register class names with aspio
aspio.register_dict(globals())

# ... (continued on next slide)
```

3-Colorability IV

Then the ASP program is loaded and evaluated in different ways.

coloring.py (cont'd)

```
# Load ASP program and input/output specifications from file
prog = aspio.Program(filename='coloring.dl')

# Iterate over all answer sets
for result in prog.solve(nodes, edges):
    print(result.colored_nodes)

# Shortcut if only one variable is needed (note prefix "each_")
for cns in prog.solve(nodes, edges).each_colored_nodes:
    print(cns)

# Compute a single answer set
result = prog.solve_one(nodes, edges)
if result is not None: print(result.colored_nodes)
else: print('no answer set exists')
```

Further examples

The 3-Colorability example and others can be found in the “examples” directory of the py-aspio repository:
`https://github.com/hexhex/py-aspio/tree/master/examples`

Outline

Input Specification

Consists of two parts:

- The parameter list defines what input is expected from the Python program.
- The predicate specifications control how atoms are generated from the input data.

In the example from before:

```
%!  INPUT (Set<Node> nodes, Set<Edge> edges) {
%!      node(n.label) for n in nodes;
%!      edge(e.first.label, e.second.label) for e in edges;
%!  }
```

Input parameter list

A comma-separated list of parameters, each of the form type followed by name, e.g. `int x` defines a parameter with name `x` of type `int`.

Available types:

int	integers
str	character strings
$Set\langle T\rangle$	set containing elements of type T
$Sequence\langle T\rangle$	list containing elements of type T
$Dictionary\langle K, V\rangle$	dictionary with keys of type K and values of type V
$Tuple\langle T_1, \ldots, T_n\rangle$	n-tuple with components of types $T_1, \ldots, T_n$
MyClass, ...	user-defined classes

In py-aspio, i.e., the Python implementation of the specification language, the actual types are not checked at runtime but serve as documentation.

Predicate specifications

First part defines the form of generated facts. Example:

```
edge(e.first.label, e.second.label, e.someIntArray[3])
```

This will create facts of the form edge("node1", "node2", 27).

Available components:

- ASP predicate
- Variables: input parameter or introduced by loop (next slide)
- Attribute access
- Subscription (currently only for integer subscripts)

How are python objects mapped to ASP terms?

- int : Integer constant
- str : Quoted string constant (special characters are escaped transparently by py-aspio)
- Everything else is converted to str first

Iteration

Second part of predicate specification allows iteration over collections. Example:

$$\texttt{p(...) for } x_1 \texttt{ in } y_1 \texttt{ ... for } x_k \texttt{ in } y_k\texttt{.}$$

Loop variables x_i must be new and refer to elements of collections y_i
(the y_is consist of variables plus attribute access and subscription).

Evaluated from left to right (same as list comprehensions in Python), meaning a variable x_i can be used in all y_j to the right ($y_{i+1}, \ldots, y_k$), allowing for nested iteration.

Value of the loop variables:

- If y_i is a set, x_i will loop over all elements of y_i.
- If y_i is a sequence, x_i will loop over all $(index, element)$ pairs of y_i.
- If y_i is a dictionary, x_i will loop over all $(key, value)$ pairs of y_i.

What do set, sequence, dictionary mean in Python?
$\rightarrow$ Any classes that implement the abstract base classes collections.abc.Set, collections.abc.Sequence, collections.abc.Mapping, respectively!

Shortcuts

- Iterating over a sequence (e.g., a list instance) produces $(index, element)$ pairs $\Rightarrow$ index and element can be accessed with subscripts x[0] and x[1]
- Shortcut: use list of variables $x_1, \ldots, x_n$ instead of single iteration variable x $\Rightarrow$ write `for (i,v) in y` instead of `for x in y` for a sequence y to access components directly
- Analogously for dictionaries and $(key, value)$-pairs.
- This shortcut can also be used for tuples.

Example

```
INPUT (Sequence<Tuple<int, int>> readings) {
    temperature(x[0], x[1][0]) for x in readings;
    humidity(t, hum) for (t,(temp,hum)) in readings;
}
```

The specification of temperature uses subscripts to access the components, while the specification of humidity assigns the values directly to different variables.

Additionally, the underscore _ serves as *don't care* variable to ignore unneeded values.

Outline

Output Specification

Enables access to results of the ASP program:

- Each answer set is mapped to a single result object.
- The output specification defines one or more attributes for the result object by referring to the answer set.

Syntactically, it is just a list of assignments of the form "name = object".

Example

```
%! OUTPUT {
%!     labels = set { query: label(X); content: X; };
%! }
```

Expressions overview

The objects on the right-hand side can be built from the following types of expressions:

- Integer constants, e.g. 0, 123, -5, ...
- String constants, e.g. "hello", "one\ntwo", ...
- Tuples: $(e_1, e_2, \ldots, e_n)$ with subexpressions $e_1, \ldots, e_n$
- Instantiating user-defined classes: *MyClass* $(e_1, e_2, \ldots, e_n)$
- Sets: set { query: q; content: e; }
- Shortcuts for sets: set { p/n } and set { p/n -> *MyClass* }
- Sequences/lists: sequence { query: q; index: i; content: e; }
- Dictionaries: dictionary { query: q; key: k; content: e; }
- Variables (after being introduced by a query): X, Y, Color, ...
- References: &*name*

Collection Expressions I

Collections expressions use a query q to access the answer set.
There are three types:

- set { query: q; content: e; }
- sequence { query: q; index: i; content: e; }
- dictionary { query: q; key: k; content: e; }

The query q is a list of (possibly non-ground) ASP literals that are to be matched against the answer set. Any variables introduced by (non-ground literals in) the query q can be used in the expressions e, i, k.
Every match of the query yields an element of the collection, with any ASP variables introduced in q replaced by their matched values.

- The content e is an expression that defines the elements of the collection.
- In case of a sequence, the index i (must be a variable) determines the position of the element. The values of i over all matches must form a consecutive range of integers without gaps or duplicates.
- In case of a dictionary, the key k determines the key of the element. As e, the key can be any expression.

Note that in Python, set elements and dictionary keys must be hashable.

Collection Expressions II

Example

Consider the following output specification:

```
OUTPUT {
    labels = set { query: vertex(X); content: X; };
    red_nodes = set { query: color(X, red); content: X; };
}
```

It defines two attributes, labels and red_nodes, whose concrete values depend on the current answer set. Now consider the answer set

$$I = \{vertex(a), vertex(b), vertex(c), edge(a,b), edge(a,c),$$
$$color(a, blue), color(b, red), color(c, red)\}.$$

The query vertex(X) matches all atoms of the predicate $vertex$, thus labels will have the value {"a", "b", "c"}.
The query color(X, red) has a constant as second argument, which means only atoms with this value are matched, here: $color(b, red)$ and $color(c, red)$. The value of red_nodes will be {"b", "c"}.

Collection Expressions III

- Since the query refers to elements of the ASP code, the same naming conventions apply: words starting with an upper-case letter are variables, and words starting with a lower-case letter are constant symbols.
- Note that for consistency, all variable values are strings. If an integer value is required, use int(X) instead of X. The only exception is the index clause in sequence expressions, which is always interpreted as an integer.
- As in ordinary ASP, anonymous variables _ can be used in the query.

Collection Expressions IV

Nested collections

Collection expressions can be nested. The following extracts a dictionary that maps colors to the set of nodes with that color:

```
%! OUTPUT {
%!     labels_by_color = dictionary {
%!         query: color(_, C);
%!         key: C;
%!         content: set { query: color(L, C); content: L; };
%!     };
%! }
```

If this expression is evaluated under the answer set

$$\{color(a, blue), color(b, red), color(c, red)\},$$

the dictionary labels_by_color yields the mappings $blue \mapsto \{a\}$ and $red \mapsto \{b, c\}$. Note that the variable C is introduced in the dictionary expression. For every match of its query color(_, C), the nested set expression is evaluated with C fixed to the matched value, thus generating a set of labels colored by the color C.

Shortcuts for Set Expressions

- set { p/n } stands for set { query: $p(X_1, \ldots, X_n)$; content: $(X_1, \ldots, X_n)$; }
- set { p/n –> *MyClass* } stands for
  set { query: $p(X_1, \ldots, X_n)$; content: *MyClass*$(X_1, \ldots, X_n)$; }

Example

```
OUTPUT {
  labels = set { vertex/1 };
  colored_nodes = set { color/2 -> ColoredNode };
}
```

is equivalent to

```
OUTPUT {
  labels = set { query: vertex(X); content: X; };
  colored_nodes = set { query: color(X, Y); content: ColoredNode(X, Y); };
}
```

Outline

Loading the ASP Program

ASP programs are represented by the class Program.
Programs can be loaded from files or strings:

```
import aspio
prog1 = aspio.Program(filename='aspfile.dl')
prog2 = aspio.Program(code=r'''
    a :- not b.
    b :- not a.
''')
prog3 = aspio.Program()   # empty program
```

To merge ASP code from multiple sources:

```
prog.append_file('anotherfile.dl')
prog.append_code('c :- b.')
```

The input and output specifications are extracted automatically. However, note that one program may contain at most one input specification and at most one output specification.

Evaluating the ASP Program

Two methods for evaluating a program p:

- prog.solve(...): Returns an iterable allowing to iterate over all answer sets of the program. Note that due to the semantics of ASP, the concrete order of the returned objects has no meaning!
- prog.solve_one(...): Returns the first answer set found by the solver, or None if the program does not have any answer sets.

The positional arguments of these methods are the parameters defined by the input specification.
The answer sets are represented by instances of the class Result. These objects have attributes as defined by the output specification (see next slide).

Additional keyword arguments:

- options: specify additional options to pass to the solver (more details later).
- solver: specify the ASP solver to use. Currently, only dlvhex is supported.

Working with the Output of an ASP Program

Consider the annotations of the 3-Colorability example (with parts omitted):

```
%! INPUT (Set<Node> nodes, Set<Edge> edges) { ... }
%! OUTPUT { colored_nodes = set { ... }; }
```

According to the input specification, the evaluation requires two arguments.

- Use the method solve to iterate over all answer sets. The collection returned by solve contains one object per answer set, with attributes as defined in the output specification:

  ```
  for result in prog.solve(nodes, edges):
      print(result.colored_nodes)
  ```

  A shortcut if only one attribute is needed (note the required prefix each_):

  ```
  for cns in prog.solve(nodes, edges).each_colored_nodes:
      print(cns)
  ```

- Use the method solve_one if only a single arbitrary answer set is needed, or if only the consistency of the program is of interest:

  ```
  result = prog.solve_one(nodes, edges)
  if result is not None: print(result.colored_nodes)
  else: print("no answer set exists")
  ```

Solver Options

Instance of class SolverOptions with the following attributes:

- max_answer_sets: Set to an integer n to compute at most n answer sets, or to None to compute all (default: None).
- max_int: Set maximum integer value (may be necessary for arithmetic).
- capture: A list of strings; the names of predicates that should be captured (in addition to the output specification). The captured data is available through the attribute answer_set of the class Result (cf. below example).
- custom: A list of strings; any custom options that are to be passed to the solver subprocess.

Example

The following snippet show how to evaluate an ASP program with the max_int option set to 25, and accessing the raw data of predicate p:

```
prog = aspio.Program(...)
opts = aspio.SolverOptions(max_int=25, capture=['p'])
for result in prog.solve(input1, input2, options=opts):
    ps = result.answer_set['p']  # type: Iterable[Tuple[str, ...]]
```

User-defined Classes in Output

User-defined classes that are to be used in an output specification, either

- must be fully qualified (e.g., package.module.MyClass), or
- have to be registered with py-aspio as described below.

There are three methods to register classes for a program prog:

- Pass the constructor and its name to the method register:

```
prog.register(MyClass, 'MyClass')
```

- If the name is missing, the constructor's attribute __name__ is used:

```
prog.register(MyClass)
```

- Or simply register all callable objects in the current global scope:

```
prog.register_dict(globals())
```

Alternatively, these methods may be called on the aspio module in order to register classes globally (i.e., for all ASP programs in the current application):

```
import aspio
aspio.register(MyClass)
```

References I

Erdem, E., Gelfond, M., and Leone, N. (2016).

Applications of answer set programming.

AI Magazine, 37(3):53–68.

Gelfond, M. and Lifschitz, V. (1991).

Classical Negation in Logic Programs and Disjunctive Databases.

9(3–4):365–386.

Ielpa, S. M., Iiritano, S., Leone, N., and Ricca, F. (2009).

An ASP-Based System for e-Tourism.

In Erdem, E., Lin, F., and Schaub, T., editors, *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *Lecture Notes in Computer Science*, pages 368–381. Springer.

Ricca, F., Grasso, G., Alviano, M., Manna, M., Lio, V., Iiritano, S., and Leone, N. (2012).

Team-building with answer set programming in the Gioia-Tauro seaport.

TPLP, 12(3):361–381.