The A-POL System

Mauricio Osorio[1] and Enrique Corona[1]

Universidad de las Américas, CENTIA.
Sta. Catarina Mártir, Cholula, Puebla,
72820 México
{josorio,is108990}@mail.udlap.mx

Abstract. Answer Set Programming (ASP) is a formalism widely used for knowledge representation since its introduction in 1988 by Gelfond and Lifschitz [4]. Nowadays there are powerful implementations of this paradigm, like DLV[1] and Smodels[2]. In order to increment the descriptive power of this tools, several extensions to their languages have been done. For example, the weak constraints of Smodels and more recently the aggregate functions of DLV. Partial Order Programming is very similar to mathematic programming, where a function is minimized (or maximized) and has a set of restrictions, the difference is that the domain of values is a partial order [12]. Theorical work has already been done to integrate this paradigms [11], one of the most important results is that partial-order clauses can be expressed as normal clauses [8]. The main purpose of this work is to present A-POL[3], an extension for ASP that allow us to express optimization problems in a very suitable way, integrating disjunctive clauses and partial-order clauses.

1 Introduction

Partial-order clauses are introduced and studied in [6, 9, 10], we refer the reader to this papers for a full account of the paradigm. In comparision with traditional equational clauses for defining functions, partial-order clauses offer better support for defining recursive aggregate operations. Partial-order clauses are actually a generalization of subset program clauses [2, 5]. There are two basic forms of a partial-order clause [10]:

$f(terms) \geq expression$
$f(terms) \leq expression$

Since these clauses are used to define functions, we requiere that each variable ocurring in $expression$ should also occur in $terms$. Terms are made up of constants and variables, while expressions are in addition made up of user-defined-functions. Informally, the declarative meaning of a partial-order clause is that, for all its ground instances, the function f applied to the argument $terms$ is $\geq$ (respectively, $\leq$) than the ground term denoted by the $expression$ on the right-hand side. In general, multiple partial-order

[1] http://www.dbai.tuwien.ac.at/proj/dlv/
[2] http://www.tcs.hut.fi/Software/smodels/
[3] http://www.udlap.mx/~is108990/apol

clauses may be used in defining some function f. We define the meaning of a ground expression $f(terms)$ to be the `least-upper bound` (respectively, `greatest-lower bound`) of the resulting terms defined by the different partial-order clauses for f. In practice the lattice domains that commonly occur in applications are sets (under the subset ordering) and numbers (under the numeric ordering). In the former case, the `lub` and `glb` operations are set union and intersection, respectively; and in the latter case, these operations are numeric greater-than and less-than, respectively. All these operations can be implemented quite efficiently, as shown in [5]. To use information from an extensional database in partial-order clauses we need to extend them as follows:

$$f(terms) \geq expression :- p_1(terms_1), \ldots, p_n(terms_n).$$
$$f(terms) \leq expression :- p_1(terms_1), \ldots, p_n(terms_n).$$

where p_i $(1 \leq i \leq n)$ is a predicate symbol and the variables in $terms_i$ may appear in $terms$ or $expression$.

Example 1. [10] Suppose that a graph is defined by a predicate $edge(X, Y, C)$, where C is the non-negative distance associated with a directed edge from node X to node Y, the shortest distance from X to Y can be declaratively specified through the following partial-order clauses:

$$short(X, Y) \leq C :- edge(X, Y, C).$$
$$short(X, Y) \leq short(X, Z) + short(Z, Y) :- edge(X, Z, C).$$

The meaning of a ground expression such as $short(a, b)$ is the `glb` (smallest number in the above example) of the results defined by the different partial-order clauses. In order to have a well-defined function using partial-order clauses, whenever a function is circularly defined (as could happen in the above example when the underlying graph is cyclic), it is necessary that the constituent functions be monotonic.

Example 2. The 0-1 Knapsack Problem [14] is a well-known optimization problem that is known to be NP-complete. Suppose we are given weights w_i and profits p_i, for $1 \leq i \leq n$, and a capacity m. For $0 \leq M \leq m$, and $1 \leq I \leq n$, define $ks(I, M)$ to be the profit of the optimal solution to the 0-1 knapsack problem, using objects $1, ..., I$, and knapsack capacity M. Then, ks is defined by the following inequalities:

$$ks(0, M) \geq 0.$$
$$ks(I, M) \geq ks(I-1, M) :- I \geq 1.$$
$$ks(I, M) \geq ks(I-1, M - w(I)) + p(I) :- I \geq 1, w(I) \leq M.$$

The solution for this problem (maximum profit) is the value of $ks(n, m)$.

Example 3. This is a problem from the ACM International Collegiate Programming Contest Problem Set Archive[4].

The fastfood chain McBurger owns several restaurants at highway. Recently, they have decided to build several depots along the highway, each one located at a restaurant and supplyng several

[4] `http://www.acm.inf.ethz.ch/ProblemSetArchive.html`

of the restaurants with the needed ingredients. Naturally, these depots should be placed so that the average distance between a restaurant and its assigned depot is minimized. You are to write a program that computes the optimal positions and assigments of the depots.

To make this more precise, the management of McBurger has issued the following specification: You will be given the positions of n restaurants along the highway as n integers $d_1 < d_2 < \ldots < d_n$ these are the distances measured from the company's headquarter, which happends to be at the same highway). Furthermore, a number k ($k \leq n$) will be given, the number of depots to be built.

The k depots will be built at the locations of k different restaurants. Each restaurant will be assigned to the closest depot, from which it will then receive its supplies. To minimize shipping costs, the total distance sum, defined as

$$\sum_{i=1}^{n} |d_i - \textit{(position of depot serving restaurant i)}|$$

must be as small as possible. Write a program that computes the positions of the k depots, such that the total distance sum is minimized.

A recursive solution to obtain the *total distance sum* is as follows:

$$best(nd, nr) = \begin{cases} cost(1, nr) & , nd = 1 \\ min_{i=1}^{nr} \begin{cases} cost(i, nr) + \\ \quad best(nd-1, i-1) \end{cases} & , nd < nr \end{cases}$$

Where nr is the number of restaurants, nd is the number of depots and $cost(i, j)$ represents the cost of placing a depot between restaurants i and j. A dynamic programming approach to solve this problem can be derived from this formula, and the location of the depots can be recovered from the resulting matrix. This kind of problems can be easily formulated with partial-order clauses:

$$\begin{aligned} &best(I, I) \leq 0. \,/\, location(I). \\ &best(1, I) \leq cost(1, I) :- middle(1, I) = M. \,/\, location(M). \\ &best(I, J) \leq cost(B, J) + best(I-1, B-1) :- \\ &\qquad I < J, I \leq B, B \leq J, middle(B, J) = M. \,/\, location(M). \end{aligned}$$

Where $middle(X, Y)$ is the arithmetic mean of X and Y. In this example we introduce a very useful feature of A-POL, *witness recovery*. This is not a standard feature of partial order programming, however, it is very useful for optimization problems. In this particular example, recovering the witness of $best(k, n)$ we obtain the ubication of the depots in the atoms of type $location$.

The rest of this work is organized as follows. First we present the language of A-POL, then we explain how our current implementation works and finally we provide some examples and experimental results.

2 Proposed Language

Our language includes function symbols, predicate symbols, constants and variables, it can be described with the following context free grammar:

```
program ::= clause | clause program
clause ::= disjunctive_clause | partial_order_clause
term ::= variable | constant
terms ::= terms | term, terms
expression ::= term | f(expressions)
expressions ::= expression | expression, expressions
f-p_atom_1 ::= function_atom | predicate_atom
f-p_atom_2 ::= f-p_atom_1 | not predicate_atom
function_atom ::= f(terms) = term
predicate_atom ::= p(terms)
inequality_atom ::= f(terms) ≤ expression | f(terms) ≥ expression
f-p_predicates_1 ::= f-p_atom_1 | f-p_atom_1, f-p_predicates_1
f-p_predicates_2 ::= f-p_atom_2 | f-p_atom_2, f-p_predicates_2
partial_order_clause ::= inequality_atom :- f-p_predicates_1
disjunctive_atoms ::= predicate_atom | predicate_atom ∨ disjunctive_atoms
disjunctive_head ::= disjunctive_atoms | ε
disjunctive_clause ::= disjunctive_head :- f-p_predicates_2
```

Where `f` is any function symbol and `p` is any predicate symbol. A *f-p* symbol is either a function or predicate symbol.

Definition 1. *A program P is a couple $\langle DA, PO\rangle$ where DA is a set of disjunctive clauses and PO is a set of partial-order clauses.*

We are integrating disjunctive clauses with partial order clauses in our language. Since their evaluation is done in different ways, we need to setup an order of evaluation.

Definition 2. *A program P is stratified if exists a mapping function $level_{sp}$: $F \cup Pr \to \mathcal{N}$, where $\mathcal{N}$ is a subset of consecutive natural numbers, F is the set of user defined function symbols in P and Pr is the set of predicate symbols in P. Clauses of P must have the following properties:*

(i) *For a clause of the form:*

$$f(terms) \leq term :- B.$$
$$f(terms) \geq term :- B.$$

Where B is a set of f-p *atoms, then $level_{sp}(f) \geq level_{sp}(p)$, where p is any* f-p *symbol appearing in B.*

(ii) *For a clause of the form:*

$$f(terms) \leq g(expr) :- B.$$
$$f(terms) \geq g(expr) :- B.$$

Where f and g are user defined functions and B is defined like in (i), then $level_{sp}(f) \geq level_{sp}(g)$, $level_{sp}(f) > level_{sp}(h), level_{sp}(p)$, $level_{sp}(g) > level_{sp}(p)$ where p is any predicate symbol appearing in B and h is any user defined function symbol in $expr$.

(iii) *For a clause of the form:*

$$f(terms) \leq m(g(expr)) :- B.$$
$$f(terms) \geq m(g(expr)) :- B.$$

Where B is defined like in (i) and m is a monotonic function, then $level_{sp}(f) \geq level_{sp}(g)$, $level_{sp}(f) > level_{sp}(m), level_{sp}(h), level_{sp}(p)$ where p is any predicate symbol appearing in B and h is any function symbol in $expr$.

(iv) *For a clause of the form:*

$$p_1(term_1) \vee \ldots \vee p_n(term_n) \ :- \ B$$

Where B is defined like in (i), then for each i ($1 \leq i \leq n$), $level_{sp}(p_i) > level_{sp}(q)$, where q is any f-p *symbol appearing in B.*

(v) *No other kind of clause is allowed.*

For a stratified program P the function $level_{sp}$ is obtained using the dependency graph of the *f-p* symbols defined in P.

Definition 3. *For a stratified program P we define the function $level_{sp}: F \cup Pr \rightarrow \mathcal{N}$ using the dependency graph $G_{dep}(V, E)$ as follows:*

$$level_{sp}(\{y \mid \nexists (x, y) \in E, \ \forall x \in V\}) = 1$$
$$level_{sp}(\{y \mid level_{sp}(x) = n, \ \exists (x, y) \in E\}) = n + 1$$

A restriction in our language is that predicate and function symbols cannot be in the same level, now that we have the function $level_{sp}$ we define the function $level$ to deal with this issue.

Definition 4. *Using $level_{sp}^{-1}$ we define the function $level: 0 \cup \mathcal{N} \rightarrow \mathcal{P}(F \cup Pr)$ as follows:*

$$level(2n-2) = \{x \mid x \in level_{sp}^{-1}(n) \wedge x \in Pr\}$$
$$level(2n-1) = \{x \mid x \in level_{sp}^{-1}(n) \wedge x \in F\}$$

With this construction, all the predicate symbols in P are in even levels (including zero) and all the functional symbols in P are in odd levels. The evaluation order of a stratified program P is obtained using $level$.

3 Implementation

A-POL is written in Java and uses a compiled version of DLV [3]. Our implementation follows the proposed language with a small extension for partial-order clauses that allow us to recover the witness in optimization problems. To obtain the fix-point of partial-order clauses we use a general form of dynamic programming as shown in [7].

3.1 Main Computation Cycle

Now we present the main algorithm of A-POL for computing the models of a stratified program P.

```
A-POL(P) {
    order = obtainOrder(P)
    models = ∅
    i = 0
```

```
    while (order_i ≠ φ or order_{i+1} ≠ φ) {
        if (i mod 2 = 0)
            models = ASP(models,order_i)
        else
            models = POP(models,order_i)
        i = i + 1
    }
    return models
}
```

where `obtainOrder` generates the dependency graph of the *f-p* symbols in P and obtains its evaluation order using the function $level$. `obtainOrder` returns the array $order$, where $order_i$ contains the clauses corresponding to the *f-p* symbols in $level(i)$. `ASP` obtains the stable models of disjunctive clauses in $order_i$ and `POP` obtains the models of the partial order clauses in $order_i$, both using each $model \in$ `models` as an extensional database.

3.2 DLV Wrapper

A-POL uses DLV to obtain the stable models from disjunctive clauses. DLV is used as a black box and the comunication between them is done directly from Java, this means that an input file is generated for DLV and its output models are parsed and wrapped so they can be used as an extensional database for the next level in the stratification path. To generate DLV's input file, disjunctive clauses of the form:

$$p_1(t_1) \vee \ldots \vee p_n(t_n) :- b_1(t_{n+1}), \ldots, b_m(t_{n+m}), f_1(t_{n+m+1}) = r_1, \ldots, f_s(t_{n+m+s}) = r_s.$$

are rewritten as:

$$p_1(t_1) \vee \ldots \vee p_n(t_n) :- b_1(t_{n+1}), \ldots, b_m(t_{n+m}), f_1(t_{n+m+1}, r_1), \ldots, f_s(t_{n+m+s}, r_s).$$

In our implementation only a subset of the DLV language is supported, to support other features (such as weak constraints and aggregate functions) we would have to extend the grammar of A-POL (in other words, extend the grammar of the source-input and DLV-output parsers).

3.3 Partial-order clauses compiler

A-POL is a complete partial-order clauses compiler that supports by default two lattice domains, numbers under the numeric ordering and sets under the subset ordering. As mentioned before, A-POL uses a general form of dynamic programming, because of this, witness recovery is done very natural way. The main algorithm used for computing partial-order clauses is described below.

```
POP(model,clauses) {
    clauses' = flattening(clauses)
    grounded = grounding(model,clauses')
    return fix-point(grounded)
```

}

Where `flattening` returns the flattened form of the partial-order clauses. For example, consider the clause:

$$p(X) \leq q(r(s(Y,Z))) :- t(W).$$

its flattened form is:

$$p(X) \leq v_3 :- t(W), s(Y,Z) = v_1, r(v_1) = v_2, q(v_2) = v3.$$

This is done to eliminate composed functions in the right-hand side of the inequality atom of partial-order clauses. The instantiation of `clauses` using `model` as an extensional database is done by `grounding`, this is the most expensive part of `POP` in memory (and time) because we need to store all the instanced clauses, we actually store an encoded version of this clauses for smaller memory requirements. To ground a $clause \in$ `clauses` of the form:

$$A_1 \leq v_1 :- A_2, \ldots A_m, A_{m+1} = v_n, A_{m+2} = v_{n-1}, \ldots, A_n = v_1.$$

a *grounding tree* for this $clause$ is generated as follows:

$$Ground(clause, A_x) = \sum_{j=1}^{|U(A_x)|} Ground(instance(U(A_x)_j, clause), A_{x+1})$$

where

(i) $U(A_x)$ is the set of grounded atoms of the same type of A_x that unify with A_x. For example, if we have the grounded atoms $\{p(a,b), p(a,c), p(b,c)\}$ and the partially grounded atom $p(a,X)$, then $U(p(a,X)) = \{p(a,b), p(a,c)\}$ (for A_1 the domain of the function is used to obtain $U(A_1)$).

(ii) $instance(U(A_x)_j, clause)$ replaces the variables of A_x with the values of $U(A_x)_j$ in all the *f-p* atoms of $clause$. For example, for $p(a,X)$ with $U(p(a,X)) = \{p(a,b)\}$, X will be replaced with b in all the *f-p* atoms of $clause$.

A-POL supports user defined libraries, this feature will be explained with more detail later, however, it is very important for grounding because it allow us to make an important cut in the grounding tree of each $clause$. Suppose that we have a predefined *boolean function* called $le(X,Y)$ that returns true if $X \leq Y$ and false otherwise, when X and Y are instanced we can evaluate $le(X,Y)$, if its false we eliminate $clause$ from the tree, if its true we eliminate $le(X,Y)$ from $clause$ and continue with the instantiation of its branches.

The main algorithm to obtain the fix-point of a set of partial-order clauses is described in [7], to ilustrate this procedure we will use example 1 (shortest distance between two nodes of a graph) with the following extensional database:

$$EDB = \{edge(a,b,1), edge(b,c,2), edge(a,d,4)\}.$$

First consider that A-POL will obtain $level(0) = \{edge, +\}$ and $level(1) = \{short\}$,the solution of the first level is trivial. To solve $short$ first we ground its clauses using EDB, obtaining:

```
short(a,b) ≤ 1.
short(b,c) ≤ 2.
short(a,d) ≤ 4.
short(a,a) ≤ v1 :- short(a,b)=v2, short(b,a)=v3, v2+v3=v1.
short(a,b) ≤ v1 :- short(a,b)=v2, short(b,b)=v3, v2+v3=v1.
short(a,c) ≤ v1 :- short(a,b)=v2, short(b,c)=v3, v2+v3=v1.
short(a,d) ≤ v1 :- short(a,b)=v2, short(b,d)=v3, v2+v3=v1.
short(b,a) ≤ v1 :- short(b,c)=v2, short(c,a)=v3, v2+v3=v1.
short(b,b) ≤ v1 :- short(b,c)=v2, short(c,b)=v3, v2+v3=v1.
short(b,c) ≤ v1 :- short(b,c)=v2, short(c,c)=v3, v2+v3=v1.
short(b,d) ≤ v1 :- short(b,c)=v2, short(c,d)=v3, v2+v3=v1.
short(a,a) ≤ v1 :- short(a,d)=v2, short(d,a)=v3, v2+v3=v1.
short(a,b) ≤ v1 :- short(a,d)=v2, short(d,b)=v3, v2+v3=v1.
short(a,c) ≤ v1 :- short(a,d)=v2, short(d,c)=v3, v2+v3=v1.
short(a,d) ≤ v1 :- short(a,d)=v2, short(d,d)=v3, v2+v3=v1.
```

The solution of this program is done using a matrix to store the partial values of the function. Then, we refine them using different iterations until we obtain a fix-point. The initial state S_1 of the matrix stores the values obtained by the EDB of our program:

```
short(a,b) ≤ 1.
short(a,d) ≤ 4.
short(b,c) ≤ 2.
```

	a	b	c	d
a	⊤	1	⊤	4
b	⊤	⊤	2	⊤
c	⊤	⊤	⊤	⊤
d	⊤	⊤	⊤	⊤

In the next interation, we compute state S_2. For this, we no longer use the EDB but only the IDB and S_1.

```
short(a,c) ≤ v1 :- short(a,b) + short(b,c) = v1.
```

	a	b	c	d
a	⊤	1	3	4
b	⊤	⊤	2	⊤
c	⊤	⊤	⊤	⊤
d	⊤	⊤	⊤	⊤

The next interation S_3 is as follows:

```
short(a,c) ≤ v1 :- short(a,b) + short(b,c) = v1.
```

	a	b	c	d
a	⊤	1	3	4
b	⊤	⊤	2	⊤
c	⊤	⊤	⊤	⊤
d	⊤	⊤	⊤	⊤

At this point, $S_2 = S_3$ then S_3 is our fix-point. A-POL stores also a *parent matrix*, if the value of the matrix at $\bar{X}$ is refined using the grounded clause cl the body of this clause is stored in the *parent matrix* at $\bar{X}$, using this information we can recover how a specific position was obtained, doing this in our example, we can obtain the shortest path instead of just the shortest distance between two nodes of a graph. This is the final *parent matrix* of our example:

	a	b	c	d
a		short(a,b)=1	short(a,b)=1,short(b,c)=3	short(a,d)=4
b			short(b,c)=2	
c				
d				

Suppose that we want to know the shortest path between a and c, starting from position (a, c) in the *parent matrix* we have to go to (a, b) and (b, c), since this positions do not give us more information, we are done. With a larger graph we probably would have to track more positions in the *parent matrix*, however, this simple example give us a good idea of how A-POL recovers a witness in this kind of problems.

3.4 Advanced Features

A-POL has some advanced features that allow us to increment its efficiency and descriptive power. This features are, witness recovery, user defined libraries and user defined partial-orders.

Witness recovery The inner procedure for recovering a witness was explained above, however, it is necessary to tell A-POL what information we want to retrieve from each clause, to do this, we extend the syntax of partial-order clauses as follows:

$$f(terms) \geq expression :- p_1(terms_1), \ldots, p_n(terms_n). \ / \ info(iterms).$$
$$f(terms) \leq expression :- p_1(terms_1), \ldots, p_n(terms_n). \ / \ info(iterms).$$

To ilustrate this let us use our shortest distance example (1), suppose that we want to recover the shortest path instead of just the distance, so, we want to obtain every existing edge from our clauses.

$$short(X, Y) \leq C :- edge(X, Y, C). \ / \ way(X, Y).$$
$$short(X, Y) \leq short(X, Z) + short(Z, Y) :- edge(X, Z, C).$$

Notice that nothing is added to the second clause, this is because we are only interested in the existing edges, and the second clause finds a cheaper way between two nodes using intermediate nodes. Now we need to specify which way we want to recover, suppose that we want to know the shortest path between node a and c, we can do this in A-POL as follows:

$$select\ way\ where\ short(a, c).$$

In our resulting model we can recover the path from the atoms of type way. Notice that in this particular implementation we can only obtain one path even if different paths with the same cost exists, now we present an alternative implementation of this problem to fix this issue:

$$short(X) \leq 0 :- start(X).$$
$$short(X) \leq short(Y) + C :- edge(X, Z, C). \ / \ way(Y, X).$$

Supose that our extensional database includes $start(a)$, then $short(X)$ represents the shortest distance from node a to node X.

User defined libraries User defined libraries are not just a fancy implementation issue, but a very convenient way of increasing the core functions of A-POL. Since A-POL is written in Java, loading external classes for their use in a specific program is a very easy task (like dynamic linking libraries). All predefined A-POL functions (add, sub, mul, div, etc..) are implemented as user defined libraries. For creating a new predefined function the user has to extend a Library class, override its functional method (execute), compile it and register it. For functions that are very easy to define in procedural languages most of the time is easier (and more efficient) to write and register a new library than try to code it in A-POL.

User defined partial-orders The co-domain of the functions defined by partial-order clauses is a partial or complete ordered set. In A-POL a specific partial-order can be described from the relations between its elements. To ilustrate this let us consider the set $A = \{a, b, c, d, e\}$ where $a < b < c < d < e$, the relations between their elements are $R = \{a < b, b < c, c < d, d < e\}$.

4 Examples and Results

In this section we present some examples coded in A-POL and some experimental results.

Example 4. A-POL's implementation of the shortest path between two nodes of a graph is as follows:

```
declare < short(node).
select way where short(X) <- end(X).
% extensional database
node(a). node(b). node(c). node(d).
edge(a,b,1). edge(b,c,2). edge(a,d,4).
start(a). end(f).
% rules
short(X)<= 0 :- start(X).
short(X)<= add(short(Y),C) :- edge(Y,X,C). / way(Y,X).
```

The first line is required to specify the domain and the type of partial-order clauses that define the function `short`. We also wrote a DLV version of this problem using weak constraints:

```
% rules
way(X,Y,C) v other(X,Y,C) :- edge(X,Y,C).
:- edge(X,A,C), start(A), way(X, A,C).
:- edge(D,X,C), end(D), way(D,X,C).
:- edge(X,Y,C), edge(X,Y1,C1), way(X,Y,C), way(X,Y1,C1), Y != Y1.
:- edge(X,Y,C), edge(X1,Y,C1), way(X,Y,C), way(X1,Y,C1), X != X1.
r(X) :- start(X).
r(X) :- r(Y), edge(Y,X,C), way(Y,X,C).
:- end(D), not r(D).
```

```
:~ way(X,Y,C). [C:1]
```

Now we present experimental results using complete graphs for both implementations:

# nodes	*DLV*	*A-POL*
10	$0.16s$	$0.65s$
20	$4.77s$	$0.84s$
40	$4m\ 21.45s$	$1.15s$
60	$22m\ 57.02s$	$2.21s$
80	–	$2.85s$

Example 5. Partial-order clauses can be used for carrying out sophisticated flow-analysis computations, as illustrated by the following program which computes the *reaching definitions* and *busy expressions* in a program flow graph. This information is computed by a compiler during its optimization phase [1]. The original formulation of the flow-analysis equations is:

$$out(X) = gen(X) \cup (in(X) - kill(X))$$
$$in(X) = \bigcup_P out(P)$$

Where P is a predecesor of X. Now we present an A-POL implementation of this problem (this particular example is from [1]):

```
setmode.
declare > out(block).
declare > in(block).
declare > gen(block).
declare > kill(block).
% extensional database
block(1..4).
pred(2,1). pred(2,3). pred(2,4).
pred(3,2). pred(4,2). pred(4,3).
gen(1)>= {a,b,c}. gen(2)>= {d,e}.
gen(3)>= {f}.    gen(4)>= {g}.
kill(1)>= {d,e,f,g}. kill(2)>= {a,b,g}.
kill(3)>= {c}.      kill(4)>= {a,d}.
% rules
in(X)>= out(Y) :- pred(X,Y).
out(X)>= gen(X).
out(X)>= sub(in(X),kill(X)).
```

In the above program, `kill(X)` and `gen(X)`, are predefined set-valued functions specifying the relevant information for a given program flow graph and a basic block `X`. `pred(X,Y)`, defines when `Y` *is predecessor of* `X`. The first line specify that the co-domain of the functions defined by partial-order clauses are sets. The set-difference (`sub`) is monotonic in its first argument, and hence the program has a unique intended meaning as it is shown in [1]. The operational semantics behaves exactly as the algorithm proposed in [1] to solve this problem.

Example 6. A-POL's implementation of the 0-1 knapsack is as follows:

```
maxint = 20.
declare > ks(object,tweight).
declare > profit(object).
select take1 where ks(4,10).
% extensional database
object(0..4).
tweight(0..10).
weight(1,2). weight(2,3). weight(3,5). weight(4,8).
profit(0)>= 0. profit(1)>= 1. profit(2)>= 2.
profit(3)>= 3. profit(4)>= 5.
% rules
ks(N,K)>= ks(sub(N,1),K) :- le(1,N). / take1(N,K).
ks(N,K)>= add(profit(N),ks(sub(N,1),sub(K,X))) :-
        weight(N,X), le(X,K), le(1,N). / take1(N,K).
take(X1,Y) :- take1(X1,Y), not take(X2,Y), X2 > X1, object(X2), tweight(Y).
```

In this example we have 4 objects, each `object` has a `weight` and a `profit`, notice that we recover which objects are selected from partial-order clauses and we use disjunctive clauses to make more clear the output.

Example 7. The optimal parentization for a matrix-chain multiplication is a typical dynamic programming problem [14] we present a solution to this problem in A-POL:

```
declare < r(nmat).
declare < c(nmat).
declare < m(nmat,nmat).
select asoc where m(1,6).
% extensional database
nmat(1..6).
r(1)<= 30. c(1)<= 35.
r(2)<= 35. c(2)<= 15.
r(3)<= 15. c(3)<= 5.
r(4)<= 5. c(4)<= 10.
r(5)<= 10. c(5)<= 20.
r(6)<= 20. c(6)<= 25.
% rules
m(N,N)<= 0 :- le(1,N). / asoc(N,N).
m(I,N)<= add(add(m(I,K),m(add(K,1),N)),mul(mul(r(I),c(K)),c(N)))
  :-
       le(1,N), le(1,I), lt(K,N), le(I,K), nmat(K). / asoc(I,N).
```

Where `r(X)` and `c(X)` are the number of rows and columns of matrix `X` respectively. This problem can also be found in the ACM International Collegiate Programming Contest problem set archive. The following results are using the judge's input (consider that in a real contest this inputs have to be processed by the program in less than a minute).

# matrix	*time*	*# grounded clauses*	*# iterations*
3	$0.81s$	13	17
6	$0.86s$	53	123
10	$1.18s$	195	1515
20	$4.24s$	1390	12030
40	$23.27s$	10780	106720

Example 8. The implementation of the fastfood example (3) is as follows:

```
declare < middle(dom,dom).
declare < rest(dom).
declare < sumato(dom,dom,dom).
declare < cost(dom,dom).
declare < best(dom,dom).
declare < rdist(dom,dom).
select location where best(5,10).
% extensional database
dom(1..10).
rest(1)<= 11. rest(2)<= 28.
rest(3)<= 68. rest(4)<= 84.
rest(5)<= 86. rest(6)<= 93.
rest(7)<= 96. rest(8)<= 160.
rest(9)<= 171. rest(10)<= 200.
% rules
middle(X,Y)<= div(add(X,Y),2).
rdist(I,J)<= dist(rest(I),rest(J)).
cost(I,J)<= sumato(I,J,middle(I,J)).
sumato(I,I,K)<= rdist(I,K).
sumato(I,J,K)<= add(rdist(I,K),sumato(add(I,1),J,K)) :- lt(I,J).
best(I,I)<= 0. / location(I).
best(1,I)<= cost(1,I) :- middle(1,I) = M. / location(M).
best(I,J)<= add(cost(B,J),best(sub(I,1),sub(B,1))) :-
          middle(B,J) = M, lt(I,J),
          le(I,B), le(B,J), dom(B). / location(M).
```

Where `rest` represents the distance of the restaurant and `cost(X,Y)` is the cost of placing a depot between restaurants `X` and `Y`. Now we present experimental results using some of the judge's inputs.

# rest.	*time*	*# grounded clauses*	*# iterations*
10	$1.7s$	1090	6460
25	$12.3s$	12975	212150
50	$3m7.6s$	93450	3244300

In the above examples, we use predefined functions (which are implemented as user defined libraries) such as:

Function	Meaning
`add(X,Y)`	X+Y
`sub(X,Y)`	X−Y
`mul(X,Y)`	X∗Y
`div(X,Y)`	X/Y
`dist(X,Y)`	|X-Y|
`le(X,Y)`	$X \leq Y$
`lt(X,Y)`	$X < Y$
`ge(X,Y)`	$X \geq Y$
`gt(X,Y)`	$X > Y$

If a new function is required, the user can easily code it and register it for its use.

5 Conclusions and future work

Partial-order clauses seem to be a better approach for optimization problems than disjunctive clauses. They offer a more direct way to express optimization functions and we have found domains where they are more efficient than disjunctive clauses. Since partial-order clauses can be expressed as normal clauses [8], they can be seen as a very efficient macro for certain kind of Answer Set programs. A-POL's main design offers some important features such as witness recovery and user defined functions (which also allow us to make cuts in the grounding tree). This features make A-POL a more flexible and powerful tool.

A very important issue is that A-POL has to support all the features of DLV, our current version only supports a subset of the DLV language. As mentioned before, A-POL is written in Java. This was a good choice to create a prototype in a short period of time. However, if a better integration with DLV is desired, A-POL should be rewritten in C++. The use of partial-order clauses in other front ends such as DLV-K has not been considered at this point, we believe this could be very useful mainly for efficiency reasons. Nowadays parallel answer set systems are been developed [13]. Our partial-order clauses compiler could use parallel programming techniques (mainly in the grounding) and considering that all the programs in A-POL are stratified, consecutive levels could be evaluated in parallel.

References

1. Alfred V. Aho, Ravi Setvi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison Wesley, 1988.
2. Bharat Jayaraman. Implementation of Subset-Equational Programs. *Journal of Logic Programming*, 11(4):299–324, 1992.
3. Simona Citrigno, Thomas Eiter, Wolfgang Faber, Georg Gottlob, Christoph Koch, Nicola Leone, Cristinel Mateis, Gerald Pfei fer, and Francesco Scarcello. The `dlv` System: Model Generator and Application Frontends. In *Proceedings of the 12th Workshop on Logic Programming (WLP '9 7), Research Report PMS-FB10*, pages 128–137, München, Germany, September 1997. LMU München.
4. Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In R. Kowalski and K. Bowen, editors, *5th Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.

5. Bharat Jayaraman and K. Moon. Subset logic programs and their implementation. *Journal of Logic Programming*, 41(2):71–110, 2000.
6. Bharat Jayaraman, Mauricio Osorio, and K. Moon. Partial order programming (revisited). In M. Nivat V.S. Alagar, editor, *Proc. AMAST*, LNCS 936, pages 561–575. Springer-Verlag, 1995.
7. Mauricio Osorio. Semantics of partial-order programs. *Proceedings JELIA '98, appears in LNAI series (No. 1489, 1998). Vol. Editor J. Dix.*, pages 47–61, 1998.
8. Mauricio Osorio and Bharat Jayaraman. Aggregation and negation-as-failure. *New generation computing*, 17(3):255–284, 1999.
9. Mauricio Osorio, Bharat Jayaraman, and Juan Carlos Nieves. Declarative pruning in a functional query language. In Danny De Schreye, editor, *Proceedings of the International Conference on Logic Programming*, pages 588–602. MIT Press, 1999.
10. Mauricio Osorio, Bharat Jayaraman, and David Plaisted. Theory of partial-order programming. *Science of Computer Programming*, 34(3):207–238, 1999.
11. Mauricio Osorio and Juan Carlos Nieves. Stratified partial-orden programming. In Eleni Stroulia and Stan Matwin, editors, *Canadian Conference in AI'2001*, pages 225–235. Springer-Verlag, LNAI 2056, 2001.
12. S. Parker. Partial order programming. In *Proc. 16th Symo. on Principles of Programming Languages*, pages 260–266, ACM, Press, 1989.
13. Enrico Pontelli and Omar El-Khatib. Construction and optimization of a parallel engine for answer set programming. *PADL 2001*, pages 200–303, 2001.
14. D.R. Stinson. *An introduction to the Design and Analysis of Algorithms*. The Charles Babbage Research Centre, Winnipeg, Canada, 1987.