Relaxation in Partial-Order Programming

Juan Carlos Nieves, Bharat Jayaraman

Department of Computer Science and Engineering
State University of New York at Buffalo,
Buffalo, NY 14260, USA,
jcnieves@cse.buffalo.edu (Visiting Scholar)
bharat@cse.buffalo.edu

Abstract. Optimization has been a hot topic in the area computer science for many years. In many applications, optimal solutions may be difficult or impossible to obtain, and hence we are interested in finding suboptimal solutions. In this paper we introduce the concept of relaxation in the framework of Partial-Order Programming. The idea of introduce relaxation in Partial-Order Programming is allow to the users give suboptimal solutions in Partial-Order Programming. For this purpose, we define a declarative semantics based on stable semantics which extends the standard declarative semantics to Partial Order Programs defined in [8].
keywords: Logic programming, declarative languages, optimization.

1 Introduction

In two recent papers we formulated a functional query language based upon *partial-order clauses* [2, 7], and we refer the reader to these papers for a full account of the paradigm. In comparison with traditional equational clauses for defining functions, partial-order clauses offer better support for defining recursive aggregate operations. We illustrate this with an example from [7]: 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 a node `X` to node `Y`, then the shortest distance from `X` to `Y` can be declaratively specified through partial-order clauses as follows:

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

The meaning of a ground expression (such as `short(a,b)`) is the *greatest lower bound* (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 it is necessary that the constituent functions be monotonic. This could happen in the above example when the underlying graph is cyclic. We refer to this paradigm as *partial-order programming*, and we have found that it offers conciseness, clarity, and flexibility in programming problems in graph theory, optimization, program analysis, etc. Partial-order program clauses are actually a generalization of subset program

clauses [5, 6] which allow one to elegantly and efficiently program set-oriented computations.

The present paper, we introduce the concept of relaxation in the framework of Partial-Order Programming. The idea of introduce relaxation in Partial-Order Programming is allow to the users give suboptimal solutions in Partial-Order Programming. For this purpose, we define a declarative semantics based on stable semantics which extends the standard declarative semantics to Partial Order Programs defined in [8].

In order to show the idea of relaxation in Partial-Order Programming, we consider again our example of shortest distance with the following extended database:

```
edge(a,b,10).
edge(a,m,12).
edge(b,m,5).
edge(m,x,5).
edge(x,z,20).
edge(m,z,3).
```

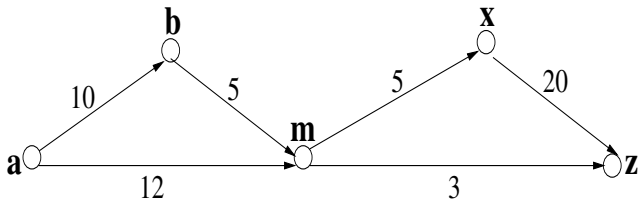

Fig. 1. Graph depicting the `edge` predicate.

The corresponding graph to the above facts is shown in Figure 1. As we can see the shortest distance between the node `a` and the node `z` is 15. But what happen, if we are interesting on the shortest distance between the node `a` and the node `z` such that it includes the node `b`. In this case, we have to relax our goal. For this purpose we introduce the concept of relaxing goal as follows:

```
relax short(a,z) such that short(a,z) = short(a,b) + short(b,z)
```

The intuitive mean of the before relaxing goal is: compute the shortest distance between the node `a` and the node `z` such that it includes the node `b`. Other interesting goal is when we want the shortest distance between the node `a` and the node `z` such that it is greatest that 15. In this case, we can express a relaxation goal as follows:

```
relax short(a,z) such that short(a,z) > 15
```

As we can look the relaxing goals allow to specify goals where the objective is compute suboptimal solutions. The declarative semantics of this framework is defined using stable semantics. And the mail idea is translate the partial order clauses to normal clauses, this idea will be presented in the section 3.

The rest of the paper is structured as follows : In section 2, we present an informal introduction to the paradigm of partial-order programming. In section 3, we present the translation of Partial-Order Programs to Normal Programs w.r.t. a relaxing goal and finally in the last section we present our conclusions.

2 Background

We give an informal introduction of the paradigm of partial-order programming. There are two kinds of partial-order clauses: unconditional and conditional. Unconditional partial-order clauses have the form:

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

where each variable in *expression* also occurs in *terms*. For simplicity of presentation in this paper, we assume that every function f is defined either with $\geq$ or $\leq$ clauses, but not both-this restriction has been easy to meet and it cover a large class of practical programs. The syntax of *terms* and *expression* is as follows:

term ::= *variable* | *constant* | *c*(*terms*)
terms ::= *term* | *term* , *terms*
expression ::= *term* | *c*(*exprs*) | *f*(*exprs*)
exprs ::= *expression* | *expression* , *exprs*

Conditional partial-order clauses are of the form:

$f(terms) \geq expression$:- *condition*
$f(terms) \leq expression$:- *condition*

where each variable in *expression* occurs either in *terms* or in *condition*, and *condition* is in general a conjunction of relational or equational goals defined as follows.

condition ::= *goal* | *goal*, *condition*
goal ::= *p(terms)* | $\neg$ *p(terms)* | $f(terms) = term$

Declaratively speaking, the meaning of a clause is that, for all its ground instantiations, the partial-order at the head is taken to be true if the *condition* is true. In general, multiple clauses may be used in defining some function *f*. For a function defined by $\geq$ clauses, we define the meaning of a ground expression *f(terms)* to be equal to the *least-upper bound* (respectively, *greatest-lower bound* for $\leq$ clauses) of the resulting terms defined by the different partial-order clauses for *f*. Procedurally, *condition* is processed first before *expression* is evaluated. When new variables appear in *condition* (i.e., those that are not on the left-hand side), the goals in *condition* are processed in such an order so that all

functional goals (*f(terms)*) and all negated goals ($\neg$ *p(terms)*) are invoked with ground arguments—note that negation-as-failure may be unsound for nonground negated goals. The predicates appearing in *p(terms)* are referred to as *extensional database predicates*(EDB) because they are defined by ground unconditional (or unit) clauses.

Definition 2.1. *A relaxing goal is of the form:* `relax f(t) such that condition`, *where* `f` *is an user function,* `t` *a ground term and condition is of the form:*

`h(expression)` *where* `h` *is a function or a relational symbol*

To illustrate the framework of Partial-Order Programming. We present some interesting examples.

Example 2.1 (Data-flow Analysis). 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 example also shows the use of monotonic functions.

```
reach_out(X) ≥ reach_in(X) - kill(X).
reach_out(X) ≥ gen(X).
reach_in(X)  ≥ reach_out(Y) :- pred(X,Y).
```

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 basic block `X`. We assume an EDB `pred(X,Y)`, that defines when *Y is predecessor of X*. The set-difference operator (`-`) is monotonic in its first argument, and hence the program has a unique intended meaning as it is shown in [1]. A general result that explains this fact can be found in [7]. Our operational semantics behaves exactly as the algorithm proposed in [1] to solve this problem.

Example 2.2 (0-1 Knapsack Problem).
This 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 $0 \leq \mathtt{M} \leq m$, and $1 \leq \mathtt{I} \leq n$, define $\mathtt{kn_{01}(I,M)}$ to be the profit of the optimal solution to the 0-1 knapsack problem, using objects $1, ..., \mathtt{I}$, and knapsack capacity `M`. Then, $\mathtt{kn_{01}}$ is defined by the following inequalities:

$\mathtt{kn_{01}(0,M)} \geq 0.$
$\mathtt{kn_{01}(I,M)} \geq \mathtt{kn_{01}(I-1,M)}$:- $\mathtt{I} \geq 1.$
$\mathtt{kn_{01}(I,M)} \geq \mathtt{kn_{01}(I-1,M-c(I))} + \mathtt{g(I)}$:- $\mathtt{I} \geq 1,\ \mathtt{c(I)} \leq \mathtt{M}.$

The top-level query might look as follows, $\mathtt{kn_{01}(5,30)}$, assuming `5` products and a budget of `30`. The meanings of the three clauses are explained below:

(1) Consider $\mathtt{kn_{01}(0,M)} \geq 0$.
If we have 0 objects (regardless of our capacity) our profit is 0. Our rule is a weaker assertion, namely, that our profit should be ≥ 0. But no other clauses support another greater value as a conclusion. Thus, by a form of closed word assumption we conclude that $\mathtt{Kn_{01}(0,M)} = 0$.

(2) Consider $\mathtt{kn_{01}(I,M)} \geq \mathtt{kn_{01}(I-1,M)}$:- $\mathtt{I} \geq 1$.

In the second clause, if we have a least one object and capacity M, then our profit is $\geq$ that we could get by taking out one element (in this case the last one).

(3) Consider $\mathtt{kn_{01}(I,M)} \geq \mathtt{kn_{01}(I-1,M-c(I))} + \mathtt{g(I)}$ `:- I ≥ 1, c(I) ≤ M.` In the last clause, suppose that the cost of our last object $\leq$ to our current capacity. Suppose in addition that we decided to carry it. Thus, our capacity will decrease to $M - c(I)$. However, we can count the profit of this object. So, our profit in this case would be $\mathtt{kn_{01}(I-1,M-c(I))} + \mathtt{g(I)}$ **(1)**. Moreover, our real profit would be surely $\geq$ **(1)**.

Note that we only have to state valid inequalities and the "current" equality is inferred by our semantics. Thus, if in some cases the second and the last clauses are satisfied, our semantics selects the one that maximizes the profits.

3 Translation

In this section, we present the translation of partial- order programs to normal programs w.r.t. a relaxation goal. We transform our programs to normal programs because we use *Stable Semantics*[3] to define the declarative semantics of our framework.

We have already study how can we translate partial order programs to normal programs in [8], so the new contribution is adding the relaxation concept.

The notation used to introduce partial-order clauses in section 2 will be referred to henceforth as the *nested* form. But, formally, we will see each clause as a short hand of a normal clause. This is done by flattening all expressions so that the arguments of all function calls are terms. Since all variables range over the universe of terms, this flattened form makes more explicit that the result of an expression must be a term (for a good introduction to flattened form see [4]). For example, we consider again our program of the introduction which is in the nested form. The flattened form of this program is as follows:

```
short(X,Y) ≤ C :- edge(X,Y,C)
short(X,Y) ≤ D :- edge(X,Z,C1), short(Z,Y)= C2, C1+ C2 = D.
```

The second step of the translation is associated a normal form to the partial-order clauses. For instance, the associated *normal form* of the above clauses are the *normal clauses*:

$\mathtt{short_{\leq}(X,Y,C)} : -\mathtt{edge(X,Y,C)}$

$\mathtt{short_{\leq}(X,Y,D)} : -\mathtt{edge(X,Z,C1)}, \mathtt{short_{\leq}(Z,Y,C2)}, \mathtt{C1 + C2 = D}.$

where each atom with predicate symbol $\mathtt{short_{\leq}}$ is called an $atom_{\leq}$. We assume that the new predicate symbol $\mathtt{short_{\leq}}$ is not present in the original language. Observe that the associated normal form to `short(Z,Y)= C2` is $\mathtt{short_{\leq}(Z,Y,C2)}$.

The last step of the transformation lies in add some normal clauses such that these compute the optimal answer w.r.t. the query's condition. For example, we suppose that we want to compute the following relaxing goal: `relax sh(a,z) such that sh(a,z) > 15`. Then the translation of the shortest distance programs to normal programs w.r.t. `relax sh(a,z) such that sh(a,z) > 15` is as follows:

1. $\texttt{short}_{\leq}(\texttt{X},\texttt{Y},\texttt{top})$.
2. $\texttt{short}_{\leq}(\texttt{X},\texttt{Y},\texttt{C})$:- $\texttt{edge}(\texttt{X},\texttt{Y},\texttt{C})$.
3. $\texttt{short}_{\leq}(\texttt{X},\texttt{Y},\texttt{C})$:- $\texttt{edge}(\texttt{X},\texttt{Z},\texttt{C1}),\ \texttt{short}_{\leq}(\texttt{Z},\texttt{Y},\texttt{C2}),\ \texttt{C} = \texttt{C1} + \texttt{C2}$.
4. $\texttt{short}_{\texttt{relax}}(\texttt{W},\texttt{W1},\texttt{X})$:- $\texttt{short}_{\leq}(\texttt{W},\texttt{W1},\texttt{X}), \texttt{short}_{\texttt{condition}}(\texttt{W},\texttt{W1},\texttt{X})$.
5. $\texttt{short}_{\texttt{condition}}(\texttt{a},\texttt{z},\texttt{C})$:- $\texttt{short}_{\leq}(\texttt{a},\texttt{z},\texttt{C}), \texttt{C} > 15$.
6. $\texttt{short}_{<}(\texttt{W},\texttt{W1},\texttt{X})$:- $\texttt{short}_{\texttt{relax}}(\texttt{W},\texttt{W1},\texttt{X1}),\ \texttt{X1} < \texttt{X}$.
7. $\texttt{short}_{=}(\texttt{W},\texttt{W1},\texttt{X})$:- $\texttt{short}_{\texttt{relax}}(\texttt{W},\texttt{W1},\texttt{X}), \neg\texttt{short}_{<}(\texttt{W},\texttt{W1},\texttt{X})$.
8. $\texttt{query}(\texttt{a},\texttt{z},\texttt{X})$:- $\texttt{short}_{=}(\texttt{a},\texttt{z},\texttt{X})$.

Of course, this program is completed with its extended database of the predicate `edge`. The clause 4 defines the predicate $\texttt{short}_{\texttt{relax}}$ which takes all the possible answers such that satisfy the query's condition. The clause 5 represents the query's condition. The purpose of the clauses 6 and 7 is compute all the optimal answers. And finally, the clause 8 takes the answer to the query.

As, we have a normal program so we can use a semantics to normal programs as *Stable Semantics* to compute the intending meaning of the program.

Now, we will present some definitions about the translation and the definition of our declarative semantics.

Definition 3.1. *Let Q_r be a relaxing goal of the form* `f(t) such that condition`. *We define $Flat(Q_r)$ as the flattened form of* `condition`.

To illustrate the before definition, we consider the following relaxing goal: `relax sh(a,e) such that sh(a,e) > 30`. So $Flat(Q_r) := sh(a,e) = C, C > 30$.

Definition 3.2. *Given a flattened form $Flat(Q_r)$ of a query's condition. We define the $normal\_form(Flat(Q_r))$ as the formal form associated to $Flat(Q_r)$.*

We suppose that $Flat(Q_r) := sh(a,e) = C, C > 30$, so $normal\_form(Flat(Q_r)) = sh_{\leq}(a,e,C), C > 30$

Definition 3.3 (Head symbols). *Given a program P, we define head(P) to be the set of head symbols of P, i.e., the head symbols on the literals on the left-hand sides of partial-order clauses.*

Definition 3.4. *Given a predicate symbol $\texttt{f}_{\geq}$ and a relaxing goal Q_r of the form* `:f(t) such that condition`, *we define* **basic_ext**$(\texttt{f}, Q_r)$ *as the following set of clauses:*

(1) $\texttt{f}_{=}(\texttt{Z},\ \texttt{S})\ \texttt{:-}\ \texttt{f}_{relax}\ (\texttt{Z},\ \texttt{S}),\ \neg\ \texttt{f}_{>}(\texttt{Z},\ \texttt{S})$

(2) $\texttt{f}_{>}(\texttt{Z},\ \texttt{S})\ \texttt{:-}\ \texttt{f}_{relax}(\texttt{Z},\texttt{S1}),\ \texttt{S1} > \texttt{S}$

(3) $\texttt{f}_{\texttt{relax}}(\texttt{Z},\texttt{S})\ \texttt{:-}\ \texttt{f}_{\geq}(\texttt{Z},\texttt{S}), \texttt{f}_{\texttt{condition}}(\texttt{Z},\texttt{S})$

(4) $\texttt{f}_{\texttt{condition}}(\texttt{t},\texttt{S})\ \texttt{:-}\ normal\_form(Flat(Q_r))$

(5) $\texttt{f}_{\geq}(\texttt{Z},\ \texttt{S})\ \texttt{:-}\ \texttt{f}_{\geq}(\texttt{Z},\texttt{S1}),\ \texttt{S1} > \texttt{S}$

(6) $\texttt{f}_{\geq}\ (\texttt{Z},\ \bot)$

(7) $\texttt{f}_{\geq}(\texttt{Z},\texttt{C})\ \texttt{:-}\ \texttt{f}_{\geq}(\texttt{Z},\texttt{C}_1),\ \texttt{f}_{\geq}(\texttt{Z},\texttt{C}_2),\ \ \texttt{lub}(\texttt{C}_1,\texttt{C}_2,\texttt{C})$.

Notice that one of the `f_condition`'s arguments is ground. In fact, this argument is the mail query's argument.

Definition 3.5. *Given a relaxing goal of the form:* `f(t) such that condition`, *we define* **query-clause**(`f`) *as following:*

$$query(t, S) : -f_{=}(t, S).$$

where $f_{=}$ is a predicate defined in **basic_ext**(`f`)

Definition 3.6 (Basic_ext and basic_trans of a program). *Given a program P and a relaxing goal Q_r, we define*
basic_ext(P,Q_r) := $\bigcup_{f_{\geq} \in\ head(P)}$ basic_ext($f_{\geq}$,Q_r), and
basic_trans(P,Q_r) := P' $\cup$ basic_ext(P,Q_r).
Where P' is the program obtained from P translating each partial-order clause of P to a normal program.

Proposition 3.1. *For any partial-order program P and a relaxing goal Q_r, basic_trans(P,Q_r) is stratified.*

Definition 3.7. *For any partial-order program P and relaxing goal Qr of the form:*`f(t) such that condition`, *we define its declarative semantics of $P \cup Qr$ denoted as $D(P \cup Qr)$, as the stable model semantics for basic_trans$(P, Q_r) \cup$ query-clause*(`f`) .

4 conclusion

In this paper we introduce the concept of relaxation in the framework of Partial-Order Programming. The idea of introduce relaxation in Partial-Order Programming is allow to the users give suboptimal solutions in Partial-Order Programming. For this purpose, we define a declarative semantics based on stable semantics which extends the standard declarative semantics to Partial Order Programs defined in [8].

References

1. Alfred V. Aho, Ravi Setvi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison Wesley, 1988.
2. Mauricio Osorio Bharat Jayaraman and K. Moon. Partial order programming (revisited). In M. Nivat V.S. Alagar, editor, *Proc. AMAST*, LNCS 936, pages 561–575. Springer-Verlag, 1995.
3. 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.
4. D. Jana and Bharat Jayaraman. Set constructors, finite sets, and logical semantics. *Journal of Logic Programming*, 38(1):55–77, 1999.

5. Bharat Jayaraman. Implementation of subset-equational programs. *Journal of Logic Programming*, 11(4):299–324, 1992.
6. Bharat Jayaraman and K. Moon. Subset logic programs and their implementation. *Journal of Logic Programming*, 41(2):71–110, 2000.
7. Bharat Jayaraman Mauricio Osorio and David Plaisted. Theory of partial-order programming. *Science of Computer Programming*, 34(3):207–238, 1999.
8. Mauricio Osorio and Bharat Jayaraman. Aggregation and negation-as-failure. *New generation computing*, 17(3):255–284, 1999.