Declarative Output by Ordering Text Pieces

STEFAN BRASS

Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik,
Von-Seckendorff-Platz 1, D-06099 Halle (Saale), Germany
(*e-mail:* brass@informatik.uni-halle.de)

submitted 24 January 2011; revised —; accepted —

Abstract

Most real-world programs must produce output. If a deductive database is used to implement database application programs, it should be possible to specify the output declaratively. There is no generally accepted, completely satisfying solution for this. In this paper we propose to specify an output document by defining the position of text pieces (building blocks of the document). These text pieces are then ordered by their position and concatenated. This way of specifying output fits well to the bottom-up way of thinking about rules (from right to left) which is common in deductive databases. Of course, when evaluating such programs, one wants to avoid sorting operations as far as possible. We show how rules involving ordering can be efficiently implemented.

KEYWORDS: declarative output, deductive databases, bottom-up evaluation, ordering, implementation

1 Introduction

Currently, database application programs are usually developed in a combination of two or more languages, e.g. PHP for programming and SQL for the database queries and updates. SQL is declarative, but most languages used for the programming part are not.

Deductive databases can be successful in practice if and only if they offer a big step forward, a "quantum leap". Recursive query evaluation alone is not such a big step, although it has become more important in the last time because tree- and graph-structured data are used more often (e.g. for XML data, web pages, and RDF).

However, the goal of deductive databases was from the beginning much broader: Only a single, declarative language should be used for programming and database tasks. The advantages of declarativity have been clearly shown in SQL: The productivity is higher (because the programs are shorter and there is no need to think about efficient evaluation), and new technology (parallel hardware and new data structures and algorithms) can be used for existing application programs without changing them (only the DBMS needs to be updated).

While in general, it might be difficult to reach acceptable performance for really declarative programs (Prolog is not completely declarative), database applications are quite special and usually not very difficult. For the most part, the task of an application program is to check the input and to generate an output document (e.g., a web page).

Although generating output is practically very important, it seems that there is no really good solution in logic programming yet. The standard solution in Prolog with a `write`-predicate is clearly non-declarative: It depends on the specific evaluation order used in Prolog.

The Gödel programming language, which improves upon Prolog in many ways, did also not solve this problem: "Gödels input/output facilities do not have a declarative semantics, so it is very important that input/output predicates are confined to as small a part of a program as possible." (Hill and Lloyd 1994). Certainly, it is a good advice to separate the generation of output documents from the more complex logic of the program. But at least for database applications, the generation of output is a substantial part of the program, and deserves a declarative specification, too.

Another standard way is to use an accumulator pair: A prefix of the document to be generated is passed as an input argument to every predicate involved in generating output, and the current state of the document after the output of the predicate is returned. For instance, this solution is used in Mercury. The Mecury tutorial (Becket 2010) contains this example:

```
main(IOState_in, IOState_out) :-
        io.write_string("Hello, ", IOState_in, IOState_1),
        io.write_string("World!", IOState_1, IOState_2),
        io.nl(IOState_2, IOState_out).
```

This is declarative, but has some problems, too:

1. It complicates the programs (many predicates have two additional arguments). In part, this problem can be solved by using special syntactic features to hide these arguments. E.g., Mecury has special state variables as a syntactic convenience (Becket 2010) (only one argument is written instead of two, and the numbering of variables is done automatically).
2. Another problem is that if backtracking is possible, the actual output cannot be immediately done, and furthermore, the possibility remains that the program might sometimes produce alternative documents, which is certainly not expected. In the Mercury language, this problem is solved by checking the determinism of predicates which perform I/O, and using destructive input and unique output modes for the arguments (Becket 2010).
3. This solution needs recursion already for simple tasks, e.g. printing the contents of a database table. As Molham Aref stated in his invited talk at last year's ICLP, one of the advantages of Datalog is that not so experienced users can be easily offered subsets of the language with restricted complexity. It seems unlucky that recursion is basically unavoidable if one uses this state-passing method for specifying output.
4. The situation is even worse, because one also needs lists or similar data structures: In databases, several answers are most naturally constructed as several

solutions to a query (e.g. via backtracking). For instance, database tables are normally represented as sets of facts. This does not fit well with requiring deterministic code for output: One must use a predicate like `findall`, and then recurse over the resulting list. (Our proposal also contains lists, but many interesting programs do not need recursion, and do not need to inspect the constructed lists, i.e. no complex terms in body literals are needed. Furthermore, the use of lists can be hidden with some syntactic sugar.)

In functional programming, monads are used for declarative output (Wadler 1997; Peyton Jones 2000). However, they depend on higher order programming, which is not common practice in logic programming. Therefore, monads are not easily understandable for logic programmers.

2 Basic Idea

The basic idea of our proposed solution is very simple: The Datalog program defines a predicate `output` with two arguments: The first argument determines the position in the output, and the second argument is a text piece.

```
output(1, 'Hello, ').
output(2, Name) ← name(Name).
output(3, '.').
name('Nina').
```

The generated output document can be computed as follows: One first uses standard bottom-up evaluation to generate all derivable facts of the form `output(P,T)`, then one sorts them by the first argument `P`, and prints the strings `T` in this sequence. Two comments must be made here:

1. Of course, nobody likes to write the numbers for the output sequence, but syntactic sugar can be used to hide that (see Section 3). Basic Horn clauses are used to have a simple, common semantic framework, but different tasks might need different syntactic variants to specify the task clearly and concisely. For instance, definite clause grammar rules are a very useful notation in Prolog for syntax analysis tasks.
2. In the same way, there is not only one algorithm for the evaluation of rules, but different algorithms can be used for special cases of rules. In the above example, it would not be necessary to first generate all derivable `output`-facts and then sort them: Since the sorting argument is given explicitly in the rules, the rules might be applied in that order, and the text pieces immediately printed. But it is nevertheless important to have a very simple evaluation algorithm to present a directly executable semantics before discussing optimizations.

Of course, using only numbers as position specifications becomes impractical as soon as one wants to produce already slightly more complex documents. Therefore, we suggest to use a list as the first argument of `output`. Lists are sorted as usual: They are compared element by element, and the first position in which they differ decides the sequence. For instance, suppose we have stored homework points in

```
output([1], '<table>\n').
output([2], '<tr><th>Points</th><th>Student</th></tr>\n').
output([3,Points,Name,1], '<tr><td>')    ←  homework(Name, Points).
output([3,Points,Name,2], Points)        ←  homework(Name, Points).
output([3,Points,Name,3], '</td><td>')   ←  homework(Name, Points).
output([3,Points,Name,4], Name)          ←  homework(Name, Points).
output([3,Points,Name,5], '</td></tr>')  ←  homework(Name, Points).
output([4], '</table>\n').
homework('Ann', 5).
homework('Bob', 10).
homework('Chris', 10).
```

Fig. 1. Generating an HTML Table with Homework Results

a predicate `homework(Name, Points)`. An HTML table which contains the data sorted by points, and for eqaul points by name, can be generated as shown in Figure 1. Note that the sorting based on the data is for free if one uses this approach: Since the ID of the text piece contains first the points and then the name, these two sorting criteria are applied with the points having higher priority.

3 Syntactic Sugar

3.1 Implicit Position

Of course, one wants to eliminate the ordering argument as far as possible. A straightforward solution is that one can avoid the position argument in the `output` predicate, i.e. if the `output`-predicate is used with only one argument, the first is filled with an implicit position computed as follows:

1. At the beginning, the implicit position is `[]`.
2. If `output` is used with only one argument, this is the text piece, and the position is generated as follows: If the current implicit position ends in a number, the position is used with this number incremented by 1. Otherwise, the element `1` is appended to the current implicit position. In both cases, the generated list is the new implicit position.
3. One can use a rule defining a special predicate `output:loop` (with an arbitrary number of arguments), i.e.

   $$\texttt{ouput} : \texttt{loop}(\mathsf{X}_1, \ldots, \mathsf{X}_n) \leftarrow \mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_m$$

   immediately followed by further rules enclosed in `{...}`. This first computes the next implicit position (i.e. adds 1 to the last number), and then appends $[\mathsf{X}_1, \ldots, \mathsf{X}_n]$ to this implicit position. Furthermore, it adds $\mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_m$ to the body of every rule in the following `{...}` section. After the loop section, it removes $[\mathsf{X}_1, \ldots, \mathsf{X}_n]$ from the implicit position.

In order to mark that these special abbreviation is used, such rules end in "`;`" instead of "`.`". (In this way, also other predicates besides `output` could be added

```
output('<table>\n');
output('<tr><th>Points</th><th>Student</th></tr>\n');
output:loop(Points, Name) ← homework(Name, Points) {
    output('<tr><td>');
    output(Points);
    output('</td><td>');
    output(Name);
    output('</td></tr>');
}
output('</table>\n');
```

Fig. 2. Generating an HTML Table with Implicit Position

an implicit first positioning argument. Of course, then every predicate must have its own implicit position.)

With these syntactic rules, the example from Figure 1 can be written as shown in Figure 2. Of course, now the sequence of rules is important. This might seem like violating again declarativity, but probably declarativity only means that there is a simple logical semantics, not refering to a specific execution algorithm, so that the system still has many realistic options for executing the program. That is given. Furthermore, output is always something sequential. If one wants that the rule sequence is not important, one must encode the sequence in the arguments.

3.2 Patterns

With the above syntax, one needs to split a text whenever a parameter value must be inserted. Small pieces of text are enclosed in quotation marks, and line breaks must be explicitly marked with special commands. In order to get a better overview, it should be possible to write longer pieces of text, just as it appears in the output, and mark the places where something must be inserted. This is possible with the pattern syntax which we propose. Figure 3 shows the running example in this syntax, and Figure 4 shows the rules that are generated by the patterns.

Each pattern corresponds to a predicate. One writes the pattern/predicate name, then "`(`", then possibly a list as part of the position specification (if the pattern is instantiated multiple times), then a "`#`", the several lines which form the text of the pattern, then "`#)`", and then possibly a rule body. Within the pattern text, two special markers can be used: "`<#p>`" links to another pattern `p`, which is embedded here, and "`<$X>`" inserts the value of a variable which is bound in the rule body.

The generated predicates have two arguments just as the `output`-predicate (one can use pattern syntax for the `output`-predicate, too). The pattern text is internally split into as many pieces as needed, at least the inserted patterns and variables must be pieces of their own, but one can in addition split the text at line breaks. Each piece gets a sequential number. Then one fact or rule is generated for each piece:

1. The position consists of the optional list of variables (specified before the `#`

```
homeworks_table(#
    <table>
    <tr><th>Points</th><th>Student</th></tr>
    <#homeworks_row>
    </table>
#).

homeworks_row([Points, Name]#
        <tr><td><$Points></td><td><$Name></td></tr>
#) ← homeworks(Name, Points).
```

Fig. 3. Generating an HTML Table with Patterns

```
homeworks_table([1], '<table>\n').
homeworks_table([2], '<tr><th>Points</th><th>Student</th></tr>\n').
homeworks_table([3|X], Y) ← homeworks_row(X, Y).
homeworks_table([4], '</table>').

homeworks_row([Points, Name, 1], '<tr><td>').
homeworks_row([Points, Name, 2], Points).
homeworks_row([Points, Name, 3], '</td><td>').
homeworks_row([Points, Name, 4], Name).
homeworks_row([Points, Name, 5], '</td></tr>').
```

Fig. 4. Internal Rules denoted by Patterns

in the first line of the pattern), then the piece number, and in case of an embedded pattern, the position argument of that pattern.

2. The second argument is either the text piece, or, a variable is inserted at this point, the variable, or, if a pattern is embedded at this point, a variable for a text piece of that pattern.
3. If the pattern has a body, it is attached to each generated rule. If this position is for an embedded pattern, a call to the corresponding predicate is added to the body.

Note that the solution with patterns avoids the slightly imperative (non-declarative) "look and feel" of the implicit position solution.

4 Efficient Evaluation

The goal is not having to do a lot of sorting when evaluating the rules. Sorting is a relatively expensive operation, and furthermore it is necessary to store intermediate results. Relational database management systems often work internally with tuple streams, where the next tuple is computed on demand only (using an iterator/cursor interface). In this way, the materialization of intermediate results is avoided. Only sorting needs intermediate storage, because the first result tuple can only be produced after the last input tuple was seen.

Often, already by choosing the right evaluation order of the rules, sorting can be avoided. Furthermore, if we can get the tuples for the predicates in the body in the right sorted sequence, this sorting can often be preserved. For the base predicates (the EDB predicates stored in database relations), sorting can sometimes be avoided, if they are stored in an ISAM file or index-organized table (if the required sorting sequence matches the key) or if a b-tree index with a fitting search key is available.

First note that the only way in which the predicate `output` should be treated specially is that the second argument is printed in the end, when the tuples (derived facts) are accessed ordered by the first argument. For instance, it should be possible to write

```
output([X, Y], Z) ← p(Z, X, Y).
```

Thus, in order to produce the tuples for `output` ordered by first argument, we need to generate the tuples for `p` ordered by second and third arguments (second argument with higher priority than third argument, i.e. the third argument is only important for determining the order of two tuples if the second argument is equal). So in general, we need to produce tuples for any predicate in the program ordered by any given sequence for the arguments.

In order to concentrate on the sorting problem, we make two simplifying assumptions in this paper:

- We do not consider recursive rules. There is a large body of work on evaluating recursive rules, but in most cases, sorting sequences for the predicates in the body do not give a sorting sequence for the derived tuples. Furthermore, in order to guarantee termination, the resulting tuples must be stored (materialized), so that duplicates can be eliminated. When this is anyway needed, one could of course choose a b-tree or similar data structure which gives the required sorting on output.
- While we need structured terms in the head of the rules (to build lists for the position specifications), we assume that the body literals contain only variables and constants.

4.1 Interface for Relations

As usual in relational DBMS, we use a tuple stream interface for relations (predicates), i.e. it is possible to open a cursor (scan, iterator) over the relation, which permits to loop over all tuples. We assume that for every normal predicate p, there is a class p`_cursor` with the following methods:

- `void open()`: Open a scan over the relation, i.e. place the cursor before the first tuple.
- `bool fetch()`: Move the cursor to the next tuple. This function must also be called to access the first tuple. It returns `true` if there is a first/next tuple, or `false`, if the cursor is at the end of the relation.

- *T* `col_`*i*`()`: Get the value of the *i*-th column (attribute). Here *T* is the type of the *i*-th column. This interface permits to get access to the attribute values without actually materializing the tuple (especially, large data values do not need to be copied).
- `void close()`: Close the cursor.
- `p_cursor_pos position()`: Return current position of the cursor for a later call to `restore` (in order to return to this position). This is only used for complicated loop structures, and very seldom more than one position must be saved at the same time.
- `void restore(p_cursor_pos)`: Return to a previously saved position of the cursor. It is possible to return multiple times to the same position.

The objects of the class `p_cursor` do not guarantee any specific order of the returned tuples. Thus, we also need classes `p_cursor_`i_1`_..._`i_k, which return tuples sorted by argument i_1 with highest priority, i_2 with second highest priority, and so on.

Our goal in the section is to create code for such cursors for the derived predicates (IDB predicates), given such cursors for the base predicates (stored relations, EDB predicates). As mentioned above, the storage structures for the base predicates sometimes give the sorting more or less for free, othewise some explicit sorting is unavoidable.

Again, in order to focus on the main problem (the sorting), and not to overload the paper with details already considered extensively in the literature, we ignore a very important optimization:

- Often, when a predicate is called, values are known for some arguments. Thus, one does not need to produce the entire extension of the predicate. One could extend the cursor interface by attaching a binding pattern to the cursor name (specifying which arguments are bound in the call, i.e. input arguments), and permit to specify values for them in the `open()` call. This is also interesting for base predicates with indexes.
- This situation is even more complicated, since some arguments are known at compile-time, others only at runtime. Furthermore, one might want to pass not only equality conditions for the arguments to the called predicate, but also other simple conditions (e.g. $\mathsf{X} \leq 5$). An extreme case in this direction is the SLDMagic-method of the author (Brass 2000). There is a large body of literature on Magic Sets that treats such problems.

4.2 Single Rule, Nested Loop Join

Consider the rule

$$\mathsf{p}(\mathsf{t}_1, \ldots, \mathsf{t}_n) \leftarrow \mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_m$$

and suppose an ordering of the derived tuples by arguments $i_1, \ldots, i_k$ is required.

Then we first determine a sequence of variables $\mathsf{X}_1, \ldots, \mathsf{X}_l$ that appear in the head such that the matches found for the body must be ordered in this sequence.

```
q_cursor_1_2 q;
r_cursor_2_1 r;
q.open();
while(q.fetch()) { // q(X, Y, a)
    if(q.col_3 == 'a') {
        int X = q.col_1();
        int Y = q.col_2();
        r.open();
        while(r.fetch()) { // r(Y, Z)
            if(r.col_1 == Y) {
                int Z = r.col_2();
                print(X, Y, Z); // Variable assignment satisfying body
            }
        }
        r.close();
    }
}
q.close();
```

Fig. 5. Nested Loop Join for $\mathsf{q}(\mathsf{X}, \mathsf{Y}, a) \wedge \mathsf{r}(\mathsf{Y}, \mathsf{Z})$

These variables are simply the variables that appear in $\mathtt{t}_{i_1}, \ldots, \mathtt{t}_{i_k}$ in the order of first appearance. For instance, given the head

$$\mathsf{p}([a, \mathsf{X}, \mathsf{Y}, b], c, [\mathsf{Z}, \mathsf{Y}, \mathsf{X}])$$

and the sorting of the derived p-tuples by arguments $1, 2, 3$, the considered variable assignments must be sorted by values for $\mathsf{X}, \mathsf{Y}, \mathsf{Z}$ in this sequence.

The simplest implementation of the rule body is a nested loop join. It is well known that that an ordering of tuples in the outer loop is preserved by the nested loop join. This holds more generally. For instance, consider the rule body

$$\mathsf{q}(\mathsf{X}, \mathsf{Y}, a) \wedge \mathsf{r}(\mathsf{Y}, \mathsf{Z})$$

A nested loop join works as shown in Figure 5. Of course, the code shown in Figure 5 must be rewritten so that it fits itself in a `fetch()` method. Currently we would need coroutines (when a match found, instead of the `print`, one must do a "yield return", so the next call to `fetch()` starts at this place). But this restructuring of the code is a simple task for an able programmer. We have used the above code so that the nested loops are easily visible.

It is quite obvious that the program in Figure 5 produces the tuples ordered by the values for X ,Y, Z: Let $(\mathsf{X}_1, \mathsf{Y}_1, \mathsf{Z}_1)$ and $(\mathsf{X}_2, \mathsf{Y}_2, \mathsf{Z}_2)$ be two tuples that the above code yields, where the second tuple is produced later. Then there are two cases to consider:

- Suppose that the second tuple is produced in a different (later) iteration of the outer loop. The cursor over q guarantees that the tuples are considered sorted by first argument, and for equal first argument, by the second argument. There is no order for the third argument, but if we can assume that there are no duplicate tuples, the constant in the third argument means that tuples produced in different iterations of the loop must differ in at least one of the first two arguments. Therefore, we can conclude that either $X_1 < X_2$ or $X_1 = X_2$ and $Y_1 < Y_2$.

- Now suppose that both tuples are produced in the same iteration of the outer loop, but the second tuple is produced in a later iteration of the inner loop. Of course, we immediately get $X_1 = X_2$ and $Y_1 = Y_2$. Since the second argument is the main sorting criterion in the inner loop, it follows that $Z_1 \leq Z_2$.

Now, in general, suppose we need the variable assignments sorted by $X_1, \ldots, X_l$, and that the body $B_1 \wedge \cdots \wedge B_m$ is evaluated by nesting loops in this order, i.e. B_1 corresponds to the outermost loop and B_m to the innermost. Then the following condition must be satisfied for each $i = 1, \ldots, l$:

- Suppose that B_j is the first body literal, in which X_i appears.
- Then for any given values for $X_1, \ldots, X_{i-1}$, there can be only a single match for $B_1, \ldots, B_{j-1}$, (i.e. a single assignment of tuples to literals such that the conditions of $B_1, \ldots, B_{j-1}$ are satisfied), and furthermore,
- the argument in which X_i appears in B_j is in the sorting specification for the cursor, preceded only by arguments in which one of $\{X_1, \ldots, X_{i-1}\}$ or a constant appears.

This ensures that the variable assignments are produced in the right order: Suppose this were not the case. Then there are tuples $(d_1, \ldots, d_n)$ and $(d'_1, \ldots, d'_n)$ of values for $X_1, \ldots, X_n$, where the second tuple is produced later and for some $i \in \{1, \ldots, n\}$: $d_1 = d'_1$, …, $d_{i-1} = d'_{i-1}$, and $d_i > d'_i$. Let B_j be the first body literal in which X_i appears. Since for the given values $(d_1, \ldots, d_{i-1})$, there is only a single match for $B_1 \wedge \cdots \wedge B_{j-1}$, the outer loops are not switched forward between the two assignments, i.e. the problem occurs within a single run through the loop for B_j. But all arguments in B_j with higher sorting priority than the argument for X_i are filled with variables from $\{X_1, \ldots, X_{i-1}\}$ (which have the same value in both tuples), or constants (also the same value). Thus, the required ordering for B_j is violated.

4.3 More Interesting Loops

Of course, the above condition cannot always be reached by a permutation of the body literals. However, using only one loop per literal is not the only possibility.

Consider a rule with body

$$p(X, Y) \wedge q(Y, Z)$$

and suppose we need the matching assignments ordered by X, Z. If the outer loop is over `p_cursor_1_2`, and the inner loop over `q_cursor_2_1`, we get a wrong sequence given the following predicate extensions:

p	
X	Y
1	3
1	4

q	
Y	Z
4	5
3	6

Result	
X	Z
1	6
1	5

```
p_cursor_1 p;
q_cursor_2 q;
int X, Y, Z;
p.open();
bool more_p = p.fetch();
while(more_p) { // Loop over X-values from p
    X = p.col_1();
    p_cursor_pos x_start = p.position();
    q.open();
    while(q.fetch()) { // Loop over q
        Y = q.col_1();
        Z = q.col_2();
        p.restore(x_start);
        do { // Loop over a single X-partition in p
            if(p.col_2() == Y)
                print(X, Y, Z); // really coroutine return
        } while((more_p = p.fetch()) && p.col_1() == X);
    }
    q.close();
}
p.close();
```

Fig. 6. Evaluation of rule body $\mathsf{p}(\mathsf{X},\mathsf{Y}) \wedge \mathsf{q}(\mathsf{Y},\mathsf{Z})$ with output order X,Z

Here the condition is violated that for every value for X, there is only a single match for $\mathsf{p}(\mathsf{X},\mathsf{Y})$. The problem is that the value for Y from the first literal selects only specific facts for the second literal, and thus the ordering on Z in the second literal is not preserved, not even for a single X value.

However, it is possible to create a loop structure that preserves the ordering as required. The idea is to iterate with the outer loop only over different X-values, then to iterate with a middle loop over the second literal, and finally iterate with an innermost loop over the tuples of the first literal for a given X-value (i.e. an X-partition of p). This looks as shown in Figure 6. Note that every tuple combination is still considered only once, like in the standard case. Therefore, there is no (big) performance penalty for doing the iteration in a way that preserves the order.

Our claim now is that for any required ordering for the variables of the rule, we can select orderings for the body literals and write nested loops such that the variable assignments are produced in the required order. This means that for single rules, it is never necessary to do explicit sorting, if we can access the base predicates in any order. Probably not very many different orders occur in actual applications, so this assumption about the base predicates is not so unrealistic. But of course, it is always an option to do explicit sorting at any point in a query evaluation plan, and in a real system, a cost-based optimizer would be used to compare the different options.

So how would one automatically create code like the above one for any rule body $\mathsf{B}_1, \ldots, \mathsf{B}_m$ and sequence of variables $\mathsf{X}_1, \ldots, \mathsf{X}_n$?

1. For every literal B_i one creates a cursor over its predicate ordered by the argument positions filled with variables from $\mathsf{X}_1, \ldots, \mathsf{X}_n$ in the sequence of the variables. E.g. if the literal is $\mathtt{p}(\mathsf{X}_5, a, \mathsf{Y}, \mathsf{X}_2)$ the cursor is `p_cursor_4_1`.
2. Now one creates loops. The first set is guided by the variables $\mathsf{X}_1, \ldots, \mathsf{X}_n$. We use i as the index of the next variable to consider and start with $i = 1$. Of course, the next loop must be over a literal B_j that contains X_i. The selected literal may also contain X_{i+1}, ..., X_{i+k}. Then the next value for i is $i = i + k + 1$. There are two aspects that must be considered:
   - Is B_j is used for the first time (i.e. there is not already an enclosing loop for this literal/cursor)?
   - Is this the last loop for B_j, i.e. there will not be an inner loop on the same literal/cursor. This is the case if it contains only variables from $\mathsf{X}_1, \ldots, \mathsf{X}_{i+k}$ and constants, or if $i + k = n$ (i.e. the sequence of the remaining variables or further inner loops is not important).

   Thus, there are four cases to consider:

   (a) If it is the first and last (i.e. only) loop for B_j, we use a normal loop and are done with this literal. I.e. this loop opens the cursor before the loop and closes it after the loop, and does the `fetch` in the `while`-condition.

   (b) If this is the first, but not the last loop for B_j, we do a single `fetch` before the loop, and store the value in a boolean variable (as for `p` in the example). In the loop, we save the current values of X_i, ..., X_{i+k} in program variables, and save the current position of the cursor. No further calls to `fetch` are generated at this point, because an inner loop calls `fetch` until one of the variables changes.

   (c) If there are already enclosing loops for the same literal, but this is the last loop, we iterate only as far as variables from $\mathsf{X}_1, \ldots, \mathsf{X}_{i-1}$, which are contained in B_j, do not change. This case corresponds to the second loop over `p` in the example.

   (d) The final case is that this is not the first and not the last loop over B_j, i.e. there are enclosing as well as inner loops over the same literal.
   When there is an outer loop over the same literal, we always need to restore its saved position (more precisely, the saved position of the next outer loop over the same literal), before the currently generated loop starts (because between the two loops there is at least one loop over a different cursor, thus the currently generated loop must be executed several times).
   When there is an inner loop over the same literal, we must save the current position of the cursor inside the currently generated loop.
   Furthermore, this inner loop will switch the cursor to the next tuple after the range in which it works, i.e. at least one of the variables $\mathsf{X}_1, \ldots, \mathsf{X}_{i+k}$ will be changed (or the end of the tuple stream reached). This is why there is no explicit `fetch` in all except the innermost loop for a literal. The loop condition must check that the end is not yet reached, and the

values of those variables among $\mathsf{X}_1, \ldots, \mathsf{X}_{i-1}$ that appear in B_j are not changed.

3. Finally, loops must be created for each literal B_j that is not completely finished, i.e. it was not selected above, or had only cases (b) and (d). Depending on whether there is an enclosing loop, this is treated like case (a) or (c).

4.4 Using Merge Joins, Explicit Sorting and Intermediate Storage

Of course, ordered tuple streams are also useful for merge joins, which can be much faster than a nested loop join. In general, the output of a merge join remains ordered only on the join columns, which sometimes might be just what is needed, but often not.

With techniques like shown above in the "interesting loops" example, one could do e.g. an outer loop over X-values, and inside do merge joins over Y. This is not as good as a single merge join, but also not as bad as a nested loop join (if there are several Y-values for a single X-value).

There are many ways to evaluate a given logic program, and especially there is always the option to require no order for the computation of the tuples for a predicate p, and then do an explicit sorting before the tuples are used. This would for instance give all options for merge joins in the evaluation of rules for p. Furthermore, if the tuples are used multiple times (e.g. in an inner loop of a nested loop join), one has the advantage of computing them only once. The performance tests we did in (Brass 2010) show that this becomes an important factor if the computation of the tuples of p is not easy. Thus, even if no explicit sorting is needed (because the tuples are produced in the right order using the techniques shown above), it might still be useful to materialize the tuples for some intermediate predicates. In general, one would use a cost-based optimizer which generates a number of alternative evaluation plans, estimates their costs, and chooses the cheapest.

4.5 Multiple Rules for a Predicate

So far, we have only considered the evaluation of a single rule. Of course, there are often several rules about a predicate.

Again, the goal is to produce the tuples sorted by given arguments. In the most general case, we implement each rule as shown above, so that each rule produces the tuples in the required order, and then merge the tuple streams produced by the rules (i.e. we compare the current tuple of every rule, output the smallest, and fetch the next tuple from that rule).

However, often by analyzing the rule heads, we can see that all tuples produced by one rule must come before all tuples produced by another rule. In that case we can of course evaluate one rule first, and then the other. I.e. the explicit merging at runtime may be avoided or reduced, because we need to compare only tuples from rules where the order cannot be determined already at compile time.

Let us again consider the example from Figure 1. From the first list element in the head it is clear, that we must first apply the first rule, then the second, then the following block of rules, and finally the last rule.

Now the block of rules with first list element 3 is an interesting case, because they all have the same body, and each of the rules has the same variables in position 2 and 3 in the list, and position 4 is again a constant which can be ordered at compile-time. Thus, when we found one match for `homework(Name, Points)`, we can output five tuples in the order of the rules. Of course, the tuples for `homework` must be produced in sorted order, with the second argument sorted with higher priority.

Note that it is not required that the rule bodies are exactly equal. What is required is that we can get a superset for the possible values for the variables in the rule head in sorted order. Since after the variables, distinct constants follow, we can give each rule a chance in turn, whether it wants to produce facts with the given values for the variables. For instance, it would be no problem if the rule bodies contained further literals besides the common literal which determines the set of values for the variables `Points` and `Name`.

Of course, in general, the special treatment of the list-valued positioning argument should not be done only for the predicate `output`. Thus, the compiler should try to determine intervals of possible values for ordering arguments. If the intervals for two rules about a predicate do not overlap, the sequence of rule applications is again clear, and no merging at runtime is needed.

4.6 Avoiding List Construction, Example

One final point for the efficient evaluation of the `output`-rules is that the positioning argument is seldom explicitly needed, and one wants of course to avoid constructing the list terms if at all possible. In most cases, computing tuples in the right sequence and knowing intervals for the possible values is all that is needed. This is especially clear if the lists appear only in the rules about `output`, or in rules about predicates which have a unique position in the final output (i.e. no explicit sorting or merging is needed for the resulting tuples).

Putting all together, we arrive at the natural program for the example shown in Figure 7. Of course, for `output`, we do not generate a cursor, but directly print the second argument of the derived facts.

5 Conclusions

In this paper, we made a very simple proposal for specifying output declaratively in deductive databases. An important feature in these languages is that one thinks from right to left, i.e. in the natural direction of the rules (corresponding to bottom-up evaluation). Therefore output, updates, and other results of program execution seem to be specified most natural in the rule heads. Furthermore, it seems normal that literals have several solutions, which all need to be printed — no backtracking should be needed for this (backtracking is not really a concept of bottom-up evaluation: it is always implicitly set-oriented).

Although in its pure form, specifying output in this way is probably no fun, with

```
main()
{
    print("<table>\n");
    print("<tr><th>Points</th><th>Student</th></tr>\n");
    homework_cursor_2_1 h;
    h.open();
    while(h.fetch()) {
        print("<tr><td>");
        print(h.col_2()); // Points
        print("</td><td>");
        print(h.col_1()); // Name
        print("</td></tr>");
    }
    h.close();
    print("</table>\n");
}
```

Fig. 7. Implementation for the Example from Figure 1

the proposed syntactic abbreviations, especially patterns, it seems quite reasonable. Of course, more can and should be done for special output problems (e.g., currently, putting a comma between list elements, but not at the end needs a self join to check whether there is still another element: this should be simplified).

In the second half of the paper, we showed that the rules can often be evaluated without explicit sorting: In the end, the proposed solution should be as efficient as a standard, imperative program (regarding output). Therefore, it is good to see that the declarative specification can be translated into a C++ program, which does only as much sorting as the data really requires and any program must do.

The web page `http://www.informatik.uni-halle.de/~brass/output/` contains a small prototype program that does bottom-up evaluation and computes the output (currently without syntactic abbreviations, but that will be added soon) and a few examples for interesting loop structures.

References

Becket, R. 2010. Mercury tutorial. Tech. rep., University of Melbourne, Dept. of Computer Science.
`http://www.mercury.csse.unimelb.edu.au/information/papers/book.pdf`.

Brass, S. 2000. SLDMagic — the real magic (with applications to web queries). In *First International Conference on Computational Logic (CL'2000/DOOD'2000)*, W. Lloyd et al., Eds. Number 1861 in LNCS. Springer, Heidelberg, Berlin, 1063–1077.

Brass, S. 2010. Implementation alternatives for bottom-up evaluation. In *International Conference on Logic Programming (ICLP'10): Technical Communications*. LIPIcs. Schloss Dagstuhl. `http://drops.dagstuhl.de/opus/volltexte/2010/2582/`.

Hill, P. M. and Lloyd, J. W. 1994. *The Gödel Programming Language*. MIT Press, Cambridge, Massachusetts.

Peyton Jones, S. 2000. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign language calls in haskell. In *Engineering theories of software construction*, T. Hoare, M. Nroy, and R. Steinbruggen, Eds. IOS Press, Amsterdam, 47–96. Updated version available at: `http://research.microsoft.com/en-us/um/people/simonpj/papers/marktoberdorf/`.

Wadler, P. 1997. How to declare an imperative. *ACM Computing Surveys 29,* 3, 240–263. See also: `http://homepages.inf.ed.ac.uk/wadler/topics/monads.html`.