An Abstract Machine for Push Bottom-Up Evaluation [with Declarative Output]

Stefan Brass

University of Halle, Germany

Declare 2017 (INAP: Int. Conf. on Applications of Declarative Programming and Knowledge Management)

Contents

Datalog: Motivation

- Datalog ist a very pure and simple version of the logic programming language Prolog.

  But programming in Datalog feels different: The rules are applied "bottom-up" to derive new facts from already known facts, whereas Prolog unfolds predicate calls. Rule and body literal order are irrelevant. Termination is guaranteed.

- Datalog ist more than SQL plus recursive views:
  It permits declarative programming.

  Of course, Datalog can be used for database queries. But it can also be used for other computations on large data sets, e.g. static analysis of programs.

- Declarative languages are good because they are
  - compact + simple (faster to program, even for non-experts),
  - logic-based (easier to verify),
  - not tied to a specific execution model (e.g., cloud computing).

The Push Method (1)

- We (Heike Stephan and I) have developed a method for bottom-up evaluation of Datalog ("Push Method").

  Actually, we just reinvented it: If formulated with procedures, it is very similar to a method proposed by Heribert Schütz in his PhD-thesis (1993). The original version of our method looked very different, it used C++ only as a portable assembler and did a lot of partial evaluation (which the procedure version does not have, but the C++ compiler does some optimizations which might give something similar). There are differences in the details, and we did very encouraging performance tests with benchmarks from the OpenRuleBench collection (which was missing in the earlier work).

- The idea is to immediately use a derived fact and "push" it to rules where it matches a body literal.

  Similar to seminaive evaluation with a delta consisting of one fact only.

- Backtracking is done if there are several usages of a fact.

The Push Method (2)

- The Push method reduces copying of data values and materialization of tuples.

  Materialization of tuples is needed for duplicate checks and for rules containing more than one body literal with a derived predicate.

  However, our SLDMagic transformation produces mostly "linear rules".

- Example:

```
p(X, Z) :- q(X, Y), r(Y, Z).
```

  - Suppose we derived a new fact `q(a, b)`.
  - `r` is a database predicate with many facts `r(b, Z)`.

  There is no need to copy the value of `X` for each value of `Z`!

- Keeping the set of actively used values small makes better use of registers and caches. Memory is recycled earlier.

  Push methods have become attractive also for classical relational DBMS.

Our Deductive Database Project

- The method (before the abstract machine) translates Datalog rules to C++, compiles the result, and links it with a library for main-memory relations and an efficient data loader.

  The first prototype produces only the main code part. Some manual work is needed to embed it into a class frame. This is sufficient for benchmark tests, but not for real applications. Also the library still has some restrictions. The main program (using the computed facts) must be written by the user.

- The resulting binary program can be executed many times for different input data (database states).
- A new version Datalog-to-C++ compiler is being developed produces procedures (instead of the low-level switch).
  - A substantial part of it is written in Datalog.
  - It also uses "templates" for declarative output.

Advantages of Abstract Machine

- Most Prolog-Systems started with an abstract machine, some later added compilation to native machine code.

  Typical speedup: Factor 3 (a paper by Costa reported between 1.3 and 5.6).

- Why bother with abstract machine if we have native code?
  - Better comparison of benchmark results.
  - User does not need to install C++ compiler.
  - Reduction of compilation time.

    Especially useful for development and debugging.
  - Better control over optimizations (partial evaluation again!).

    Maybe first step to direct native code generation via LLVM library.
  - User must trust only the emulator, not the program.
  - Easier distribution of programs for multiple platforms.

Contents

Example: Input Program

- Transitive closure of a given binary relation "edge":

```
db edge(int, int). % Declaration of DB predicate
path(X, Y) :- edge(X, Y).
path(X, Z) :- edge(X, Y) ∧ path(Y, Z).
```

- In addition, one must specify the "query predicates", for which the derivable facts are stored.

  In this case, it is `path`. Previously, we used a fixed predicate called "`answer`". With output templates, all predicates used there become query predicates.

- We developed a fast data loader for facts. It will read the input file with a large number of `edge`-facts.

  Such as "`edge(1, 2).`" In future, we will develop also a data loader for CSV-data, and possibly loaders for other formats (JSON, RDF: Turtle).

Example: Data Structures

- Main memory relations are used for storing sets of tuples with different access patterns:

ID	Relation	Comment
0	`edge_ff`	Use of edge in first rule (list)
1	`edge_fb`	Use of edge in second rule (multimap)
2	`path_bb`	For duplicate check (set)
3	`path_ff`	Result (list)

The suffix is a binding pattern, e.g. `fb` means that the first argument is "free" (output), and the second argument is "bound" (input). For an element test, both arguments are bound, for a full table scan, both are free.

- Cursors are used for iterating over sets of tuples:

ID	Relation	Comment
0	`edge_ff`	`edge(X,Y)` in rule 1 (full scan)
1	`edge_fb`	`edge(X,Y)` in rule 2 (with given `Y`)

Example: Procedure "start"

```
void start() {
    // path(X, Y) :- edge(X, Y):
    cur_2_c lit_1(&edge_ff);
    lit_1.open();
    // Loop over (X,Y) with edge(X,Y):
    while(lit_1.fetch()) {
        int X = lit_1.col_1();
        int Y = lit_1.col_2();
        // Call procedure for new path-fact:
        path(X, Y);
    }
    lit_1.close();
}
```

Example: Procedure "path"

```
void path(int c1, int c2) {
    if(!path_bb.insert(c1, c2))// Duplicate Check
        return;
    path_ff.insert(c1, c2);    // Store answer

    // path(X, Z) :- edge(X, Y), path(Y, Z):
    int Y = c1;                // Call matches
    int Z = c2;                //    body literal
    cur_1_1_c lit_1(&edge_fb); // Cursor over edge_fb
    lit_1.open(Y);             // Y is known input
    while(lit_1.fetch()) {     // Loop over edge(X,Y)
        int X = lit_1.out_1(); // X is output
        path(X, Z);            // Call for rule head
    }
}
```

Contents

Memory Areas of Abstract Machine

- Code Area (machine instructions)
- Program Counter (Instruction Pointer)
- Registers (variables for data values: integers)
- Stack (return addresses, saved registers/cursor states)
- String Data (mapped to unique integers)
- Relations
- Cursors
- Load Specifications (for data loader)

Design Principles (1)

- Parameter values are passed in registers, not on the stack.

  Only values that are overwritten in recursions are saved onto the stack and later restored.

- It is not required that the first parameter is in register 1.

  There can be several specializations of the same predicate procedure depending on where the parameters are stored or whether values of the parameters are known constants.

- E.g. a rule like

  ```
  p(Y, X) :- q(X, Y), ...
  ```

  would not require to swap the data values.

Design Principles (2)

- Instructions should be compact (more cache-friendly):
  - This implies variable length,
  - special instructions for small operands.
- Special instructions for common cases
  (e.g. for accessing relations with only one or two columns).

  This means that more constants are compiled into the code and loops unrolled.
  The general case needs more work at runtime.
- The granularity of the operations should be big,
  e.g. entire loop control in one instruction.

  This reduces the interpretation overhead.
  (Of course, there must be an instruction at the begin and at the end of a loop, but for each iteration, only one is executed.)

Example: Procedure "start"

```
// Procedure start: path(X, Y) :- edge(X, Y).
 0: LOOP_LIST_2(0, 17)      // Loop over edge_ff
 4: GET_LIST_2_COL_1(0, 0) // R0 = X    (Cursor 0)
 7: GET_LIST_2_COL_2(0, 1) // R1 = Y
10: CALL(18)               // Call path(R0, R1)
13: END_LOOP_LIST_2(0, 4)  // If more edges goto 4
17: HALT                   // End of main procedure
```

- The LOOP_LIST_2 instruction opens cursor 0 (over edge_ff) and fetches the first tuple. If there is none, the cursor is closed and execution jumps to the instruction at address 17.

  The suffix 2 means that this is a special instruction for lists of rows with two columns.

Example: Procedure "path"

```
// Procedure path(R0, R1):
18: DUPCHECK_2(2, 0, 1)        // If(R0,R1)∈path_bb: ret.
21: INSERT_LIST_2(3, 0)        // Store result: path_ff
// path(X, Z) :- edge(X, Y), path(Y, Z).
24: SAVE_VAR(0)                // Will change R0
26: SAVE_MMAP_1_1_CUR(1)       // Will change cursor 1
28: LOOP_MMAP_1_1(1, 43, 0)    // edge(X,Y) given Y=R0
33: GET_MMAP_1_1_OUT_1(1, 0)   // Store X in R0
36: CALL(18)                   // Rec. call: path(R0,R1)
39: END_LOOP_MMAP_1_1(1, 33)   // If more edges: goto 33
43: RESTORE_MMAP_1_1_CUR(1)    // Restore used cursor
45: RESTORE_VAR(0)             // Restore R0
47: RETURN                     // End of procedure path
```

- Note: `Z` (= `R1`) is not copied.

Contents

Benchmark Results (1)

- Transitive Closure Benchmark `path(_,_)`, 50 000 edge-facts:

System	Total time	Factor	Memory
Push (Switch)	1.147s	1.0	24 MB
Push (Proc.)	1.177s	1.0	31 MB
Push (Abstr.M.)	1.713s	1.5	31 MB
Seminaïve	2.227s	1.9	31 MB
XSB	5.103s	4.4	136 MB
YAP	10.840s	9.5	148 MB
DLV	51.660s	45.0	514 MB
Soufflé (SQlite)	11.240s	9.8	43 MB
Soufflé (compiled)	0.797s	0.7	4 MB

Soufflé (compiled) uses a trie data structure (might explain small memory and slightly better performance). We restricted it to one thread.

Benchmark Results (2)

- Join1 Benchmark with query `a(X,Y)`:

```
a(X, Y)  :- b1(X, Z), b2(Z, Y).
b1(X, Y) :- c1(X, Z), c2(Z, Y).
b2(X, Y) :- c3(X, Z), c4(Z, Y).
c1(X, Y) :- d1(X, Z), d2(Z, Y).
```

The EDB predicates c2, c3, c4, d1, d2 have 10.000 rows each.

System	Load	Execution	Total Time	Factor
Push (Proc.)	0.004s	1.043s	1.043s	1.0
Push (Switch)	0.004s	1.031s	1.032s	1.0
Push (Abstr.M.)	0.004s	1.496s	1.498s	1.5
XSB	0.128s	6.056s	6.460s	6.2
YAP	0.207s	3.572s	3.840s	3.7
DLV	(0.253s)	—	80.237s	76.9
Soufflé (SQlite)	(0.100s)	—	12.680s	12.2
(compiled)	(0.040s)	—	1.450s	1.4

Contents

Conclusions

- The instructions of the abstract machine are basically defined by identifying the building blocks of the previously generated C++ code.

  This combines the partial evaluation we used in the older approach with the procedures from the newer version. But there are many details to think about.

- It seems that the overhead for interpretation of "machine instructions" is not big.

  This supports our performance claims in comparison with WAM-based systems.

- XSB has beaten older systems based on bottom-up evaluation (such as Coral). Now it seems that bottom-up evaluation can be implemented in a competitive way.

  To be fair, we still have no complete system, only enough code to execute some benchmarks. XSB has been developed over decades and is used in applications. But our results show that we are working in a promising direction.

Future Work

- The compiler from Datalog to C++ will soon be ready.

  Datalog and output templates are used for the implementation.

- Then there should be a version the generates code for the abstract machine presented here.

  Simple first cut: Only exchange output templates (much of the computed Datalog facts describing the computation are the same). However, the register assignment is something missing in the C++ procedure version.

- We have many ideas, e.g. for more powerful templates.

  E.g. it should be possible to use the output of one template as input for another one. The separator string should be a special case of "hooks". Templates should be made conditional (alternative variants of a template). Output to different files should be supported. Char maps. Modules.

- ToDo: Negation, aggregation, arrays, keys, parallel evaluation.

Contents

Declarative Output with Templates (1)

- Output is needed for more or less every application.

  The example templates are taken from our Datalog-to-C++ compiler.

- Conceptionally, we
  - first compute the necessary data with Datalog rules, and
  - then use the facts to instantiate output templates.
- A template can be seen as a procedure (with parameters) that contains only
  - printing commands (of string constants and parameters), and
  - calls to other templates,
    possibly with an iteration over solutions for Datalog queries.

    One can define the sequence with a kind of "`ORDER BY`" clause, and

    a separator string to be inserted between each two template instantiations.

Declarative Output with Templates (2)

- Example (constructor of a class for representing tuples):

```
rel_class_constructor(Pred): [
    '// Constructor:' nl
    rel_class(Pred) '('
        constructor_arg(ArgNo,Type)<ArgNo>+', ' :-
            constructor_arg(Pred, ArgNo, Type).
    ') {' nl
        constructor_assign(ArgNo)<ArgNo> :-
            arg_type(Pred, ArgNo, Type).
    '}' nl
].
```

`<ArgNo>` means that the calls of the head template are sorted by the value of the variable `ArgNo`.

`+', '` specifies that the string `', '` is inserted between each two such calls.

Declarative Output with Templates (3)

- Besides this "code mode", there is also a "verbatim mode" (started and ended with "|"), where every character besides "|" and "[" is copied to the output.

  "[...]" used for switching to the standard "code mode", e.g. for inserting parameter values or template calls.

- Example:

```
constructor_assign(ArgNo): [
|       this->col_[ArgNo]= arg_[ArgNo];
|].
```

- Termination is ensured because templates cannot call themselves recursively.