On Compiling Linear Logic Programs with Comprehensions, Aggregates and Rule Priorities

Flavio Cruz[1,2] and Ricardo Rocha[2]

[1] Carnegie Mellon University, Pittsburgh, PA 15213, USA
fmfernan@cs.cmu.edu
[2] CRACS & INESC TEC and Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
{flavioc, ricroc}@dcc.fc.up.pt

Abstract. Linear logic programs are challenging to implement efficiently because facts are asserted and retracted frequently. Implementation is made more difficult with the introduction of useful features such as rule priorities, which are used to specify the order of rule inference, and comprehensions or aggregates, which are mechanisms that make data iteration and gathering more intuitive. In this paper, we describe a compilation scheme for transforming linear logic programs enhanced with those features into efficient C++ code. Our experimental results show that compiled logic programs are less than one order of magnitude slower than hand-written C programs and much faster than interpreted languages such as Python.

1 Introduction

Linear Meld (LM) is a linear logic programming language aimed for the parallel implementation of graph-based algorithms [2]. LM is a high-level declarative language that offers a concise and expressive framework to define graph based algorithms that are provably correct. LM has been applied to a wide range of problems and machine learning algorithms, including: belief propagation [6], belief propagation with residual splash [6], PageRank, graph coloring, N-Queens, shortest path, diameter estimation, map reduce, quick-sort, neural network training, minimax, and many others.

Like Datalog, LM is a *forward-chaining* logic programming language since computation is driven by a set of inference rules that are used to update a database of logical facts. In Datalog, programs are monotonic and therefore the database grows in size as more facts are inferred from the logical rules. In LM, logical facts are linear and thus can be retracted when a rule is inferred. The use of linear facts greatly increases the power of the language but also increases the complexity of the implementation since database facts are retracted often.

In previous work [3], we have described the implementation of the LM virtual machine, including its data structures and how programs are parallelized. In this paper, we describe our compilation strategy and how we have refitted the

runtime system to allow stand-alone compilation of programs by transforming logical rules into C++ code.

Our goal was to reduce the overhead of executing interpreted byte code and better understand the effectiveness and limitations of the compilation scheme. We present an algorithm that compiles logical rules, including comprehensions and aggregates, into efficient iterator-based C++ code. The compiler supports rule priorities, allowing the programmer to order rules based on their priority of inference. To the best of our knowledge, this is the first available compilation strategy for a linear logic language that supports these 3 features combined. The contributions of this paper are then three-fold: (1) a novel algorithm to compile prioritized linear logic rules with aggregates and comprehensions; (2) the interplay between the database layout and compiled code; and (3) comparison and analysis of our compilation with hand-written C programs and interpreted code. Experimental results show that our compiled programs are only 1 to 5 times slower than hand-written C programs.

The remainder of the paper is organized as follows. First, we briefly introduce the LM language. Next, we present an overview of the runtime support available to compiled rules and we discuss our contributions which include the algorithm for compiling rules into efficient iterator-based C++ code, and related work. We then present experimental results comparing our compiled programs with the old implementation and with hand-written C programs. The paper finishes with some conclusions.

2 Linear Meld

LM is a forward-chaining linear logic programming language that allows logical facts to be asserted and retracted in a structured fashion. A LM program can be seen as a graph of nodes, where each node contains a database of facts. The program is written as a set of inference rules that apply over the facts of a node.

LM rules have the form `a(X), b(Y) -o c(X, Y)` and can be read as follows: if fact `a(X)` and fact `b(Y)` exist in the database then fact `c(X, Y)` is added to the database. The expression `a(X), b(Y)` is called the *body* of the rule and `c(X, Y)` is called the *head* of the rule. A fact is a predicate, e.g., `a`, `b` or `c`, and its associated tuple of values, e.g., the concrete values of `X` and `Y`. Since LM uses linear logic as its foundation, we distinguish between *linear* and *persistent facts*. Linear facts are consumed (deleted) during the process of deriving a rule, while persistent facts are not. Program execution starts by adding the initial facts (called the axioms) to the database. Next, rules are recursively applied and the database is updated by adding new facts or deleting facts used during rule derivation. When no more rules are applicable, the program terminates. Rules have a defined priority (their position in the source file) and highest priority rules are fired first. If a new fact is derived and there is a set of applicable rules to be fired, the higher priority rule is selected before the others.

To make these ideas concrete, Fig. 1 presents a simple example for the single source shortest path (SSSP) program. The program computes the shortest dis-

```
1   type route edge(node, node, int).
2   type linear shortest(node, int, list int).
3   type linear relax(node, int, list int).
4
5   !edge(@1, @2, 3). !edge(@1, @3, 1).
6   !edge(@3, @2, 1). !edge(@3, @4, 5).
7   !edge(@2, @4, 1).
8   shortest(A, +00, []).
9   relax(@1, 0, [@1]).
10
11  shortest(A, D1, P1), D1 > D2, relax(A, D2, P2)
12     -o shortest(A, D2, P2),
13        {B, W | !edge(A, B, W) | relax(B, D2 + W, P2 ++ [B])}.
14
15  shortest(A, D1, P1), D1 <= D2, relax(A, D2, P2)
16     -o shortest(A, D1, P1).
```

Fig. 1: Single Source Shortest Path program code.

tance from node `@1` to all other nodes in the graph. The SSSP program starts (lines 1-3) with the declaration of the predicates. Predicates specify the facts used in the program. The first predicate, `edge`, is a persistent predicate that describes the relationship between the nodes of the graph, where the third argument represents the weight of the edge (the `route` modifier informs the compiler that the `edge` predicate determines the structure of the graph). The predicates `shortest` and `relax` are specified as linear facts and thus are deleted when deriving new facts. In the example, every node has a `shortest` fact that can be improved with new `relax` facts. Lines 5-9 declare the axioms of the program: `edge` facts describe the graph; `shortest(A, +00, [])` is the initial shortest distance (infinity) for all nodes; and `relax(@1, 0, [@1])` starts the algorithm by setting the distance from `@1` to `@1` to be 0.

The first rule of the program (lines 11-13) reads as follows: if the current `shortest` path `P1` with distance `D1` is larger than a new `relax` path with distance `D2`, then replace the current shortest path with `D2`, delete the new `relax` and propagate new paths to the neighbors (line 13) using a *comprehension*. The comprehension iterates over the edges of node `A` and derives a new `relax` fact for each node `B` with the distance `D2 + W`, where `W` is the weight of the edge.

The second rule of the program (lines 15-16) is read as follows: if the current shortest distance `D1` is shorter than a new `relax` distance `D2`, then delete the new `relax` fact and keep the current shortest path. Figure 2 shows a graphical representation of the application of the SSSP program rules.

2.1 LM Syntax

The abstract syntax for LM programs is presented in Fig. 3. A LM rule is written as $BE \multimap HE$ where BE is the body and HE is the head of the rule. The body may contain linear (L) and persistent (P) *fact expressions* and *constraints* (C). Fact expressions instantiate facts from the database and contain variables as arguments that may or may not be bound to concrete values or to other variables. Variables in the body of the rule can also be used in the head when instantiating

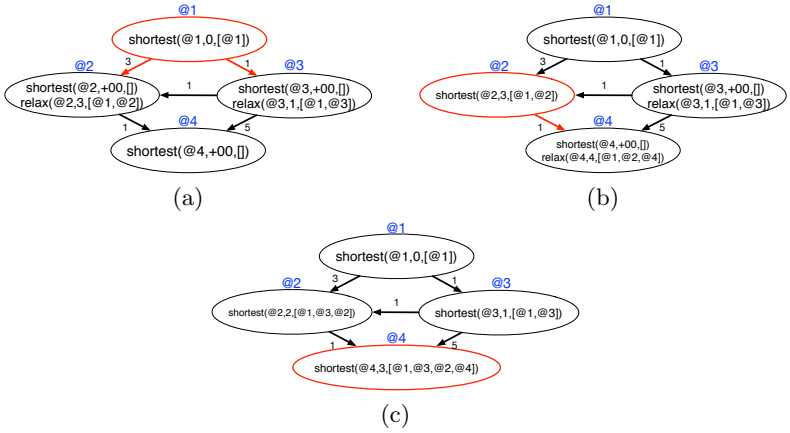

Fig. 2: Graphical representation of the SSSP program: (a) represents the program after propagating the initial distance at node `@1`, followed by (b) where the first rule is applied in node `@2` and by (c) that represents the final state of the program, where all the shortest paths have been computed.

Program	$Prog$	$::= \Sigma, D$
List Of Rules	Σ	$::= \cdot \mid \Sigma, R$
Database	D	$::= \Gamma; \Delta$
Rule	R	$::= BE \multimap HE \mid \forall_x.R$
Body Expression	BE	$::= L \mid P \mid C \mid BE, BE \mid \mathbf{1}$
Head Expression	HE	$::= L \mid P \mid HE, HE \mid EE \mid CE \mid AE \mid \mathbf{1}$
Linear Fact	L	$::= l(\hat{x})$
Persistent Fact	P	$::= !p(\hat{x})$
Constraint	C	$::= c(\hat{x})$
Selector Operation	S	$::= \texttt{min} \mid \texttt{max} \mid \texttt{random}$
Comprehension	CE	$::= \{\ \widehat{x};\ SB;\ SH\ \}$
Aggregate	AE	$::= [\ A \Rightarrow y;\ \widehat{x};\ SB;\ SH_1;\ SH_2\ ]$
Aggregate Operation	A	$::= \texttt{min} \mid \texttt{max} \mid \texttt{sum} \mid \texttt{count} \mid \texttt{collect}$
Sub-Body	SB	$::= L \mid P \mid SB, SB \mid \exists_x.SB$
Sub-Head	SH	$::= L \mid P \mid SH, SH \mid \mathbf{1}$
Known Linear Facts	Δ	$::= \cdot \mid \Delta, l(\hat{t})$
Known Persistent Facts	Γ	$::= \cdot \mid \Gamma, !p(\hat{t})$

Fig. 3: Abstract syntax of LM.

facts. Constraints are essential for matching rules since they represent database *joins* and database *selects*. While selects filter out possible combinations from the database, body constraints (C) further restrict combinations by acting as guards using small variables from fact expressions. Constraints use a small functional language that includes mathematical operations, boolean operations, external functions and literal values.

The head of a rule, HE, contains linear (L) and persistent (P) *fact templates* which are uninstantiated facts and will derive new facts. The head can also have *comprehensions* (CE) and *aggregates* (AE). All those expressions may use all the variables instantiated in the body.

Comprehensions are similar to the functional programming construct of the same name. Comprehensions are sub-rules that are applied for all possible combinations. In a comprehension $\{\ \widehat{x};\ SB;\ SH\ \}$[3], $\widehat{x}$ is a list of variables, SB is the body of the comprehension and SH is the head. The body SB is used to generate all possible combinations for the head SH, according to the facts in the database. An example was shown in Fig. 1 (line 13), where `!edge(A, B, W)` facts are iterated over in order to derive `relax(A, D2 + W, P2 ++ [B])` facts for each combination.

Aggregates build on top of comprehensions and allow the capture of values that appear in each combination of the sub-rules. This list of values is then combined using one operator into a single value and then used to derive a set of fact expressions. In the abstract syntax $[\ A \Rightarrow y;\ \widehat{x};\ SB;\ SH_1;\ SH_2\ ]$, A is the aggregate operation, $\widehat{x}$ is the list of variables introduced in BE and SH_1 and y is the variable in the body SB that represents the values to be aggregated using A. Like comprehensions, we use $\widehat{x}$ to try all the combinations of SB, but, in addition to deriving SH_1 for each combination, we aggregate the values represented by y into a new y variable that is used inside the head SH_2. LM provides several aggregate operations, including the `min` (minimum value), `max` (maximum value), `sum` (add all numbers), `count` (count combinations) and `collect` (collect items into a list). Consider, for example, the following rule:

```
const P = ... // number of nodes
const damp = ... // probability of random jump to another page (for PageRank computation)

update(A), pagerank(A, OldRank)
      -o [sum => V | B | neighbor-pagerank(A, B, V) | neighbor-pagerank(A, B, V) |
            pagerank(A, damp/P + (1.0 - damp) * V)].
```

The rule uses an aggregate to accumulate the sum of the neighbor's PageRank into a single value `V`. This aggregate value is then assigned to a new `pagerank` fact via the expression `damp/P + (1.0 - damp)*V`, where `V` is the result of adding all the `V` values in `neighbor-pagerank(A, B, V)` facts.

3 Supporting Runtime and Database Data Structures

In this section, we review the supporting runtime that is used by the compiler. We focus mostly on the structure of the nodes since inference rules are compiled from the point of view of the node data structure.

Figure 4 presents the layout of the node data structures. Each node of the graph stores 4 main data structures: (1) the *rule matching engine*; (2) a *fact buffer* for storing incoming and temporary facts; (3) the *database of linear facts*; and (4) the *database of persistent facts*.

[3] We substitute ; for | in the abstract syntax to avoid confusion with the grammar choice operator.

The rule engine maintains a simplified view of the two fact databases and efficiently decides which rules need to be executed. For instance, if a rule r needs facts `a` and `b` to be applied and the database already contains `a` facts, once a `b` fact is derived, the rule engine schedules r to be executed. The compiler is responsible for the code that is executed when a rule is scheduled. A compiled rule contains instructions to search and match facts from the database and to derive new facts when the body of the rule is matched.

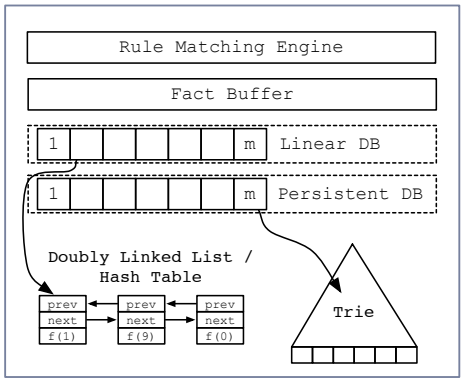

Fig. 4: Node data structures.

In this context, the organization of the database structures is critical because linear facts can be retracted and asserted frequently. This means that the database needs to allow fast insertions and deletions but also needs to have reasonably fast mechanisms for lookup. The database of facts is partitioned by predicate, therefore, each predicate can have its own data structure depending on the patterns of access for that particular predicate. Linear facts are stored using the following data structures:

- *Doubly-Linked List Data Structures.* Each linear fact is a node of the linked list. Allows constant $\mathcal{O}(1)$ insertion and deletion of facts given the pointer of the target node. Although lookup operations take linear time, this is not critical since most predicates tend to have a small number of facts.
- *Hash Table Data Structures.* For predicates with many facts we use hash tables. Hash tables are efficient for repetitive lookup operations using a specific argument (i.e., searching for facts with a concrete value) and build upon lists by hashing facts using a specific argument and then using separate chaining with doubly-linked lists for collision resolution. Hash tables are, on average, $\mathcal{O}(1)$ for insertion, deletion and lookup, however they require more memory.

For persistent tuples, we use *Trie Data Structures*, which are trees where facts are indexed by a common prefix. Since persistent facts are never deleted, it's not expensive to index facts by a common prefix, which also tends to save memory in the long run.

4 Compiling Rules

In this section, we present the main algorithm of the compiler, that turns inference rules into C++ code, and we discuss the key optimizations for efficient code execution.

4.1 Constraints

After an inference rule is compiled, it must respect the *fact constraints* (facts must exist in the database) and the *join constraints* that can be represented by variable constraints and/or boolean expressions. For instance, consider again the second rule of the SSSP program presented in Fig. 1:

```
shortest(A, D1, P1), D1 <= D2, relax(A, D2, P2)
   -o shortest(A, D1, P1).
```

The fact constraints include the facts required to trigger the rule, namely `shortest(A, D1, P1)` and `relax(A, D2, P2)`, and the join constraints include the expression `D1 <= D2`. However, rules may also have other less obvious join constraints, such as variable constraints, as in the following rule:

```
new-neighbor-pagerank(A, B, New),
neighbor-pagerank(A, B, Old)
   -o neighbor-pagerank(A, B, New).
```

where variable `B` must have the same value in both body facts[4].

4.2 Iterators

The data structures for facts presented in Section 3 support the *iterator* pattern. For linked lists, the iterator goes through every fact in the list while the hash table iterator can either iterate through the whole table or iterate through a single bucket. A bucket iterator is in fact a linked list iterator that starts from a given argument. For tries, while the default iterator goes through every fact in the trie, it can be customized with a matching specification in order to reduce search. A matching specification includes argument assignments (e.g., argument $i = V$, where V is a concrete value).

Iterators are heavily used in the compiled code. For instance, the second rule of the SSSP program presented in Fig. 1 is compiled as follows:

```
1   for(auto it1(list("shortest").begin()); it1 != list("shortest").end(); ) {
2      fact *f1(*it1);
3      for(auto it2(list("relax").begin()); it2 != list("relax").end(); ) {
4         fact *f2(*it2);
5         if(f1->get_int(1) <= f2->get_int(1)) { // D1 <= D2
6            fact *new_shortest(new fact("shortest"));
7            new_shortest->set_int(1, f1->get_int(1));
8            new_shortest->set_list(2, f1->get_list(2));
9            // new fact was derived
10           list("shortest").push_back(new_shortest);
11           // deleting facts
12           it1 = list("shortest").erase(it1); // remove from list
13           it2 = list("relax").erase(it2);
14           return;
15        }
16        ++it2;
17     }
18     ++it1;
19  }
```

[4] Rule taken from an asynchronous PageRank program.

The compilation algorithm iterates through the fact expressions in the body of the rule and creates nested loops to try all the possible combinations of facts. For this rule, all pairs of `shortest` and `relax` facts must be matched until the constraint `D1 <= D2` is true. First, an iterator for `shortest` is created that will loop through all `shortest` facts in the list. Inside the loop, a nested iterator is created for predicate `relax`. This inner loop includes a check for the `D1 <= D2` constraint. If the constraint fails, another `relax` fact is then attempted by incrementing `it2`. Likewise, if the current `f1` fact fails for all `f2` facts, then `it1` is incremented in order to try the next `shortest` fact. Otherwise, if the constraint succeeds then the rule matches and a new `shortest` fact is derived. Additionally, the two used linear facts are retracted by erasing the iterators from the linked lists. Note that after the rule is derived, the code must return since there is a higher priority rule that may be triggered with the new `shortest` fact (see Fig. 2). This enforces the priority semantics of the language.

Figure 5 presents the algorithm for compiling rules into C++ code. First, we split the body of the rule into fact expressions and constraints. Fact expressions map directly to iterators while fact constraints map to *if* expressions. A possible compilation strategy is to first compile all the fact expressions and then compile the constraints. However, this may require unneeded database lookups since some constraints may fail early. Therefore, our compiler introduces constraints as soon as all the variables in the constraint are all included in the already compiled fact expressions. The order in which fact expressions are selected for compilation does not interfer with the correctness of the compiled code, thus our compiler selects the fact expressions ($RemoveBestFactExpr$) by their potential to activate constraints, therefore avoiding undesirable database lookups. If two fact expressions have the same number of new constraints, then the compiler always picks the persistent fact expression since persistent facts are not deleted.

Derivation of new facts belonging to the local node implies adding the new fact to the local node data structure. Facts that belong to other nodes are sent using an appropriate runtime API.

4.3 Persistence Checking

Not all linear facts need to be deleted. For instance, in the compiled rule above, the fact `shortest(A, D1, P1)` is re-derived in the head. Our compiler is able to turn linear loops into persistent loops for linear facts that are retracted and then asserted. The rule is then compiled as follows:

```
1  for(auto it1(list("shortest").begin()); it1 != list("shortest").end(); ) {
2     fact *f1(*it1);
3     for(auto it2(list("relax").begin()); it2 != list("relax").end(); ) {
4        fact *f2(*it2);
5        if(f1->get_int(1) <= f2->get_int(1)) {
6           it2 = list("relax").erase(it2);
7           goto next;
8        }
9        ++it2;
10 next: continue;
11    }
12    ++it1;
13 }
```

```
Data: Rule R1, Rules
Result: Compiled Code
FactExprs ⟵ FactExprsFromRule(R1);
Constraints ⟵ ConstraintsFromRule(R1);
Code ⟵ CreateFunctionForRule();
Iterators ⟵ [];
CompiledFacts = [];
while FactExprs not empty do
    Fact ⟵ RemoveBestFactExpr(FactExprs);
    CompiledFacts.push(Fact);
    Iterator ⟵ Code.InsertIterator(Fact);
    Iterators.push(Iterator);
    /* Select constraints that are covered by CompiledFacts.       */
    NextConstraints ⟵ RemoveConstraints(Constraints, CompiledFacts);
    Code.InsertConstraints(NextConstraints);
end
HeadFacts = HeadTemplatesFromRule(R1);
while HeadFacts not empty do
    Fact ⟵ RemoveFact(HeadFacts);
    Code.InsertDerivation(Fact);
end
for Iterator ∈ Iterators do
    if IsLinear(Iterator) then
        Code.InsertRemove(Iterator);
    end
end
/* Enforce rule priorities.                                        */
if FactsDerivedUsedBefore(Head, Program, R1) then
    Code.InsertReturn();
else
    Code.InsertGoto(FirstLinear(Iterators));
end
return Code
```

Fig. 5: Compiling LM rules into C++ code.

In this new version of the code, only the `relax` facts are deleted, while the `shortest` facts remain untouched. In the SSSP program, each node has one `shortest` fact and this compiled code simply filters out the `relax` facts with the distances that are equal or greater than the current best distance. Note that now we have a *goto statement* (line 7) that is executed when the rule is fired. In this case, since no new `shortest` fact was derived, we avoid returning to enforce rule priorities and continue to try to fire the rule as many times as possible.

All the rule combinations are attempted in cases where a rule does not derive any facts or the facts derived do not appear before the rule, that is, the new facts are only used in lower priority rules. This is specified in the final *if statement* in Fig. 5. If the rule does not return, then we always jump to the first loop that

uses linear facts. We must jump to the first linear loop because we cannot use the next fact from the deepest loop since we may have constraints between the first linear loop and the deepest loop that were previously validated using facts that were deleted in the meantime.

4.4 Updating Facts

Many inference rules retract and then derive the same predicate but with different arguments. The compiler recognizes those cases and instead of retracting the fact from its linked list or hash table, it updates the fact in-place. As an example, consider the following rule:

```
new-neighbor-pagerank(A, B, New),
neighbor-pagerank(A, B, Old)
   -o neighbor-pagerank(A, B, New).
```

Assuming that `neighbor-pagerank` is stored in a hash table and indexed by the second argument, the code for the rule above is as follows:

```
1   for(auto it1(list("new-neighbor-pagerank").begin()); it1 !=
2      list("new-neighbor-pagerank").end(); )
3   {
4      fact *f1(*it1);
5      // hash table for neighbor-pagerank is indexed by the second argument therefore
6      // we search for the bucket using the second argument of new-neighbor-pagerank
7      hash_bucket bucket(hash_table("neighbor-pagerank").find(f1->get_node(1));
8      for(auto it2(bucket.begin()); it2 != bucket.end(); ) {
9         fact *f2(*it2);
10        if(f1->get_node(1) == f2->get_node(1)) {
11           f2->set_float(2, f1->get_float(2)); // update neighbor-pagerank
12           it1 = list("new-neighbor-rank").erase(it1);
13           goto next;
14        }
15        ++it2;
16     }
17     ++it1;
18  next: continue;
19  }
```

Note that `neighbor-pagerank` is updated using `set_float`. The rule also does not return since this is the highest priority rule. If there was a higher priority rule using `neighbor-pagerank`, then the code would have to return since an updated fact represents a new fact.

4.5 Enforcing Linearity

We have already introduced the *goto* statement as a way to avoid reusing retracted linear facts. However, this is not enough in order to enforce linearity of facts. Consider the following inference rule:

```
add(A, N1), add(A, N2) -o add(A, N1 + N2).
```

Using the standard compilation algorithm, two nested loops are created, one for each `add` fact. However, notice that there is an implicit constraint when

creating the iterator for `add(A, N2)` since this fact cannot be the same as the first one. That would invalidate linearity since a single linear fact would be used to prove two linear facts. This is easily solved by adding a constraint for the inner loop that ensures that the two facts are different (line 5).

```
1   for(auto it1(list("add").begin()); it1 != list("add").end(); ) {
2      fact *f1(*it1);
3      for(auto it2(list("add").begin()); it2 != list("add").end(); ) {
4         fact *f2(*it2);
5         if(f1 != f2) {
6            f1->set_int(1, f1->get_int(1) + f2->get_int(1));
7            it2 = list("add").erase(it2);
8            goto next;
9         }
10        ++it2;
11     }
12     ++it1;
13  next: continue;
14  }
```

Figure 6 presents the steps for executing this rule when the database contains three facts. Initially, the two iterators point to the first and second facts and the former is updated while the latter is retracted. The second iterator then moves to the next fact and the first fact is updated again, now to the value `6`, the expected result.

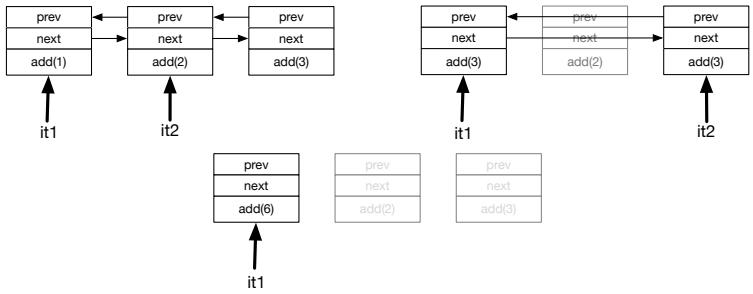

Fig. 6: Executing the add rule.

4.6 Comprehensions

Comprehensions were initially presented in the first rule of the SSSP program.

```
shortest(A, D1, P1), D1 > D2, relax(A, D2, P2)
   -o shortest(A, D2, P2), {B, W | !edge(A, B, W) | relax(B, D2 + W, P2 ++ [B])}.
```

The attentive reader will remember that comprehensions are sub-rules, therefore they should be compiled like normal rules. However, they do not need to return due to rule priorities since all the combinations of the comprehension must be derived. However, the rule itself must return if any of its comprehensions has derived a fact that is used by a higher priority rule. In the case of the

above example, the rule does not need to return since it has the highest priority and the `relax` facts derived in the comprehension are all sent to other nodes. The code for the rule is shown below:

```
for(auto it1(list("shortest").begin()); it1 != list("shortest").end(); ) {
   fact *f1(*it1);
   for(auto it2(list("relax").begin()); it2 != list("relax").end(); ) {
      fact *f2(*it2);
      if(f1->get_int(1) > f2->get_int(1)) {
         // comprehension code
         for(auto it3(trie("edge").begin()); it3 != trie("edge").end(); ) {
            fact *f3(*it3);
            fact *new_relax(new fact("relax"));
            new_relax->set_int(1, f2->get_int(1) + f3->get_int(2));
            new_relax->set_list(append(f2->get_list(2), list(f3->get_node(1))));
            send_fact(new_relax, f3->get_node(1));
            ++it3;
         }
         f1->set_int(1, f2->get_int(1));
         f1->set_list(2, f2->get_list(2));
         it2 = list("relax").erase(it2);
         goto next;
      }
      ++it2;
   }
   ++it1;
next: continue;
}
```

Special care must be taken when the comprehension's sub-rule uses the same predicates that are derived by the main rule. Rule inference must be atomic in the sense that after a rule matches, the comprehensions in the head of the rule can use the facts that were present before the body of the rule was matched. Consider a rule with n comprehensions or aggregates, where CB_i and CH_i are the body and head of the comprehension/aggregate, respectively, and H represents the fact templates found in the head of the rule. The formula used by the compiler to detect conflicts between predicates is the following:

$$\bigcup_{i}^{n}[CB_i \cap H] \cup \bigcup_{i}^{n}[CB_i \cap \bigcup_{j}^{n}[CH_j]]$$

If the result of the formula is not empty, then the compiler disables optimizations for the conflicting predicates and derives the corresponding facts into the fact buffer that are then added back into the database. Fortunately, most rules in LM programs do not show conflicts and thus can be fully optimized.

4.7 Aggregates

Aggregates are similar to comprehensions. They are also sub-rules but a value is accumulated for each combination of the sub-rule. After all the combinations are inferred, a final head term is derived with the accumulated term. Consider again the following PageRank rule:

```
update(A), pagerank(A, OldRank)
      -o [sum => V | B | neighbor-pagerank(A, B, V) | neighbor-pagerank(A, B, V) |
            pagerank(A, damp/P + (1.0 - damp) * V)].
```

The variable `V` is initialized to `0.0` and sums all the PageRank values of the neighbors as seen in the code below. The aggregate value is then used to update the second argument of the initial `pagerank` fact.

```
1   for(auto it1(list("pagerank").begin()); it1 != list("pagerank").end(); ) {
2      fact *f1(*it1);
3      for(auto it2(list("update").begin()); it2 != list("update").end(); ) {
4         fact *f2(*it2);
5         double acc(0.0); // aggregate accumulator.
6         for(auto it3(list("neighbor-pagerank").begin()); it3 !=
7               list("neighbor-pagerank").end(); ) {
8            fact *f3(*it3);
9            acc += f3->get_float(2);
10           ++it3; // the sub-rule has no head since neighbor-pagerank is re-derived
11        }
12        // head of the aggregate
13        f1->set_float(1, damp / P + (1.0 - damp) * V);
14        goto next;
15     }
16     ++it1;
17  next: continue;
18  }
```

5 Related Work

LM shares many similarities [1] with Constraint Handling Rules (CHR) [5]. CHR is a concurrent committed-choice constraint language used to write constraint solvers. A CHR program is a set of rules and a set of constraints. The constraint store can be seen as a database of facts and rules manipulate the constraint store. Many basic optimizations used in the LM compiler such as join optimizations and the use of different data structures for indexing facts were inspired in work done on CHR [7]. Wuille et al. [9] have described a CHR to C compiler that follows some of the ideas presented here and De Koninck et al. [4] showed how to compile CHR programs with dynamic priorities into Prolog. Our work distinguishes itself from these two works by supporting a novel combination of comprehensions, aggregates and rule priorities. Compilation of LM programs is also novel due to the implicit parallelism of rules, allowing for programs to be parallelized [2].

6 Experimental Results

This section presents experimental results for our compilation strategy. We compare the execution speed of our new compiled code against hand-written implementations in C of the same programs. We also compare the results against interpreted execution in order to help us understand the limitations of the compilation scheme when removing the interpretation overhead.

For our experimental setup, we used a computer with a 24 (4x6) Core AMD Opteron(tm) Processor 8425 HE @ 800 MHz with 64 GBytes of RAM memory running the Linux kernel 3.15.10-201.fc20.x86_64. The C++ compiler used is GCC 4.8.3 (g++) with the flags: `-O3 -std=c+0x -march=x86-64`. We run all experiments 3 times and averaged the execution time.

We have implemented 5 different LM programs and their corresponding C versions. The programs are the following:

- Shortest Path (SP): a slightly modified version of the program presented in Fig. 2, where the shortest distance is computed from all nodes to all nodes.
- N-Queens: the classic puzzle for placing queens on a chess board so that no two queens threaten each other.
- Belief Propagation: a machine learning algorithm to denoise images.
- Heat Transfer: an asynchronous program that performs transfer of heat between nodes.
- MiniMax: the AI algorithm for selecting the best player move in a Tic-Tac-Toe game. The initial board was augmented in order to provide a longer running benchmark.

Table 1 presents experimental results comparing the compiled and interpreted code versions against the C program versions. Comparisons to other systems are shown under the **Other** column. Note that for some programs, we present different program sizes shown in ascending order.

Program	**Size**	**C Time** (s)	**Compiled**	**Interpreted**	**Other**
Shortest Path	US Airports	0.1	3.9	13.9	13.3 (python)
	OCLinks	0.4	5.6	14.2	11.2 (python)
	Powergrid	0.9	3.5	11.3	10.6 (python)
N-Queens	11	0.2	1.4	3.9	20.8 (python)
	12	1.3	3.2	5.3	24.1 (python)
	13	7.8	3.8	6.6	26.0 (python)
	14	49	4.5	8.9	28.0 (python)
Belief Propagation	50	2.8	1.3	1.4	1.1 (GL)
	200	51	1.3	1.4	1.1 (GL)
	300	141	1.3	1.4	1.1 (GL)
	400	180	1.3	1.4	1.1 (GL)
Heat Transfer	80	7.3	4.6	9.9	-
	120	32	5.3	10.5	-
MiniMax	-	7.3	3.2	7.1	9.3 (python)

Table 1: Experimental results comparing different programs against hand-written versions in C. For the C versions, we show the execution time in seconds (column **C Time** (s)). For the other approaches, we show the overhead ratio compared with the corresponding C version. The overhead numbers (**lower is better**) are computed by dividing the execution time of the approach on that column by the execution time of the similar hand-written version in C.

The Shortest Path program shows good improvements from the interpreted version, since the run time is reduced between 61% and 72%. The good performance results come from the fact that the program performs repeated comparisons between integer numbers, which tend to be slower in interpreted code, and from the fact that the program has only two rules where the shortest distance fact is updated or kept. The distance facts are also indexed by the source node,

which helps the code filter through the candidate distances faster. This is helpful since the program computes the shortest distance between pairs of nodes.

N-Queens presents some scalability issues for our compilation scheme due to the exponential increase of facts as the problem size increases. The same behavior can be observed for the Python programs. Regarding the comparison with the interpreted version, the compiled version reduces the interpreted run time by almost 50% which indicates that there are more database operations in N-Queens than in Shortest Path.

The Belief Propagation program is made of many expensive floating point calculations. The interpreted version used external functions written in C to implement those operations because otherwise it would be too slow. Therefore, and since the rules tend to manipulate a small number of facts, the interpreted and compiled versions perform about the same. This program has also the best results which proves that the program spends a huge amount of time performing floating point calculations. For comparison purposes, we used GraphLab [8] (GC in the table), an efficient machine learning framework for writing parallel graph-based machine learning algorithms in C++. GraphLab's version of the algorithm is slightly slower than the C version.

The Heat Transfer program also performs floating point operations but in a much smaller scale than Belief Propagation. This is noticiable from the results since the slowdown is much larger than Belief Propagation. The program also needs to compute many sum aggregates, which makes the interpreted version incur in some overhead due to the integer operations.

While all the other programs perform computations on a pre-defined set of nodes, the MiniMax program creates the nodes of the graph dynamically. Creating new nodes requires creating new databases which tends to take a considerable fraction of the run time. However, we have seen a good reduction in run time when compared to the interpreted version, which we think is the result of low-level optimizations that were applied in the compiled version.

It should be noted that in these programs there is a parallelization overhead since LM's supporting runtime is designed to explore parallelism implicitly. For instance, we measured a 20% overhead for N-Queens, a program that needs to reference count many lists during run time. Fortunately, if the programmer takes advantage of the parallel facilities of LM, she will be able to run most of these programs faster than C by using between 2 and 4 threads.

7 Conclusions

In this paper, we have presented a compilation strategy for linear logic programs with comprehensions, aggregates and rule priorities. Rule priorities allow the programmer to assign priorities to rules so that higher priority rules are applied before lower priority rules, while comprehensions and aggregates allow a more expressive way for the programmer to iterate through the database to derive new facts or aggregate data. To the best of our knowledge, our compilation strategy is the first to consider programs with these three important features and the

first efficient compilation strategy for forward-chaining linear logic programs. We have also implemented and described important optimizations such as fact updates and persistence checking and the importance of choosing the right data structures for the needs of linear logic programs. Our experimental results show that LM is competitive when compared to hand-written C programs.

Acknowledgments

This work is partially funded by the ERDF (European Regional Development Fund) through the COMPETE Programme and by FCT (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program and project SIBILA (NORTE-07-0124-FEDER-000059). Flavio Cruz is funded by the FCT grant SFRH/BD/51566/2011.

References

1. Betz, H., Frühwirth, T.: A linear-logic semantics for constraint handling rules. In: Principles and Practice of Constraint Programming, Lecture Notes in Computer Science, vol. 3709, pp. 137–151 (2005)
2. Cruz, F., Rocha, R., Goldstein, S., Pfenning, F.: A Linear Logic Programming Language for Concurrent Programming over Graph Structures. Journal of Theory and Practice of Logic Programming, International Conference on Logic Programming, Special Issue pp. 493–507 (July 2014)
3. Cruz, F., Rocha, R., Goldstein, S.C.: Design and Implementation of a Multithreaded Virtual Machine for Executing Linear Logic Programs. In: International Symposium on Principles and Practice of Declarative Programming. pp. 43–53. ACM Press (September 2014)
4. De Koninck, L., Stuckey, P., Duck, G.: Optimizing compilation of CHR with rule priorities. In: Functional and Logic Programming, LNCS, vol. 4989, pp. 32–47 (2008)
5. Frhwirth, T.: Constraint handling rules. In: Constraint Programming: Basics and Trends, LNCS, vol. 910, pp. 90–107. Springer (1995)
6. Gonzalez, J., Low, Y., Guestrin, C.: Residual splash for optimally parallelizing belief propagation. In: Artificial Intelligence and Statistics (2009)
7. Holzbaur, C., de la Banda, M.J.G., Stuckey, P.J., Duck, G.J.: Optimizing compilation of constraint handling rules in HAL. CoRR cs.PL/0408025 (2004)
8. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Graphlab: A new framework for parallel machine learning. In: Conference on Uncertainty in Artificial Intelligence (UAI). pp. 340–349 (2010)
9. Wuille, P., Schrijvers, T., Demoen, B.: CCHR: the fastest CHR implementation, in C. In: Workshop on Constraint Handling Rules. pp. 123–137 (2007)