TLP **14** (4–5): 493–507, 2014. © Cambridge University Press 2014
doi:10.1017/S1471068414000167

A Linear Logic Programming Language for Concurrent Programming over Graph Structures

FLAVIO CRUZ[†‡], RICARDO ROCHA[‡], SETH COPEN GOLDSTEIN[†] and FRANK PFENNING[†]

[†]*Carnegie Mellon University, Pittsburgh, PA 15213*
(*e-mail:* `fmfernan, seth, fp@cs.cmu.edu`)
[‡]*CRACS & INESC TEC, Faculty of Sciences, University Of Porto*
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
(*e-mail:* `ricroc@dcc.fc.up.pt`)

submitted 13 February 2014; revised 18 April 2014; accepted 1 May 2014

Abstract

We have designed a new logic programming language called LM (Linear Meld) for programming graph-based algorithms in a declarative fashion. Our language is based on linear logic, an expressive logical system where logical facts can be consumed. Because LM integrates both classical and linear logic, LM tends to be more expressive than other logic programming languages. LM programs are naturally concurrent because facts are partitioned by nodes of a graph data structure. Computation is performed at the node level while communication happens between connected nodes. In this paper, we present the syntax and operational semantics of our language and illustrate its use through a number of examples.

KEYWORDS: Language Design, Semantics, Linear Logic, Concurrent Programming, Graphs

1 Introduction

Due to the popularity of social networks and the explosion of the content available in the World Wide Web, there has been increased interest in running graph-based algorithms concurrently. Most of the available frameworks are implemented as libraries on top of imperative programming languages, which require knowledge of both the library and the interface, making it difficult for both novice and expert programmers to learn and use correctly. Reasoning about the programs requires knowing how the library schedules execution and the operational semantics of the underlying language.

Some good examples are the Dryad, Pregel and GraphLab systems. The Dryad system (Isard *et al.* 2007) is a framework that combines computational vertices with communication channels (edges) to form a data-flow graph. Each program is scheduled to run on multiple computers or cores and data is partitioned during runtime. Routines that run on computational vertices are sequential, with no locking required. The Pregel system (Malewicz *et al.* 2010) is also graph-based, although programs have a more strict structure. They must be represented as a sequence of iterations where each iteration is composed of computation and message passing. Pregel is aimed at solving very big graphs and to scale to large architectures. GraphLab (Low *et al.* 2010) is a C++ library for developing parallel machine learning algorithms. While Pregel uses message passing, GraphLab allows nodes to have read/write access to different scopes through different concurrent access models in order to balance performance and data consistency. Each consistency model provides different guarantees that are suited to multiple classes of algorithms. GraphLab also provides several schedulers that dictate the order in which node's are computed.

An alternative promising approach for graph-based algorithms is logic programming. For instance, the P2 system (Loo *et al.* 2006), used Datalog to map nodes of a computer network to a graph, where each node would do computation locally and could communicate with neighbor nodes. Another good example is the Meld language, created by Ashley-Rollman et al. (Ashley-Rollman *et al.* 2007; Ashley-Rollman *et al.* 2009). Meld was itself inspired in the P2 system but adapted to the concept of massively distributed systems made of modular robots with a dynamic topology. Logic-based systems are more amenable to proof since a program is just a set of logical clauses.

In this paper, we present a new logic programming language called LM (Linear Meld) for concurrent programming over graph structures designed to take advantage of the recent architectures such as multicores or clusters of multicores. LM is based on the Meld language, but differs from other logic programming languages such as Datalog or Prolog in three main aspects. First, it integrates both classical logic and linear logic into the language, allowing some facts to be retracted and asserted in a logical fashion. Second, unlike Prolog, LM is a bottom up logic programming language (similar to Datalog) since the database is updated incrementally as rules are applied. Third, LM is a language created to solve general graph-based algorithms, unlike P2 or Meld which were designed for more specific domains.

In the following sections, we present the syntax and semantics of our language and explain how to write programs that take advantage of its expressive power. We identify three key contributions in our work:

Linear Logic: We integrate linear logic into the original Meld language so that program state can be encoded naturally. Meld started as a classical logic programming language where everything that is derived is true until the end of the execution. Linear logic turns logical facts into resources that will be consumed when a rule is applied. In turn, this makes it possible to represent program state in a natural and declarative fashion.

Concurrency: LM programs are naturally concurrent because facts are partitioned by vertices of a graph data structure. While the original Meld sees graphs as a network of robots, we see each node as a member of a distributed data structure. This is made possible due to the restrictions on derivation rules which only use local facts but also permit node communication.

Semantics: Starting from a fragment of linear logic used in LM, we formalize a high level dynamic semantics that is closely related to this fragment. We then design a low level dynamic semantics and sketch the soundness proof of our low level semantics with respect to the high level language specification. The low level specification provides the basis for a correct implementation of LM.

To realize LM, we have implemented a compiler and a virtual machine that executes LM programs on multicore machines[1]. We also have a preliminary version that runs on networks by using OpenMPI as a communication layer. Our experimental results show that LM has good scalability. Several interesting programs were implemented such as belief propagation (Gonzalez *et al.* 2009), belief propagation with residual splash (Gonzalez *et al.* 2009), PageRank, graph coloring, N queens, shortest path, diameter estimation, map reduce, game of life, quick-sort, neural network training, among others. While these results are evidence that LM is a promising language, this paper will only focus on the more formal aspects of our work.

2 LM By Example

Linear Meld (LM) is a *forward chaining* logic programming language in the style of Datalog (Ullman 1990). The program is defined as a *database of facts* and a set of *derivation rules*. Initially, we populate the database with the program's axioms and then determine which derivation rules can be applied by using the current database. Once a rule is applied, we derive new facts, which are then added to the database. If a rule uses linear facts, they are consumed and thus deleted from the database. The program stops when we reach *quiescence*, that is, when we can no longer apply any derivation rule.

The database of facts can be seen as a graph data structure where each node or vertex contains a fraction of the database. Since derivation rules can only manipulate facts belonging to a node, we are able to perform independent rule derivations.

Each fact is a predicate on a tuple of *values*, where the type of the predicate prescribes the types of the arguments. LM rules are type-checked using the predicate declarations in the header of the program. LM has a simple type system that includes types such as *node*, *int*, *float*, *string*, *bool*. Recursive types such as *list* X and *pair* $X; Y$ are also allowed.

The first argument of every predicate must be typed as a *node*. For concurrency and data partitioning purposes, derivation rules are constrained by the expressions that can be written in the body. The body of every rule can only refer to facts in the same node (same first argument). However, the expressions in the head may refer to other nodes, as long as those nodes are instantiated in the body of the rule.

Each rule in LM has a defined priority that is inferred from its position in the source file. Rules at the beginning of the file have higher priority. At the node level, we consider all the new facts that have been not consider yet to create a set of

[1] Source code is available at `http://github.com/flavioc/meld`.

```
1  type edge(node, node). // define direct edge
2  type linear message(node, string, list node). // message format
3
4  message(A, Content, [B | L]), !edge(A, B)
5     -o message(B, Content, L). // message derived at node B
6
7  message(A, Content, [])
8     -o 1. // message received
9
10 !edge(@1, @2). !edge(@2, @3). !edge(@3, @4). !edge(@1, @3).
11 message(@1, 'Hello World', [@3, @4]).
```

Fig. 1. Message program.

candidate rules. The set of candidate rules is then applied (by priority) and updated as new facts are derived.

Our first program example is shown in Fig. 1. This is a message routing program that simulates message transmission through a network of nodes. We first declare all the predicates (lines 1-2), which represent the different facts we are going to use. Predicate `edge/2` is a non `linear` (persistent) predicate and `message/3` is linear. While linear facts may be retracted, persistent facts are always true once they are derived.

The program rules are declared in lines 4-8, while the program's axioms are written in lines 10-11. The general form of a rule is $\mathtt{A}_1, ..., \mathtt{A}_n$ `-o` $\mathtt{B}_1, ..., \mathtt{B}_m$, where $\mathtt{A}_1, ..., \mathtt{A}_n$ are matched against local facts and $\mathtt{B}_1, ..., \mathtt{B}_m$ are locally asserted or transmitted to a neighboring node. When persistent facts are used (line 4) they must be preceded by `!` for readability.

The first rule (lines 4-5) grabs the next node in the route list (third argument of `message/3`), ensures that a communication edge exists with `!edge(A, B)` and derives a new `message(B, Content, L)` fact at node `B`. When the route list is empty, the message has reached its destination and thus it is consumed (rule in lines 7-8). Note that the '`1`' in the head of the rule on line 8 means that nothing is derived.

Figure 2 presents another complete LM program which given a graph of nodes visits all nodes reachable from node @1. The first rule of the program (lines 6-7) is fired when a node `A` has both the `visit(A)` and `unvisited(A)` facts. When fired, we first derive `visited(A)` to mark node `A` as *visited* and use a *comprehension* to go through all the edge facts `edge(A,B)` and derive `visit(B)` for each one (comprehensions are explained next in detail). This forces those nodes to be visited. The second rule (lines 9-10) is fired when a node `A` is already visited more than once: we keep the `visited(A)` fact and delete `visit(A)`. Line 14 starts the process by asserting the `visit(@1)` fact.

If the graph is connected, it is easy to prove that every node `A` will derive `visited(A)`, regardless of the order in which rules are applied.

3 The LM Language

Table 1 shows the abstract syntax for rules in LM. An LM program $Prog$ consists of a set of derivation rules Σ and a database D. A derivation rule R may be written as $BE \multimap HE$ where BE is the body of the rule and HE is the head. We can

Table 1. *Abstract syntax of LM.*

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

```
1   type edge(node, node).
2   type linear visit(node).
3   type linear unvisited(node).
4   type linear visited(node).
5
6   visit(A), unvisited(A)
7      -o visited(A), {B | !edge(A, B) | visit(B)}. // mark node as visited and visit neighbors
8
9   visit(A), visited(A)
10     -o visited(A). // already visited
11
12  !edge(@1, @2). !edge(@2, @3). !edge(@1, @4). !edge(@2, @4).
13  unvisited(@1). unvisited(@2). unvisited(@3). unvisited(@4).
14  visit(@1).
```

Fig. 2. Visit program.

also explicitly universally quantify over variables in a rule using $\forall_x.R$. If we want to control how facts are selected in the body, we may use *selectors* of the form $[\, S \Rightarrow y;\ BE\,] \multimap HE$ (explained later).

The body of the rule, BE, may contain linear (L) and persistent (P) *fact expressions* and constraints (C). We can chain those elements by using BE, BE or introduce body variables using $\exists_x.BE$. Alternatively we can use an empty body by using 1, which creates an axiom.

Fact expressions are template facts that instantiate variables (from facts in the database) such as `visit(A)` in line 10 in Fig. 2. Constraints are boolean expressions that must be true in order for the rule to be fired (for example, `C = A + B`). Constraints use variables from fact expressions and are built using 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 *exist expressions* (EE), *comprehensions* (CE) and *aggregates* (AE). All those expressions may use all the variables instantiated in the body. We can also use an empty head by choosing 1.

Selectors When a rule body is instantiated using facts from the database, facts are picked non-deterministically. While our system uses an implementation dependent order for efficiency reasons, sometimes it is important to sort facts by one of the arguments because linearity imposes commitment during rule derivation. The abstract syntax for this expression is $[\ S \Rightarrow y;\ BE\ ] \multimap HE$, where S is the selection operation and y is the variable in the body BE that represents the value to be selected according to S. An example using concrete syntax is as follows:

```
[min => W | !edge(A, B), weight(A, B, W)] -o picked(A, B, W).
```

In this case, we order the `weight` facts by `W` in ascending order and then try to match them. Other operations available are `max` and `random` (to force no pre-defined order).

Exists Expression Exists expressions (EE) are based on the linear logic term of the same name and are used to create new node addresses. We can then use the new address to instantiate new facts for this node. The following example illustrates the use of the exists expression, where we derive `perform-work` at a new node `B`.

```
do-work(A, W) -o exists B. (perform-work(B, W)).
```

Comprehensions Sometimes we need to consume a linear fact and then immediately generate several facts depending on the contents of the database. To solve this particular need, we created the concept of comprehensions, which are sub-rules that are applied with all possible combinations of facts from the database. In a comprehension $\{\ \widehat{x};\ BE;\ SH\ \}$, $\widehat{x}$ is a list of variables, BE is the comprehension's body and SH is the head. The body BE is used to generate all possible combinations for the head SH, according to the facts in the database. Note that BE is also locally restricted.

We have already seen an example of comprehensions in the visit program (Fig. 2 line 7). Here, we match `!edge(A, B)` using all the combinations available in the database and derive `visit(B)` for each combination.

Aggregates Another useful feature in logic programs is the ability to reduce several facts into a single fact. In LM we have aggregates (AE), a special kind of sub-rule similar to comprehensions. In the abstract syntax $[\ A \Rightarrow y;\ \widehat{x};\ BE;\ SH_1;\ SH_2\ ]$, A is the aggregate operation, $\widehat{x}$ is the list of variables introduced in BE, SH_1 and SH_2 and y is the variable in the body BE that represents the values to be aggregated using A. We use $\widehat{x}$ to try all the combinations of BE, but, in addition to deriving SH_1 for each combination, we aggregate the values represented by y and derive SH_2 only once using y.

Let's consider a database with the following facts and a rule:

```
price(@1, 3). price(@1, 4). price(@1, 5).
count-prices(@1).
count-prices(A) -o [sum => P | . | price(A, P) | 1 | total(A, P)].
```

By applying the rule, we consume `count-prices(@1)` and derive the aggregate which consumes all the `price(@1, P)` facts. These are added and `total(@1, 12)` is

```
type edge(node, node, int).
type linear path(node, int, int).

const used = 1.
const notused = 0.

path(startnode, 0, notused).

path(A, D, used), path(A, D, notused)
   -o path(A, D, used).

path(A, D1, X), path(A, D2, Y), D1 <= D2
   -o path(A, D1, X). // keep the shorter distance

path(A, D, notused), A <> finalnode
   -o {B, W | !edge(A, B, W) | path(B, D + W, notused)}, path(A, D, used). // propagate new distance
```

Fig. 3. Shortest Distance Program.

derived. LM provides aggregate operations such as `min` (minimum), `max` (maximum), `sum` and `count`.

4 Some Sample LM Programs

We now present LM programs in order to illustrate common programming techniques[2].

Shortest Distance Finding the shortest distance between two nodes in a graph is another well known graph problem. Fig. 3 presents the LM code to solve this particular problem.

We use an `edge/3` predicate to represent directed edges between nodes and their corresponding weights. To represent the shortest distance to a node `startnode` we have a `path(A,D,F)` where `D` is the distance to `startnode` and `F` is a flag to indicate if such distance has been propagated to the neighbors. Since the distance from the `startnode` to itself is `0`, we start the algorithm with the axiom `path(startnode,0,notused)`.

The first rule avoids propagating paths with the same distance and the second rule eliminates paths where the distance is already larger than some other distance. Finally, the third rule, marks the path as `used` and propagates the distance to the neighboring nodes by taking into account the edge weights. Eventually, the program will reach quiescence and the shortest distance between `startnode` and `finalnode` will be determined.

In the worst case, this algorithm runs in O(NE), where N is the number of nodes and E is the number of edges. If we decide to always propagate the shortest distance of the graph, we get Dijkstra's algorithm (Dijkstra 1959). However, this is not feasible, since we would need to globally decide which node to run next, removing concurrency.

PageRank PageRank (Page 2001) is a well known graph algorithm that is used to compute the relative relevance of web pages. The code for a synchronous version of the algorithm is shown in Fig. 4. As the name indicates, the pagerank is computed

[2] More examples of LM programs are available at `http://github.com/flavioc/meld`.

```
 1  type output(node, node, float).
 2  type linear pagerank(node, float, int).
 3  type numLinks(node, int).
 4  type numInput(node, int).
 5  type linear accumulator(node, float, int, int).
 6  type linear newrank(node, node, float, int).
 7  type linear start(node).
 8
 9  start(A).
10
11  start(A), !numInput(A, T)
12     -o accumulator(A, 0.0, T, 1), pagerank(A, 1.0 / float(@world), 0).
13
14  pagerank(A, V, Id), !numLinks(A, C), Id < iterations, Result = V / float(C)
15     -o {B, W | !output(A, B, W) | newrank(B, A, Result, Id + 1)}. // propagate new pagerank value
16
17  accumulator(A, Acc, 0, Id), !numInput(A, T), V = 0.85 + 0.15 * Acc, Id <= iterations
18     -o pagerank(A, V, Id), accumulator(A, 0.0, T, Id + 1). // new pagerank value
19
20  newrank(A, B, V, Id), accumulator(A, Acc, T, Id), T > 0
21     -o [sum => S, count => C | D | newrank(A, D, S, Id) | 1 | accumulator(A, Acc + V + S, T - 1 - C, Id)].
```

Fig. 4. Synchronous PageRank program.

for a certain number of iterations. The initial pagerank is the same for every page and is initialized in the first rule (line 12) along with an accumulator.

The second rule of the program (lines 14-15) propagates a newly computed pagerank value to all neighbors. Each node will then accumulate the pagerank values that are sent to them through the fourth rule (lines 20-21) and it will immediately add other currently available values through the use of the aggregate. When we have accumulated all the values we need, the third rule (lines 17-18) is fired and a new pagerank value is derived.

N-Queens The N-Queens (Hoffman *et al.* 1969) puzzle is the problem of placing N chess queens on an NxN chessboard so that no pair of two queens attack each other. The specific challenge of finding all the distinct solutions to this problem is a good benchmark in designing parallel algorithms. The LM solution is presented in the Appendix.

First, we consider each cell of the chessboard as a node that can communicate with the adjacent left (`left`) and adjacent right (`right`) cells and also with the first two non-diagonal cells in the next row (`down-left` and `down-right`). For instance, the node at cell `(0, 3)` (fourth cell in the first row) will connect to cells `(0, 2)`, `(0, 4)` and also `(1, 1)` and `(1, 5)`, respectively. The states are represented as a list of integers, where each integer is the column number where the queen was placed. For example `[2, 0]` means that a queen is placed in cell `(0, 0)` and another in cell `(1, 2)`.

An empty state is instantiated in the top-left node `(0, 0)` and then propagated to all nodes in the same row (lines 19-20). Each node then tries to place a queen on their cell and then send a new state to the row below (lines 52-54). Recursively, when a node receives a new state, it will (i) send the state to the left or to the right and (ii) try to place the queen in its cell (using `test-y`, `test-diag-left` and `test-diag-right`). When a cell cannot place a queen, that state is deleted (lines 29, 37 and 45). When the program ends, the states will be placed in the bottom row (lines 49-50).

Table 2. *Connectives from Linear Logic used in LM.*

Connective	Description	LM Place	LM Example
$fact(\hat{x})$	Linear facts.	Body/Head	`path(A, P)`
$!fact(\hat{x})$	Persistent facts.	Body/Head	`!edge(X, Y, W)`
1	Represents rules with an empty head.	Head	`1`
$A \otimes B$	Connect two expressions.	Body/Head	`p(A), e(A, B)`
$\forall x.A$	For variables defined in the rule.	Rule	`p(A)` $\multimap$ `r(A)`
$\exists x.A$	Instantiates new node variables.	Head	`exists A.(p(A, P))`
$A \multimap B$	$\multimap$ means "linearly implies". A is the body and B is the head.	Rule	`p(A, B)` $\multimap$ `r(A, B)`
$\mathsf{def} A.B$	Extension called definitions. Used for comprehensions and aggregates.	Head	`{B | !e(A, B) | v(B)}`

Most parallel implementations distribute the search space of the problem by assigning incomplete boards as tasks to workers. Our approach is unusual because our tasks are the cells of the board.

5 Proof Theory

We now present the sequent calculus of a fragment of intuitionistic linear logic (Girard 1987) used by LM followed by the dynamic semantics of LM built on top of this fragment.

We use a standard set of connectives except the $\mathsf{def}\ A$ connective, which is inspired on Baelde's work on least and greatest fixed points in linear logic (Baelde 2012) and is used to logically justify comprehensions and aggregates. The sequent calculus (shown in the Appendix) has the form $\Psi;\Gamma;\Delta \rightarrow C$, where Ψ is the typed term context used in the quantifiers, Γ is the set of persistent terms, Δ is the multi-set of linear propositions and C is the proposition to prove. Table 2 relates linear logic with LM.

In a comprehension, we want to apply an implication to as many matches as the database allows. Our approach is to use definitions: given a comprehension $C = \{\ \widehat{x};\ A;\ B\ \}$ with a body A and a head B, then we can build the following recursive definition:

$$\mathsf{def}\ C \triangleq 1 \ \&\ ((A \multimap B) \otimes \mathsf{def}\ C)$$

We unfold $\mathsf{def}\ C$ to either stop (by selecting 1) or get a linear implication $A \multimap B$ and a recursive definition. This uses linear logic's additive conjunction $\&$. This form of definition does not capture the desired *maximality* aspect of the comprehension, since it commits to finding a particular form of proof and not all possible proofs. The low level operational semantics will ensure maximality.

Aggregates work identically, but they need an extra argument to accumulate the aggregated value. If a sum aggregate C has the form $[\ \mathsf{sum} \Rightarrow y;\ \widehat{x};\ A;\ B_1;\ B_2\ ]$, then the definition will be as follows (the aggregate is initiated as $\mathsf{def}\ C\ 0$):

$$\mathsf{def}\ C\ V \triangleq (\lambda v.B_2)V \ \&\ (\forall x.((Ax \multimap B_1) \otimes \mathsf{def}\ C\ (x + V)))$$

$$\frac{\mathsf{apply}\ \Gamma;\Delta;R \rightarrow \Xi';\Delta';\Gamma'}{\mathsf{run}\ \Gamma;\Delta;R,\Phi \rightarrow \Xi';\Delta';\Gamma'}\ \mathsf{run}\ \mathit{rule} \qquad \frac{\mathsf{match}\ \Gamma;\Delta \rightarrow A \quad \mathsf{derive}\ \Gamma;\Delta'';\Delta;\cdot;\cdot;B \rightarrow \Xi';\Delta';\Gamma'}{\mathsf{apply}\ \Gamma;\Delta,\Delta'';A \multimap B \rightarrow \Xi';\Delta';\Gamma'}\ \mathsf{apply}\ \mathit{rule}$$

$$\frac{}{\mathsf{match}\ \Gamma;\cdot \rightarrow 1}\ \mathsf{match}\ 1 \qquad \frac{}{\mathsf{match}\ \Gamma;p \rightarrow p}\ \mathsf{match}\ p \qquad \frac{}{\mathsf{match}\ \Gamma,p;\cdot \rightarrow !p}\ \mathsf{match}\ !p$$

$$\frac{\mathsf{match}\ \Gamma;\Delta_1 \rightarrow A \quad \mathsf{match}\ \Delta_2 \rightarrow B}{\mathsf{match}\ \Gamma;\Delta_1,\Delta_2 \rightarrow A \otimes B}\ \mathsf{match}\ \otimes$$

$$\frac{\mathsf{derive}\ \Gamma;\Delta;\Xi;\Gamma_1;p,\Delta_1;\Omega \rightarrow \Xi';\Delta';\Gamma'}{\mathsf{derive}\ \Gamma;\Delta;\Xi;\Gamma_1;\Delta_1;p,\Omega \rightarrow \Xi';\Delta';\Gamma'}\ \mathsf{derive}\ p \qquad \frac{\mathsf{derive}\ \Gamma;\Delta;\Xi;\Gamma_1,p;\Delta_1;\Omega \rightarrow \Xi';\Delta';\Gamma'}{\mathsf{derive}\ \Gamma;\Delta;\Xi;\Gamma_1;\Delta_1;!p,\Omega \rightarrow \Xi';\Delta';\Gamma'}\ \mathsf{derive}\ !p$$

$$\frac{\mathsf{derive}\ \Gamma;\Delta;\Xi;\Gamma_1;\Delta_1;A,B,\Omega \rightarrow \Xi';\Delta';\Gamma'}{\mathsf{derive}\ \Gamma;\Delta;\Xi;\Gamma_1;\Delta_1;A \otimes B,\Omega \rightarrow \Xi';\Delta';\Gamma'}\ \mathsf{derive}\ \otimes \qquad \frac{\mathsf{derive}\ \Gamma;\Delta;\Xi;\Gamma_1;\Delta_1;\Omega \rightarrow \Xi';\Delta';\Gamma'}{\mathsf{derive}\ \Gamma;\Delta;\Xi;\Gamma_1;\Delta_1;1,\Omega \rightarrow \Xi';\Delta';\Gamma'}\ \mathsf{derive}\ 1$$

$$\frac{}{\mathsf{derive}\ \Gamma;\Delta;\Xi';\Gamma';\Delta';\cdot \rightarrow \Xi';\Delta';\Gamma'}\ \mathsf{derive}\ \mathit{end} \qquad \frac{\mathsf{match}\ \Gamma;\Delta_a \rightarrow A \quad \mathsf{derive}\ \Gamma;\Delta_b;\Xi,\Delta_a;\Gamma_1;\Delta_1;B,\Omega \rightarrow \Xi';\Delta';\Gamma'}{\mathsf{derive}\ \Gamma;\Delta_a,\Delta_b;\Xi;\Gamma_1;\Delta_1;A \multimap B,\Omega \rightarrow \Xi';\Delta';\Gamma'}\ \mathsf{derive}\ \multimap$$

$$\frac{\mathsf{derive}\ \Gamma;\Delta;\Xi;\Gamma_1;\Delta_1;1 \mathbin{\&} (A \multimap B \otimes \mathsf{comp}\ A \multimap B),\Omega \rightarrow \Xi';\Delta';\Gamma'}{\mathsf{derive}\ \Gamma;\Delta;\Xi;\Gamma_1;\Delta_1;\mathsf{comp}\ A \multimap B,\Omega \rightarrow \Xi';\Delta';\Gamma'}\ \mathsf{derive}\ \mathit{comp}$$

$$\frac{\mathsf{derive}\ \Gamma;\Delta;\Xi;\Gamma_1;\Delta_1;A,\Omega \rightarrow \Xi';\Delta';\Gamma'}{\mathsf{derive}\ \Gamma;\Delta;\Xi;\Gamma_1;\Delta_1;A \mathbin{\&} B,\Omega \rightarrow \Xi';\Delta';\Gamma'}\ \mathsf{derive}\ \& L \qquad \frac{\mathsf{derive}\ \Gamma;\Delta;\Xi;\Gamma_1;\Delta_1;B,\Omega \rightarrow \Xi';\Delta';\Gamma'}{\mathsf{derive}\ \Gamma;\Delta;\Xi;\Gamma_1;\Delta_1;A \mathbin{\&} B,\Omega \rightarrow \Xi';\Delta';\Gamma'}\ \mathsf{derive}\ \& R$$

Fig. 5. High Level Dynamic Semantics.

Dynamic Semantics The dynamic semantics formalize the mechanism of matching and deriving a single rule at the node level. The semantics receive the node database and the program's rules as inputs and return as outputs the consumed linear facts, derived linear facts and derived persistent facts. Then, it is possible to compute the program as a sequence of steps, by updating the database through sending or asserting.

High Level Dynamic Semantics The High Level Dynamic (HLD) Semantics are closely related to the linear logic fragment presented above. From the sequent calculus, we consider Γ and Δ the database of persistent and linear facts, respectively. We consider the rules of the program as persistent linear implications of the form $!(A \multimap B)$ that we put in a separate context Φ. We ignore the right hand side C of the calculus and use inversion on the Δ and Γ contexts so that we only have atomic terms (facts). To apply rules we use chaining by focusing (marc Andreoli 1992) on the derivation rules of Φ. The HLD semantics are shown in Fig. 5 and are composed of four judgments:

1. $\mathsf{run}\ \Gamma;\Delta;\Phi \rightarrow \Xi';\Delta';\Gamma'$ picks a rule from Φ and applies it using facts from Γ and Δ. Ξ', Δ' and Γ' are the outputs of the derivation process. Ξ' are the linear facts consumed, Δ' are the linear facts derived and Γ' the new persistent facts;
2. $\mathsf{apply}\ \Gamma;\Delta;R \rightarrow \Xi';\Delta';\Gamma'$ picks a subset of linear facts from Δ and matches the body of the rule R and then derives the head;

3. $\mathsf{match}\ \Gamma; \Delta \rightarrow A$ verifies that all facts in Δ (set of consumed linear facts) prove A, the body of the rule. The context Γ will be used to prove any persistent term in A;
4. $\mathsf{derive}\ \Gamma; \Delta; \Xi; \Gamma_1; \Delta_1, \Omega \rightarrow \Xi'; \Delta'; \Gamma'$ deconstructs and instantiates the ordered head terms Ω (we start with the head of the rule B) and adds them to Δ_1 and Γ_1, the contexts for the newly derived linear and persistent facts, respectively.

Comprehensions are derived by non-deterministically deciding to apply the comprehension ($\mathsf{derive}\ \& L$ and $\mathsf{derive}\ \& R$) and then using the match judgment in the rule derive *comp*. We note that the HLD semantics do not take distribution into account, since we assume that the database is global. We do not deal with unification or quantifiers since this is a well understood problem (Baader and Siekmann 1994).

Low Level Dynamic Semantics The Low Level Dynamic (LLD) Semantics (shown in the Appendix) improve upon HLD by adding rule priorities, by removing non-determinism when matching the body of rules by modeling all the matching steps and by applying comprehensions or aggregates as many times as the database allows. Selectors can also be trivially implemented in LLD, although they are not shown in paper.

In LLD we try all the rules in order. For each rule, we use a *continuation stack* to store the *continuation frames* created by each fact template p present in the body of the rule. Each frame considers all the facts relevant to the template given the current variable bindings (match_{LLD} rules), that may or not fail during the remaining matching process. If we fail, we backtrack to try other alternatives (through cont_{LLD} rules). If the continuation stack becomes empty, we backtrack to try the next rule (rule cont_{LLD} *next rule*). When we succeed the facts consumed are known (match_{LLD} *end*).

The derivation process in LLD is similar to the one used in HDL, except for the case of comprehensions or aggregates. For such cases (derive_{LLD} *comp*), we need to create a continuation stack and start matching the body of the expression as we did before. When we match the body (match_{LLDc} judgment), we fully derive the head (derive_{LLDc} judgment) and then we reuse the continuation stack to find which other combinations of the database facts can be consumed (derive_{LLDc} *end*). By definition, the continuation stack contains enough information to go through all combinations in the database.

However, in order to reuse the stack, we need to *fix* it by removing all the frames pushed after the first continuation frame of a linear fact. If we tried to use those frames, we would assumed that the linear facts used by the other frames were still in the database, but that is not true because they have been consumed during the first application of the comprehension. For example, if the body is `!a(X), b(X), c(X)` and the continuation stack has three frames (one per fact), we cannot backtrack to the frame of `c(X)` since at that point the matching process was assuming that the previous `b(X)` linear fact was still available. Moreover, we also remove the consumed linear facts from the frames of `b(X)` and `!a(X)` in order to make the stack fully consistent with the new database. This is performed by rules using the update_{LLD} and fix_{LLD} judgments.

We finally stop applying the comprehension when the continuation stack is empty ($\mathsf{cont}_{LLDc}\ end$). Aggregates use the same mechanism as comprehensions, however we also need to keep track of the accumulated value.

Soundness The soundness theorem proves that if a rule was successfully derived in the LLD semantics then it can also be derived in the HLD semantics. The completeness theorem cannot be proven since LLD lacks the non-determinism inherent in HLD.

We need prove to prove matching and derivation soundness of LLD in relation to HLD. The matching soundness lemma uses induction on the size of the continuation frames, the size of the continuation stack and the size of terms to match.

The derivation soundness lemma is trivial except for the case of comprehensions and aggregates. For such cases we use a modified version of the matching soundness theorem applied to the comprehension's body. It gives us n match and n derive proofs (for maximality) that are used to rebuild the full derivation proof in HLD. This theorem is proved by induction on the size of the continuation stack and continuation frames and uses lemmas that prove the correctness of the continuation stack after each application.[3]

6 Concurrency

Due to the restrictions on LM rules and the partitioning of facts across the graph, nodes are able to run rules independently without using other node's facts. Node computation follows a *don't care* or *committed choice* non-determinism since any node can be picked to run as long as it contains enough facts to fire a derivation rule. Facts coming from other nodes will arrive in order of derivation but may be considered partially and there is no particular order among the neighborhood. To improve concurrency, the programmer is encouraged to write rules that take advantage of non-deterministic execution.

LM programs can then be made parallel by simply processing many nodes simultaneously. Our implementation partitions the graph of N nodes into P subgraphs and then each processing unit will work on its subgraph. For improved load balancing we use node stealing during starvation. Our results show that LM programs running on multicores have good scalability. The implementation of the compiler and virtual machine and the analysis of experimental results will be presented in a future paper.

7 Related Work

To the best of our knowledge, LM is the first bottom-up linear logic programming language that is intended to be executed over graph structures. Although there are a few logic programming languages such as P2 (Loo *et al.* 2006), Meld (Ashley-Rollman *et al.* 2009), or Dedalus (Alvaro *et al.* 2009) that already do this, they are based on classical logic, where facts are persistent. For most of these systems, there is no concept of state, except for Dedalus where state is modeled as time.

[3] Details can be found in `https://github.com/flavioc/formal-meld/blob/master/doc.pdf?raw=true`.

Linear logic has been used in the past as a basis for logic-based programming languages (Miller 1985), including bottom-up and top-down programming languages. Lolli, a programming language presented in (Hodas and Miller 1994), is based on a fragment of intuitionistic linear logic and proves goals by lazily managing the context of linear resources during top-down proof search. LolliMon (López *et al.* 2005) is a concurrent linear logic programming language that integrates both bottom-up and top-down search, where top-down search is done sequentially but bottom-up computations, which are encapsulated inside a monad, can be performed concurrently. Programs start by performing top-down search but this can be suspended in order to perform bottom-up search. This concurrent bottom-up search stops until a fix-point is achieved, after which top-down search is resumed. LolliMon is derived from the concurrent logical framework called CLF (Watkins *et al.* 2004; Cervesato *et al.* 2002; Watkins *et al.* 2003).

Since LM is a bottom-up linear logic programming language, it also shares similarities with Constraint Handling Rules (CHR) (Betz and Frühwirth 2005; Betz and Frühwirth 2013). 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. Constraints can be consumed or generated during the application of rules. Unlike LM, in CHR there is no concept of rule priorities, but there is an extension to CHR that supports them (De Koninck *et al.* 2007). Finally, there is also a CHR extension that adds persistent constraints and it has been proven to be sound and complete (Betz *et al.* 2010).

Graph Transformation Systems (GTS) (Ehrig and Padberg 2004), commonly used to model distributed systems, perform rewriting of graphs through a set of graph productions. GTS also introduces concepts of concurrency, where it may be possible to apply several transformations at the same time. In principle, it should be possible to model LM programs as a graph transformation: we directly map the LM graph of nodes to GTS's initial graph and consider logical facts as nodes that are connected to LM's nodes. Each LM rule is then a graph production that manipulates the node's neighbors (the database) or sends new facts to other nodes. On the other hand, it is also possible to embed GTS inside CHR (Raiser and Frühwirth 2011).

8 Closing Remarks

In this paper, we have presented LM, a new linear logic programming language designed with concurrency in mind. LM is a bottom-up logic language that can naturally model state due to its foundations on linear logic. We presented several LM programs that show the viability of linear logic programming to solve interesting graph-based problems.

We also gave an overview of the formal system behind LM, namely, the fragment of linear logic used in the language, along with the high level and low level dynamic semantics. While the former is closely tied to linear logic, the latter is closer to a real implementation. The low level semantics can be used as a blueprint for someone that intends to implement LM.

Acknowledgments

This work is partially funded by the ERDF (European Regional Development Fund) through the COMPETE Programme; by FCT (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program and within projects SIBILA (NORTE-07-0124-FEDER-000059) and PEst (FCOMP-01-0124-FEDER-037281); and by the Qatar National Research Fund under grant NPRP 09-667-1-100. Flavio Cruz is funded by the FCT grant SFRH/BD/51566/2011.

References

Alvaro, P., Marczak, W., Conway, N., Hellerstein, J. M., Maier, D., and Sears, R. C. 2009. Dedalus: Datalog in time and space. Tech. Rep. UCB/EECS-2009-173, EECS Department, University of California, Berkeley. Dec.

Ashley-Rollman, M. P., Lee, P., Goldstein, S. C., Pillai, P., and Campbell, J. D. 2009. A language for large ensembles of independently executing nodes. In *International Conference on Logic Programming (ICLP)*.

Ashley-Rollman, M. P., Rosa, M. D., Srinivasa, S. S., Pillai, P., Goldstein, S. C., and Campbell, J. D. 2007. Declarative programming for modular robots. In *Workshop on Self-Reconfigurable Robots/Systems and Applications at IROS 2007*.

Baader, F. and Siekmann, J. H. 1994. Handbook of logic in artificial intelligence and logic programming. Oxford University Press, Inc., New York, NY, USA, Chapter Unification Theory, 41–125.

Baelde, D. 2012. Least and greatest fixed points in linear logic. *ACM Transactions on Computational Logic 13,* 1, 1–44.

Betz, H. and Frühwirth, T. 2005. A linear-logic semantics for constraint handling rules. In *Principles and Practice of Constraint Programming - CP 2005*. Lecture Notes in Computer Science, vol. 3709. 137–151.

Betz, H. and Frühwirth, T. W. 2013. Linear-logic based analysis of constraint handling rules with disjunction. *ACM Trans. Comput. Logic 14,* 1 (Feb.), 1:1–1:37.

Betz, H., Raiser, F., and Frühwirth, T. W. 2010. A complete and terminating execution model for constraint handling rules. *CoRR abs/1007.3829*.

Cervesato, I., Pfenning, F., Walker, D., and Watkins, K. 2002. A concurrent logical framework ii: Examples and applications. Tech. rep.

De Koninck, L., Schrijvers, T., and Demoen, B. 2007. User-definable rule priorities for chr. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP '07. New York, NY, USA, 25–36.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik 1,* 1, 269–271.

Ehrig, H. and Padberg, J. 2004. Graph grammars and petri net transformations. In *Lectures on Concurrency and Petri Nets*. Lecture Notes in Computer Science, vol. 3098. 496–536.

Girard, J.-Y. 1987. Linear logic. *Theoretical Computer Science 50,* 1, 1–102.

Gonzalez, J., Low, Y., and Guestrin, C. 2009. Residual splash for optimally parallelizing belief propagation. In *Artificial Intelligence and Statistics (AISTATS)*.

Hodas, J. S. and Miller, D. 1994. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation 110*, 32–42.

Hoffman, E. J., Loessi, J. C., and Moore, R. C. 1969. Construction for the solutions of the M queens problem. *Mathematics Magazine 42,* 2, 66–72.

Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*. 59–72.

Loo, B. T., Condie, T., Garofalakis, M., Gay, D. E., and Hellerstein, J. M. 2006. Declarative networking: Language, execution and optimization. In *International Conference on Management of Data (SIGMOD)*. 97–108.

López, P., Pfenning, F., Polakow, J., and Watkins, K. 2005. Monadic concurrent linear logic programming. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP '05. New York, NY, USA, 35–46.

Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. 2010. Graphlab: A new framework for parallel machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*. 340–349.

Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. 2010. Pregel: a system for large-scale graph processing. In *International Conference on Management of Data (SIGMOD)*. 135–146.

marc Andreoli, J. 1992. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation 2*, 297–347.

Miller, D. 1985. An overview of linear logic programming. In *in Computational Logic*. 1–5.

Page, L. 2001. Method for node ranking in a linked database. US Patent 6,285,999. Filed January 9, 1998. Expires around January 9, 2018.

Raiser, F. and Frühwirth, T. W. 2011. Analysing graph transformation systems through constraint handling rules. *Theory and Practice of Logic Programming 11,* 1 (Jan.), 65–109.

Ullman, J. D. 1990. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies.*

Watkins, K., Cervesato, I., Pfenning, F., and Walker, D. 2003. A concurrent logical framework i: Judgments and properties. Tech. rep.

Watkins, K., Cervesato, I., Pfenning, F., and Walker, D. 2004. A concurrent logical framework: The propositional fragment. In *Types for Proofs and Programs*. Lecture Notes in Computer Science, vol. 3085. 355–377.