A Linear Logic Programming Language for Concurrent Programming over Graph Structures

Flavio Cruz†‡, Ricardo Rocha‡, Seth Copen Goldstein†, 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 1 January 2003; revised 1 January 2003; accepted 1 January 2003

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: Linear Logic, Concurrent Programming, Semantics, Language Design, 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. While some programs only need to access the local node’s data, others may need to update edge information. 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), which was designed to program massively distributed systems made of modular robots with a dynamic topology. Meld was itself inspired in the P2 system because it is based on the graph model of computation but adapted to the concept of moving robots. 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, safe 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. 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. 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 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 $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$, where $A_1, \ldots, A_n$ are matched against local facts and $B_1, \ldots, B_m$ are locally asserted or transmitted to a neighboring node. When persistent facts are used (line 5) they must be preceded by 1 for readability.

\(^1\) Source code is available at http://github.com/flavioc/meld.
type edge(node, node). // define direct edge
type linear message(node, string, list node). // message format

message(A, Content, [B | L]),
!edge(A, B)
-o message(B, Content, L). // message derived at node B

message(A, Content, []) -o 1. // message received

!edge(@1, @2). !edge(@2, @3). !edge(@1, @4). !edge(@2, @4).
message(@1, 'Hello World', [@3, @4]).

Fig. 1. Message program.

The first rule (lines 4-6) grabs the next node in the route list (third argument of message/3) and ensures that a communication edge exists (through edge(A, B)). A new message(B, Content, L) fact is derived at node B. When the route list is empty, the message has reached its destination and thus it is consumed (rule in lines 8-9). 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 (line 9) is fired when a node A is already visited more than once: we keep the visited(A) fact and delete visit(A). Line 13 starts the process by asserting the visit(@1) fact.

Fig. 2. Visit program.

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 $\Sigma$ and a database $D$. A derivation rule $R$ may be written as $BE \rightarrow HE$ where $BE$ is the body of the rule and $HE$ is the head. We can 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 $[S \Rightarrow y; BE] \rightarrow HE$ (explained later).

<table>
<thead>
<tr>
<th>Program</th>
<th>$Prog$</th>
<th>::=</th>
<th>$\Sigma, D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Of Rules</td>
<td>$\Sigma$</td>
<td>::=</td>
<td>$\cdot</td>
</tr>
<tr>
<td>Database</td>
<td>$D$</td>
<td>::=</td>
<td>$\Gamma; \Delta$</td>
</tr>
<tr>
<td>Rule</td>
<td>$R$</td>
<td>::=</td>
<td>$BE \rightarrow HE \mid \forall x.R \mid [S \Rightarrow y; BE] \rightarrow HE$</td>
</tr>
<tr>
<td>Body Expression</td>
<td>$BE$</td>
<td>::=</td>
<td>$L \mid P \mid C \mid BE \mid \exists x.BE \mid 1$</td>
</tr>
<tr>
<td>Head Expression</td>
<td>$HE$</td>
<td>::=</td>
<td>$L \mid P \mid HE \mid HE \mid EE \mid CE \mid AE \mid 1$</td>
</tr>
<tr>
<td>Linear Fact</td>
<td>$L$</td>
<td>::=</td>
<td>$l(\hat{x})$</td>
</tr>
<tr>
<td>Persistent Fact</td>
<td>$P$</td>
<td>::=</td>
<td>$!p(\hat{x})$</td>
</tr>
<tr>
<td>Constraint</td>
<td>$C$</td>
<td>::=</td>
<td>$c(\hat{x})$</td>
</tr>
<tr>
<td>Selector Operation</td>
<td>$S$</td>
<td>::=</td>
<td>min \mid max \mid random</td>
</tr>
<tr>
<td>Exists Expression</td>
<td>$EE$</td>
<td>::=</td>
<td>$\exists x.SH$</td>
</tr>
<tr>
<td>Comprehension</td>
<td>$CE$</td>
<td>::=</td>
<td>${ \exists x.BE; SH }$</td>
</tr>
<tr>
<td>Aggregate</td>
<td>$AE$</td>
<td>::=</td>
<td>$[A \Rightarrow y; \exists x.BE; SH_1; SH_2]$</td>
</tr>
<tr>
<td>Aggregate Operation</td>
<td>$A$</td>
<td>::=</td>
<td>$\min</td>
</tr>
<tr>
<td>Sub-Head</td>
<td>$SH$</td>
<td>::=</td>
<td>$L \mid P \mid SH \mid \text{count}$</td>
</tr>
<tr>
<td>Known Linear Facts</td>
<td>$\Delta$</td>
<td>::=</td>
<td>$\cdot</td>
</tr>
<tr>
<td>Known Persistent Facts</td>
<td>$\Gamma$</td>
<td>::=</td>
<td>$\cdot</td>
</tr>
</tbody>
</table>

Table 1. Abstract syntax of LM.

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 uninstantiate 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 $[S \Rightarrow y; BE] \rightarrow HE$, 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:

$$[\min \Rightarrow W \mid !\text{edge}(A, B), \text{weight}(A, B, W)] \rightarrow \text{picked}(A, B, W).$$
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 \( B \).

\[
\text{do-work}(A, W) \rightarrow \text{exists } B. \text{ (perform-work}(B, W)).
\]

**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 \( \{ \bar{x}; BE; SH \} \), \( \bar{x} \) 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 \( [A \Rightarrow y; \bar{x}; BE; SH_1; SH_2] \), \( A \) is the aggregate operation, \( \bar{x} \) 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 \( \bar{x} \) 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:

\[
\text{price}(\@1, 3). \text{price}(\@1, 4). \text{price}(\@1, 5). \\
\text{count-prices}(\@1). \\
\text{count-prices}(A) \rightarrow [\text{sum} \Rightarrow P | . | \text{price}(A, P) | 1 | \text{total}(A, P)].
\]

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 derived.

LM provides aggregate operations such as minimum, 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 predicate to represent directed edges between nodes and their corresponding weights. To represent the shortest distance to a node startnode we have a

\(^2\) More examples of LM programs are available at http://github.com/flavioc/meld.
path(A, B, F) where B 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 with the axiom path(startnode, 0, notused) to start the algorithm.

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 quies-
cence and the shortest distance between startnode and finalnode will be determined.

1 type edge(node, node, int).
2 type linear path(node, int, int).
3 const used = 1.
4 const notused = 0.
5 path(startnode, 0, notused).
6 path(A, B, used), path(A, B, notused).
7 -o path(A, B, used).
8 path(A, B1, X), path(A, B2, Y), B1 <= B2
9 -o path(A, B1, X) // keep the shorter distance
10 path(A, D, notused), A <> finalnode // propagate distance
11 -o {B, W | !edge(A, B, W) | path(B, D + W, notused)}, path(A, D, used).

Fig. 3. Shortest Distance Program.

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 for a certain number
of iterations. The initial pagerank is the same for every page and is initialized in the first
rule (lines 15-16) along with an accumulator.

The second rule of the program (lines 18-22) 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 30-32) 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 24-28) is fired and a new pagerank value is derived.

We also have an asynchronous version of the algorithm that trades correctness with
convergence speed since it does not synchronize between iterations.

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 Appendix A.

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
Flavio Cruz, Ricardo Rocha, Seth Copen Goldstein and Frank Pfenning

1. type output(node, node, float).
2. type linear pagerank(node, float, int).
3. type numLinks(node, int).
4. type linear accumulator(node, float, int).
5. type linear numInput(node, int).
6. type linear newrank(node, node, float, int).
7. type linear start(node).

8. const damping = 0.85.
9. const iterations = str2int(@arg1).
10. const pages = @world.

11. start(A).

12. start(A), !numInput(A, T)
13. -o accumulator(A, 0.0, T, 1), pagerank(A, 1.0 / float(pages), 0).
14. pagerank(A, V, Id), // propagate new pagerank value
15. !numLinks(A, C),
16. Id < iterations,
17. Result = V / float(C)
18. -o {B, W | !output(A, B, W) | newrank(B, A, Result, Id + 1)}.
19. accumulator(A, Acc, 0, Id), // new pagerank
20. !numInput(A, T),
21. V = damping + (1.0 - damping) * Acc,
22. Id <= iterations,
23. Result = V / float(C)
24. -o pagerank(A, V, Id), accumulator(A, 0.0, T, Id + 1).
25. newrank(A, B, V, Id), accumulator(A, Acc, T, Id), T > 0
26. -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.

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 20-22). Each node then tries to place a queen on their cell and then send a new state to the row below (lines 63-65). 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). With this method, all states will be computed since we have facts for each valid state at that point. When a cell cannot place a queen, that state is deleted (lines 31, 42 and 54). When the program ends, the states will be placed in the bottom row (lines 59-61).

We find our solution very elegant, since it can be easily distributed or executed in parallel. To the best of our knowledge it is also an unusual approach to this problem.

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 built on top of this fragment.

The fragment of linear logic used by LM is presented in Appendix B. We use a standard set of connectives except the def A connective, which is used to logically justify comprehensions and aggregates. The sequent calculus has the form $\Gamma; \Delta \rightarrow C$, where $\Psi$ is the typed term context used in the quantifiers, $\Gamma$ is the set of persistent terms, $\Delta$ is
Connective Description LM Place LM Example

<table>
<thead>
<tr>
<th>Connective</th>
<th>Description</th>
<th>LM Place</th>
<th>LM Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>fact((\hat{x}))</td>
<td>Linear facts.</td>
<td>Body/Head</td>
<td>path(A, P)</td>
</tr>
<tr>
<td>!fact((\hat{x}))</td>
<td>Persistent facts.</td>
<td>Body/Head</td>
<td>!edge(X, Y, W)</td>
</tr>
<tr>
<td>1</td>
<td>Represents rules with an empty head.</td>
<td>Head</td>
<td>1</td>
</tr>
<tr>
<td>(A \otimes B)</td>
<td>Connect two expressions.</td>
<td>Body/Head</td>
<td>p(A), e(A, B)</td>
</tr>
<tr>
<td>(\forall x.A)</td>
<td>For variables defined in the rule.</td>
<td>Rule</td>
<td>p(A) (\rightarrow) r(A)</td>
</tr>
<tr>
<td>(\exists x.A)</td>
<td>Instantiates new node variables.</td>
<td>Head</td>
<td>exists A.(p(A, P))</td>
</tr>
<tr>
<td>(A \rightarrow B)</td>
<td>(\rightarrow) means “linearly implies”.</td>
<td>Rule</td>
<td>p(A, B) (\rightarrow) r(A, B)</td>
</tr>
<tr>
<td>def (A, B)</td>
<td>Extension called definitions. Used for comprehensions and aggregates.</td>
<td>Head</td>
<td>{B</td>
</tr>
</tbody>
</table>

Table 2. Connectives from Linear Logic used in LM.

the multi-set of linear propositions and \(C\) is the proposition we want to prove. Table 2 relates the linear logic directives with the LM syntax.

For definitions, we took inspiration from Baelde’s work on least and greatest fixed points in linear logic (Baelde 2012). Baelde’s system goes beyond simple recursive definitions and allows proofs using induction and co-induction in linear logic. Fortunately, the proof system satisfies cut-elimination (Baelde 2012) which makes it consistent.

In a comprehension, we want to apply an implication to as many matches as the database allows. One way to formally describe comprehensions would be to use persistent rules that would be used a few times and then would be forgotten. A more reasonable approach is to use definitions. Given a comprehension \(C = \{ \hat{x}; A; B \}\) with a body \(A\) and a head \(B\), then we can build the following recursive definition:

\[
\text{def } C \triangleq 1 \& ((A \rightarrow B) \otimes \text{def } C)
\]

We can unfold \(\text{def } C\) to either stop (by selecting 1) or get a new linear implication \(A \rightarrow B\) to apply the comprehension once. Because we also get another \(\text{def } C\) by selecting the right hand side, the comprehension can be applied again, recursively.

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 \([\text{sum } \Rightarrow y; \hat{x}; A; B_1; B_2]\), then the definition will be as follows (the aggregate is initiated as \(\text{def } C 0\)):

\[
\text{def } C \ V \triangleq (\lambda v.B_2)V \& (\forall x.(Ax \rightarrow B_1) \otimes \text{def } C \ (x + V)))
\]

5.1 Dynamic Semantics

The high level dynamic semantics (HLD) formalize the mechanism of matching and deriving a single rule. It is closely related to the fragment presented above. From the
sequent calculus, we consider \( \Gamma \) and \( \Delta \) the database of persistent and linear facts, respectively. We consider the rules of the program as persistent linear implications of the form \( !(A \rightarrow B) \) that we put in a separate context \( \Phi \). We ignore the right hand side \( C \) of the calculus and use inversion on the \( \Delta \) and \( \Gamma \) contexts so that we only have atomic terms (facts). To apply rules we use chaining by focusing on the derivation rules of \( \Phi \).

\[
\text{apply } \Gamma; \Delta; R \rightarrow \Sigma'; \Lambda'; \Gamma' \\
\text{run } \Gamma; \Delta; \Phi \rightarrow \Sigma'; \Lambda'; \Gamma'
\]

\[
\frac{\text{match } \Gamma; \Delta \rightarrow A \quad \text{derive } \Gamma; \Delta''; \Delta; \vdots; B \rightarrow \Sigma'; \Lambda'; \Gamma'}{\text{apply } \Gamma; \Delta, \Delta''; A \rightarrow B \rightarrow \Sigma'; \Lambda'; \Gamma'}
\]

Fig. 5. High Level Dynamic Semantics.

The HLD semantics are shown in Fig. 5 and are composed of four judgments:

1. run \( \Gamma; \Delta; \Phi \rightarrow \Sigma'; \Lambda'; \Gamma' \) picks a rule from \( \Phi \) and applies it using facts from \( \Gamma \) and \( \Delta, \Sigma', \Lambda' \) and \( \Gamma' \) are the outputs of the derivation process. \( \Sigma' \) are the linear facts consumed, \( \Lambda' \) are the linear facts derived and \( \Gamma' \) the new persistent facts;

2. apply \( \Gamma; \Delta; R \rightarrow \Sigma'; \Lambda'; \Gamma' \) picks a subset of linear facts from \( \Delta \) and matches the body of the rule \( R \) and then derives the head;
3. match $\Gamma; \Delta \rightarrow A$ verifies that $\Delta$ proves $A$ directly;
4. derive $\Gamma; \Delta; \Xi; \Gamma_1; \Delta_1, \Omega \rightarrow \Xi'; \Delta'; \Gamma'$ deconstructs and instantiates the ordered head terms $\Omega$ and adds them to $\Delta_1$ and $\Gamma_1$, the contexts for new linear and persistent facts, respectively.

Comprehensions are derived by non-deterministically deciding to apply the comprehension (derive & L and 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 also omit the quantifiers and do not deal with unification since it complicates the formal system.

The low level dynamic semantics (LLD) shown in Appendix C 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.

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 (matchLLD rules), that may or not fail during the remaining matching process. If we fail, we backtrack to try other alternatives (through contLLD rules). If the continuation stack becomes empty, we backtrack to try the next rule (rule contLLD next rule). When we succeed the facts consumed are known (matchLLD 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 (deriveLLD 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 (matchLLDx judgment), we fully derive the head (deriveLLDx judgment) and then we reuse the continuation stack to find which other combinations of the database facts can be consumed (deriveLLDx 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 updateLLD and fixLLD judgments.

We finally stop applying the comprehension when the continuation stack is empty (contLLDx end). Aggregates use the same mechanism as comprehensions, however we also need to keep track of the accumulated value.

5.2 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 prove that matching the body of a rule in LLD can also be performed in HLD (matching soundness) and that deriving the head of the rule is also sound (derivation soundness). The matching soundness uses induction on the size of the continuation frames, the size of the continuation stack and the size of terms to match to reconstitute a particular matching tree in HLD.

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.

6 Concurrency

Due to the restrictions on LM rules and the partitioning of facts across the graph, nodes are able to execute independently. Node computation happens non-deterministically 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.

LM programs can then be made concurrent by simply processing many nodes concurrently. In practice this can be done in many ways, for instance, given $P$ processing units and $N$ nodes, we could place the nodes in a work queue and then each processing unit would take a node from the queue, process it, and then place it back at the end of the queue. This trivial technique works but there is no consideration for node connections and data locality. 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 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 programming language that can naturally model state due to its foundations on linear logic. We presented several programs written in LM that show the viability of a linear logic based programming language 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 dynamic semantics can be used as a blueprint for someone that intends to implement our language.

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


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


Appendix A N-Queens program

1 type left(node, node).
2 type right(node, node).
3 type down-right(node, node).
4 type down-left(node, node).
5 type coord(node, int, int).
6 type linear propagate-left(node, list node, list int).
7 type linear propagate-right(node, list node, list int).
8 type linear test-and-send-down(node, list node, list int).
9 type linear test-y(node, int, list int, list node, list int).
10 type linear test-diag-left(node, int, int, list int, list node, list int).
11 type linear test-diag-right(node, int, int, list int, list node, list int).
12 type linear send-down(node, list node, list int).
13 type linear final-state(node, list node, list int).
14
15 propagate-right(@0, [], []).
16 propagate-left(A, Nodes, Coords)
17 -o {L | !left(A, L), L <> A | propagate-left(L, Nodes, Coords)}.
18 test-and-send-down(A, Nodes, Coords).
19 propagate-right(A, Nodes, Coords)
20 -o {R | !right(A, R), R <> A | propagate-right(R, Nodes, Coords)}.
21 test-and-send-down(A, Nodes, Coords).
22
23 test-and-send-down(A, Nodes, Coords),
24 !coord(A, X, Y)
25 -o test-y(A, Y, Coords, Nodes, Coords).
26
27 // test if we have a queen on this column
28 test-y(A, Y, [], [], Coords), !coord(A, OX, OY)
29 -o test-diag-left(A, OX - 1, OY - 1, Coords, Nodes, Coords).
30 test-y(A, Y, [X, Y1 | RestCoords], Nodes, Coords), Y = Y1 - o 1. // fail
31 test-y(A, Y, [X, Y1 | RestCoords], Nodes, Coords), Y <> Y1 - o
32 test-y(A, Y, RestCoords, Nodes, Coords).
33
34 // test if we have a queen on the left diagonal
35 test-diag-left(A, X, Y, Nodes, Coords),
36 !coord(A, OX, OY)
37 -o test-diag-right(A, OX - 1, OY, Coords, Nodes, Coords).
38 test-diag-left(A, X, Y, [X1, Y1 | RestCoords], Nodes, Coords),
39 X = X1, Y = Y1
40 - o 1. // fail
41 test-diag-left(A, X, Y, [X1, Y1 | RestCoords], Nodes, Coords),
42 X <> X1 || Y <> Y1
43 - o test-diag-left(A, X - 1, Y - 1, RestCoords, Nodes, Coords).
44
45 // test if we have a queen on the right diagonal
46 test-diag-right(A, X, Y, [], [], Coords),
47 !coord(A, OX, OY)
48 -o send-down(A, [A | Nodes], [OX, OY | Coords]). // add new queen
49 test-diag-right(A, X, Y, [X1, Y1 | RestCoords], Nodes, Coords),
50 X = X1, Y = Y1
51 - o 1. // fail
52 test-diag-right(A, X, Y, [X1, Y1 | RestCoords], Nodes, Coords),
53 X <> X1 || Y <> Y1
54 - o test-diag-right(A, X - 1, Y + 1, RestCoords, Nodes, Coords).
55
56 send-down(A, Nodes, Coords),
57 !coord(A, size - 1, _)
58 - o final-state(A, Nodes, Coords).
59
60 send-down(A, Nodes, Coords)
61 - o {B | !down-right(A, B), B <> A | propagate-right(B, Nodes, Coords)}.
62 {B | !down-left(A, B), B <> A | propagate-left(B, Nodes, Coords)}. 
Appendix B  Linear Logic fragment used in LM

\[
\begin{align*}
\frac{}{\Psi; \Gamma; \Delta \Rightarrow A} & \qquad \frac{}{\Psi; \Gamma; \Delta \Rightarrow B} \quad \& \Rightarrow \quad \frac{ \Psi; \Gamma; \Delta \Rightarrow A \, B }{} \\
\frac{}{\Psi; \Gamma; \Delta \Rightarrow A \, \& \, B} & \quad \& \Rightarrow \quad \frac{ \Psi; \Gamma; \Delta; A \, \& \, B \Rightarrow C }{} \\
\frac{ \Psi; \Gamma; \Delta \Rightarrow C }{} & \quad \& \Rightarrow \quad \frac{ \Psi; \Gamma; \Delta \Rightarrow A \, \& \, B \Rightarrow C }{} \\
\frac{}{\Psi; \Gamma; \Delta \Rightarrow A} \quad \& \Rightarrow \quad \frac{}{\Psi; \Gamma; \Delta \Rightarrow B} \quad \& \Rightarrow \quad \frac{ \Psi; \Gamma; \Delta, A \, \& \, B \Rightarrow C }{} \\
\frac{}{\Psi; \Gamma; \Delta, A \, \& \, B \Rightarrow C} & \quad \& \Rightarrow \quad \frac{ \Psi; \Gamma; \Delta \Rightarrow A \, B }{} \\
\frac{}{\Psi; \Gamma; \Delta \Rightarrow A \, B} & \quad \& \Rightarrow \quad \frac{ \Psi; \Gamma; \Delta \Rightarrow A \, B }{} \\
\frac{}{\Psi; \Gamma; \Delta \Rightarrow A} \quad \& \Rightarrow \quad \frac{ \Psi; \Gamma; \Delta \Rightarrow B }{} \\
\frac{}{\Psi; \Gamma; \Delta \Rightarrow B} & \quad \& \Rightarrow \quad \frac{ \Psi; \Gamma; \Delta \Rightarrow A }{} \\
\frac{}{\Psi; \Gamma; \Delta \Rightarrow A} & \quad \& \Rightarrow \quad \frac{}{\Psi; \Gamma; \Delta \Rightarrow B} \quad \& \Rightarrow \quad \frac{ \Psi; \Gamma; \Delta, A \, \& \, B \Rightarrow C }{} \\
\frac{}{\Psi; \Gamma; \Delta \Rightarrow A \, \& \, B \Rightarrow C} & \quad \& \Rightarrow \quad \frac{ \Psi; \Gamma; \Delta \Rightarrow A \, B }{} \\
\frac{}{\Psi; \Gamma; \Delta \Rightarrow A \, B \Rightarrow C} & \quad \& \Rightarrow \quad \frac{ \Psi; \Gamma; \Delta \Rightarrow A \, B }{} \\
\frac{}{\Psi; \Gamma; \Delta \Rightarrow A} \quad \& \Rightarrow \quad \frac{ \Psi; \Gamma; \Delta \Rightarrow B }{} \\
\frac{}{\Psi; \Gamma; \Delta \Rightarrow B} & \quad \& \Rightarrow \quad \frac{ \Psi; \Gamma; \Delta \Rightarrow A }{} \\
\end{align*}
\]
Appendix C Low Level Dynamic Semantics

All the judgments in this system share a few arguments, namely:

- $\Gamma$: Set of persistent facts.
- $\Xi'$: Multi-set of facts consumed after applying one rule (output).
- $\Delta'$: Multi-set of linear facts derived after applying one rule (output).
- $\Gamma'$: Set of persistent facts derived after applying one rule (output).

C.1 Application

The whole process is started by the $\text{run}_{\text{LLD}}$ and $\text{apply}_{\text{LLD}}$ judgments. The $\text{run}_{\text{LLD}} \Gamma; \Delta; \Phi \rightarrow \Xi'; \Delta'; \Gamma'$ judgment starts with facts $\Delta$ and $\Gamma$ and an ordered list of rules that can be applied $\Phi$. The $\text{apply}_{\text{LLD}} \Gamma; \Delta; A \rightarrow B; R \rightarrow \Xi'; \Delta'; \Gamma'$ tries to apply the rule $A \rightarrow B$ and stores the rule continuation $R$ so that if the current rule fails, we can try another one (in order).

\[
\frac{\text{match}_{\text{LLD}} \Gamma; \Delta; \cdots; A; B; \cdots; R \rightarrow \Xi'; \Delta'; \Gamma'}{\text{apply}_{\text{LLD}} \Gamma; \Delta; A \rightarrow B; R \rightarrow \Xi'; \Delta'; \Gamma'} \frac{\text{apply}_{\text{LLD}} \Gamma; \Delta; R; (\Phi, \Delta) \rightarrow \Xi'; \Delta'; \Gamma'}{\text{run}_{\text{LLD}} \Gamma; \Delta; R; \Phi \rightarrow \Xi'; \Delta'; \Gamma'} \frac{\text{run}_{\text{LLD}} \Gamma; \Delta; R; \Phi \rightarrow \Xi'; \Delta'; \Gamma'}{\text{run}_{\text{LLD}} \Gamma; \Delta; R; \Phi \rightarrow \Xi'; \Delta'; \Gamma'}
\]

C.2 Continuation Frames

Continuation frames are used for backtracking in the matching process.

C.2.1 Persistent Frame

A persistent frame has the form $[\Gamma'; \Delta; \Xi; !p; \Omega; \Lambda; \Upsilon]$, where:

- $\Delta$: Remaining multi-set of linear facts.
- $\Xi$: Multi-set of linear resources we have consumed to reach this point.
- $!p$: Current fact expression that originated this choice point.
- $\Omega$: Remaining terms we need to match past this choice point. This is an ordered list.
- $\Lambda$: Multi-set of linear fact expressions that we have matched to reach this choice point. All the linear facts that match these terms are located in $\Xi$.
- $\Upsilon$: Multi-set of persistent fact expressions that we matched up to this point.

C.2.2 Linear Frame

A linear frame has the form $(\Delta; \Delta'; \Xi; p; \Omega; \Lambda; \Upsilon)$, where:

- $\Delta$: Multi-set of linear resources that are not of type $p$ plus all the other $p$'s we have already tried, including the current $p$.
- $\Delta'$: All the other $p$'s we haven’t tried yet. It is a multi-set of linear resources.
- $\Xi$: Multi-set of linear resources we have consumed to reach this point.
- $p$: Current fact expression that originated this choice point.
- $\Omega$: Remaining terms we need to match past this choice point. This is an ordered list.
The matching judgment uses the form \( \text{match}_{\text{LLD}}(\Gamma; \Delta; \Xi; \Omega; H; C; R \rightarrow \Xi'; \Delta'; \Gamma') \) where:

- \( \Delta \): Multi-set of linear fact expressions still available to complete the matching process.
- \( \Xi \): Multi-set of linear fact expressions consumed up to this point.
- \( \Omega \): Ordered list of terms we want to match.
- \( H \): Head of the rule.
- \( C \): Ordered list of frames representing the continuation stack.
- \( R \): Rule continuation. If the matching process fails, we try another rule.

Matching will attempt to use facts from \( \Delta \) and \( \Gamma \) to match the terms of the body of the rule represented as \( \Omega \). During this process continuation frames are pushed into \( C \). If the matching process fails, we use the continuation stack through the \( \text{cont}_{\text{LLD}} \) judgment.

\[
\begin{align*}
\text{match}_{\text{LLD}}(\Gamma; \Delta; \Xi; \Omega; H; C; R \rightarrow \Xi'; \Delta'; \Gamma') &\quad \text{match}_{\text{LLD}} p \text{ ok first} \\
\text{match}_{\text{LLD}}(\Gamma; \Delta; \Xi; \Omega; H; C; R \rightarrow \Xi'; \Delta'; \Gamma') &\quad \text{match}_{\text{LLD}} p \text{ ok other } q \\
\text{match}_{\text{LLD}}(\Gamma; \Delta; \Xi; \Omega; H; C; R \rightarrow \Xi'; \Delta'; \Gamma') &\quad \text{match}_{\text{LLD}} p \text{ ok other } \Lambda \\
\text{match}_{\text{LLD}}(\Gamma; \Delta; \Xi; \Omega; H; C; R \rightarrow \Xi'; \Delta'; \Gamma') &\quad \text{match}_{\text{LLD}} p \text{ fail} \\
\text{match}_{\text{LLD}}(\Gamma; \Delta; \Xi; \Omega; H; C; R \rightarrow \Xi'; \Delta'; \Gamma') &\quad \text{match}_{\text{LLD}} !p \text{ ok first} \\
\text{match}_{\text{LLD}}(\Gamma; \Delta; \Xi; \Omega; H; C; R \rightarrow \Xi'; \Delta'; \Gamma') &\quad \text{match}_{\text{LLD}} !p \text{ ok other } q \\
\text{match}_{\text{LLD}}(\Gamma; \Delta; \Xi; \Omega; H; C; R \rightarrow \Xi'; \Delta'; \Gamma') &\quad \text{match}_{\text{LLD}} !p \text{ ok other } \Lambda \\
\text{match}_{\text{LLD}}(\Gamma; \Delta; \Xi; \Omega; H; C; R \rightarrow \Xi'; \Delta'; \Gamma') &\quad \text{match}_{\text{LLD}} !p \text{ fail} \\
\text{match}_{\text{LLD}}(\Gamma; \Delta; \Xi; \Omega; H; C; R \rightarrow \Xi'; \Delta'; \Gamma') &\quad \text{match}_{\text{LLD}} \odot \\
\text{match}_{\text{LLD}}(\Gamma; \Delta; \Xi; \Omega; H; C; R \rightarrow \Xi'; \Delta'; \Gamma') &\quad \text{match}_{\text{LLD}} \odot \\
\end{align*}
\]
C.4 Continuation

If the previous matching process fails, we pick the top frame from the stack \( C \) and restore the matching process using another fact and/or context. The continuation judgment \( \text{cont}_{LLD} C; H; R; \Gamma; \Xi'; \Delta'; \Gamma' \) deals with the backtracking process where the meaning of each argument is as follows:

\( C \) : Continuation stack.
\( H \) : Head of the current rule we are trying to match.
\( R \) : Next available rules if the current one does not match.

\[
\begin{align*}
\text{run}_{LLD} \Gamma; \Delta; \Phi & \rightarrow \Xi'; \Delta'; \Gamma' \\
\text{cont}_{LLD} : H; (\Phi, \Delta); \Gamma; \Xi'; \Delta'; \Gamma' \\
\text{match}_{LLD} \Gamma; \Delta; \Delta'; \Xi; p; \Omega; H; (\Delta, p; \Delta''; p, \Omega; H; \Xi; A; \Upsilon), C; R & \rightarrow \Xi'; \Delta'; \Gamma' \\
\text{cont}_{LLD} & \Gamma; \Delta; \Xi; (\Delta, p; \Delta''; p, \Omega; H; \Xi; A; \Upsilon), C; H; R; \Gamma; \Xi'; \Delta'; \Gamma' \\
\text{cont}_{LLD} & \Gamma; \Delta; \Xi; (\Delta, p; \Delta''; p, \Omega; H; \Xi; A; \Upsilon), C; H; R; \Gamma; \Xi'; \Delta'; \Gamma' \\
\text{match}_{LLD} & \Gamma; \Delta; \Xi; \Omega; H; (\Gamma'; \Delta; p; \Omega; \Xi; A; \Upsilon), C; R & \rightarrow \Xi'; \Delta'; \Gamma' \\
\text{cont}_{LLD} & \Gamma; \Delta; \Xi; (\Gamma'; \Delta; p; \Omega; \Xi; A; \Upsilon), C; H; R; \Gamma; \Xi'; \Delta'; \Gamma' \\
\text{cont}_{LLD} & \Gamma; \Delta; \Xi; (\Gamma'; \Delta; p; \Omega; \Xi; A; \Upsilon), C; H; R; \Gamma; \Xi'; \Delta'; \Gamma' \\
\end{align*}
\]

C.5 Derivation

Once the list of terms \( \Omega \) in the \( \text{match}_{LLD} \) judgment is exhausted, we derive the terms of the head of the rule. The derivation judgment uses the form \( \text{derive}_{LLD} \Gamma; \Delta; \Xi; \Gamma; \Delta; \Omega & \rightarrow \Xi'; \Delta'; \Gamma' \):

\( \Delta \) : Multi-set of linear facts we started with minus the linear facts consumed during the matching of the body of the rule.
\( \Xi \) : Multi-set of linear facts consumed during the matching of the body of the rule.
\( \Gamma_1 \) : Set of persistent facts derived up to this point in the derivation.
\( \Delta_1 \) : Multi-set of linear facts derived up to this point in the derivation.
\( \Omega \) : Remaining terms to derive as an ordered list. We start with \( B \) if the original rule is \( A \rightarrow B \).
C.6 Match Comprehension

The matching process for comprehensions is similar to the process used for matching the body of the rule, however we use two continuation stacks, C and P. In P, we put all the initial persistent frames and in C we put the first linear frame and then everything else. With this we can easily find out the first linear frame and remove everything that was pushed on top of such frame. The full judgment has the form

$$\text{match}_{\text{LLD}} \gamma; \Delta; \Xi \Xi; \Gamma'$$

\[ \Delta : \text{Multi-set of linear facts remaining up to this point in the matching process.} \]
\[ \Xi : \text{Multi-set of linear facts used during the matching process of the body of the rule.} \]
\[ \Gamma_{N1} : \text{Set of persistent facts derived up to this point in the head of the rule.} \]
\[ \Delta_{N1} : \text{Multi-set of linear facts derived by the head of the rule.} \]
\[ \Xi : \text{Multi-set of linear facts consumed up to this point.} \]
\[ \Omega : \text{Ordered list of terms that need to be matched for the comprehension to be applied.} \]
\[ C : \text{Continuation stack that contains both linear and persistent frames. The first frame must be linear.} \]
\[ P : \text{Initial part of the continuation stack with only persistent frames.} \]
\[ AB : \text{Comprehension} \comp A \rightarrow B \text{ that is being matched.} \]
\[ \Omega_N : \text{Multi-set of linear facts that were still available after matching the body of the rule.} \]

Note that \(\Delta, \Xi = \Delta_N\).
C.7 Match Comprehension Continuation

If the matching process fails, we need to backtrack to the previous frame and restore the matching process at that point. The judgment that backtrack has the form

$$\text{cont}_{\text{LLDC}}: \Gamma; \Delta N; \Xi N; \Delta N1; C; P; AB; \Omega N \rightarrow \Xi'; \Delta'; \Gamma'$$

- $\Delta N$: Multi-set of linear facts that were still available after matching the body of the rule.
- $\Xi N$: Multi-set of linear facts used during matching of the body of the rule.
- $\Gamma N1$: Set of persistent facts derived by the head of the rule.
- $\Delta N1$: Multi-set of linear facts derived by the head of the rule.
- $C, P$: Continuation stacks.
- $AB$: Comprehension $\text{comp} A \rightarrow B$ that is being matched.
- $\Omega N$: Ordered list of remaining terms of the head of the rule to be derived.

$$\text{match}_{\text{LLDC}}: \Gamma, \Gamma' ; p; A; \Xi N; \Gamma N1; \Delta N1; \Xi; \Omega; [\Gamma'; \Delta; \Xi; \Omega; A, \gamma, Y], \Delta C, C; P; AB; \Omega N; \Delta N \rightarrow \Xi'; \Delta'; \Gamma'$$

$$\text{match}_{\text{LLDC}}: \Gamma, \Gamma' ; p; A; \Xi N; \Gamma N1; \Delta N1; \Xi; \Omega; [\Gamma'; \Delta; \Xi; \Omega; A, \gamma, Y], \Delta C, C; P; AB; \Omega N; \Delta N \rightarrow \Xi'; \Delta'; \Gamma'$$

$$\text{match}_{\text{LLDC}}: \Gamma, \Gamma' ; p; A; \Xi N; \Gamma N1; \Delta N1; \Xi; \Omega; [\Gamma'; \Delta; \Xi; \Omega; A, \gamma, Y], \Delta C, C; P; AB; \Omega N; \Delta N \rightarrow \Xi'; \Delta'; \Gamma'$$
The judgment has the form \( \text{match}_{\text{LLD}} \Gamma; \Delta_N; \Xi; \gamma; \delta; \Theta; \omega; \eta \Gamma' \). The judgment has the form \( \text{cont}_{\text{LLD}} \Gamma; \Delta_N; \Xi; \gamma; \delta; \Theta; \omega; \eta \Gamma' \).

**C.8 Stack Transformation**

After a comprehension is matched and before we derive the head of such comprehension, we need to "fix" the continuation stack. In a nutshell, we remove all the frames except the first linear frame and remove the consumed linear facts from the remaining frames so that they are still valid for the next application of the comprehension. The judgment that fixes the stack has the form \( \text{fix}_{\text{LLD}} \Gamma; \Delta_N; \Xi; \gamma; \delta; \Theta; \omega; \eta \Gamma' \):

- \( \Xi_N \): Multi-set of linear facts used during the matching process of the body of the rule.
- \( \Gamma_N \): Set of persistent facts derived by the head of the rule.
- \( \Delta_N \): Multi-set of linear facts derived by the head of the rule.
- \( \Xi \): Multi-set of linear facts consumed by the previous application of the comprehension.
- \( C, P \): Continuation stacks for the comprehension.
- \( AB \): Comprehension comp \( A \rightarrow B \) that is being matched.
- \( \Omega_N \): Ordered list of remaining terms of the head of the rule to be derived.
- \( \Delta_N \): Multi-set of linear facts that were still available after matching the body of the rule.

**C.9 Comprehension Derivation**

After the fixing process, we start deriving the head of the comprehension. All the facts derived go directly to \( \Gamma_1 \) and \( \Delta_1 \). The judgment has the form \( \text{derive}_{\text{LLD}} \Gamma; \Xi; \Delta_1; \Theta; \omega; \delta; \Theta; \omega; \eta \Gamma' \).
Ξ : Multi-set of linear facts consumed both by the body of the rule and all the comprehension applications.
Γ₁ : Set of persistent facts derived by the head of the rule and comprehensions.
∆₁ : Multi-set of linear facts derived by the head of the rule and comprehensions.
Ω : Ordered list of terms to derive.
C, P : New continuation stacks.
AB : Comprehension comp A → B that is being derived.
Ωₐ : Ordered list of remaining terms of the head of the rule to be derived.
∆ₐ : Multi-set of remaining linear facts that can be used for the next comprehension applications.

\[
\text{derive}_{LLD} \Gamma; \Xi; \Gamma_1; \Delta_1; p; \Omega; C; P; \text{comp} \ A \rightarrow B; \Omega_N; \Delta_N \rightarrow \Xi'; \Delta'; \Gamma'
\]

\[
\text{derive}_{LLD} \Gamma; \Xi; \Gamma_1; \Delta_1; p; \Omega; C; P; \text{comp} \ A \rightarrow B; \Omega_N; \Delta_N \rightarrow \Xi'; \Delta'; \Gamma'
\]

\[
\text{derive}_{LLD} \Gamma; \Xi; \Gamma_1; \Delta_1; \cdot; \Omega; C; P; \text{comp} \ A \rightarrow B; \Omega_N; \Delta_N \rightarrow \Xi'; \Delta'; \Gamma'
\]

\[
\text{cont}_{LLD} \Gamma; \Delta_N; \Xi; \Gamma_1; \Delta_1; C; P; \text{comp} \ A \rightarrow B; \Omega_N \rightarrow \Xi'; \Delta'; \Gamma'
\]

\[
\text{derive}_{LLD} \Gamma; \Xi; \Gamma_1; \Delta_1; C; P; \text{comp} \ A \rightarrow B; \Omega_N; \Delta_N \rightarrow \Xi'; \Delta'; \Gamma'
\]

\[
\text{derive}_{LLD} \Gamma; \Xi; \Gamma_1; \Delta_1; C; P; \text{comp} \ A \rightarrow B; \Omega_N; \Delta_N \rightarrow \Xi'; \Delta'; \Gamma'
\]

\[
\text{derive}_{LLD} \Gamma; \Xi; \Gamma_1; \Delta_1; C; P; \text{comp} \ A \rightarrow B; \Omega_N; \Delta_N \rightarrow \Xi'; \Delta'; \Gamma'
\]

\[
\text{derive}_{LLD} \Gamma; \Xi; \Gamma_1; \Delta_1; C; P; \text{comp} \ A \rightarrow B; \Omega_N; \Delta_N \rightarrow \Xi'; \Delta'; \Gamma'
\]