Linear Logic and Narrative Programming

Henriette Vanvik Kopstad & Astri Marie Ravnaas

March 6, 2015

Abstract

This paper is a writeup on the subject Linear Logic Programming for the course 290C Programming Languages at UCSB. The purpose of this paper is to account for motivations for the use of linear logic when representing narratives with programming. To do so, we provide an introduction to linear logic, its main principles and role in programming, and how a program using linear logic can function. We then explain the principles of narratives and how linear logic is suitable for representing them in terms of resources. We provide simple and extended examples throughout the paper to better ensure the reader's understanding of the paper's topics.

1. Introduction to linear logic in programming

Formal logic has been a part of computer science for decades. Therefore, natural consequences of development within the understanding of logic are new advances within programming and computer science. Linear logic, proposed as a refinement of classical and intuitionistic logic, is no exception.

Written computer programs based on logic are comprised of sentences in logical form. The program itself is a set of facts and rules, while the computation is *proof search*[1]. In programs written using classical logic, the facts can be viewed as *truths* that can be used an unlimited amount of times during a proof search. In linear logic, however, facts are viewed as *resources*

that are consumed and produced at different points during a proof search, and can therefore not be used frivolously[2]. Hopefully we will make it apparent why this type of resource-sensitive nature makes linear logic a suitable model for narrative representations[3]. First we will give a clearer overview of linear logic through explaining its propositions.

2. Propositions in linear logic programming

As traditional and linear logic differ from each other in their way of viewing resources, so do their propositions. This section lists and explains the various connectors in linear logic used in implementation. Here, A, B and C range over propositions and X over propositional constants. Note that further clarifications on this subject can be found by viewing Figure 4 in the paper "A taste of linear logic"[2].

A, B, C ::= X | A⊸B | A⊗B | A&B | A⊕B | !A | @A | A ⊸ B

A⊸B, read as "A lollipop B" or "consuming A yields B, is linear logic's version of implication. In contrast to simple implication, lollipop here represents that *consumption* of A yields B. Resources on the left are consumed and resources on the right produced. As an example, consider the way you pay one dollar (A) to obtain a snack (B) and no longer have your dollar.

A⊗B, read as "A tensor B", means *both* A and B. In an example C⊸A⊗B, where consuming C yields A⊗B you obtain both A and B from the resource C.

A&B, read as "A with B", means your *choice* of A and B. For example, if you have one dollar and an apple and a banana cost one dollar each, you can choose between spending your dollar to obtain the apple or the banana.

A⊕B, read as "A or B", means either A or B can be obtained. To illustrate how this differs from &, think about the previous example with the banana and the apple. The proposition that you can buy either an apple or a banana for a dollar (A⊕B) still holds even if there are no more bananas left, but you no longer have your *choice* (A&B) of which fruit to buy.

!A, read as “bang A” or “of course A”, is a programmers way of allowing control over duplication of resources. !A produces a single “copy” of the resource A to be consumed, but does not limit further use of the resource. Put very simply, imagine yourself having an infinite number of dollar bills in your bank account. To buy an apple, you can produce *one* of these dollar bills from your account to obtain the apple. You will no longer have that particular dollar bill, but you can still produce another dollar from your account in the future.

@A, read “at A”, is called an *affine* resource, which *can* be used at most once, but does not have to be used at all. Let us say that in addition to dollar bills, you have a button in your pocket. Even though you have that resource it is certainly not necessary in order to obtain an apple. M and would in this example be considered an affine resource. Both @A and !A are the programmer’s ways of controlling resource duplication and discarding in linear logic programming while still getting all the advantages of linearity[2].

To round off this section, we wish to indicate the importance of noting that the lollipop behaves differently when used in a rule than in a goal. For simplicity, in this paragraph we will refer to lillipop in a rule as B □ A. In this context it signifies that if we want to prove B, we must prove A. In a goal however, R ⊸ G, it means we want to see *if* we can prove G, given that we have the resource A. To answer this question, one would use the appropriate *rules*. Because of the notation in our example later on in the paper, and because most documentation on the lollipop writes it “pointing” right, ⊸, that is how we represent it throughout this paper.

3. Using linear logic in narrative programming

Narratives are a way of representing time, actions and causality through the occurrence of narrative *actions* and the *changes* they cause in the environment. With linear logic programming, these actions are the program’s rules, and the changes are the consumption and production of its resources. This way, narratives are a way to represents *stories*, either interactive or with a pre-defined plot, through a computer program. Note that non-linear and linear stories are not connected to the “linear” in linear logic; a linear story simply means it has the same outcome

every time it is walked through. The goal of interactive storytelling (IS), however, is to create stories that can be modified in real-time to respond to user's reactions[4].

Some actions, in both stories and real life, are dependent on each other or can simply not be executed simultaneously due to the state of the environment and the changes the actions cause. For example, consider the task of making breakfast where you have enough ingredients for making an omelette *or* pancakes. You can not eat the pancakes before you make the batter out of eggs, flour and milk and fry it. If you have used your eggs to make pancakes, you can no longer make an omelette. This resource-sensitive characteristic of stories, makes linear logic a good choice of representing them as programs.

As stated earlier, computation of a program is proof search, and this still applies in narrative programming. Here, a program consists of an initial state, in addition to the rules and resources[3]. The initial state depicts what resources are available at the starting point of the story and the rules, or actions, are linear implications. Their left side indicates what resources are needed in order for the action to be performed and the right-side resources to be produced. The proof search is done through a query containing a goal state, and is therefore a way to determine if a certain state can be reached from the initial state through the valid actions. Section 4 will provide further explanation on the workings of proof searching in narratives with a simple example.

3.1 Proof Search

Given the goal state in the query, the program will try to find as many solutions as possible. A solution is only found if there are no linear resources left unconsumed during the proof search, as they *must* be consumed by definition. If one or more solutions are found, the program will also be able to provide the path it took to find it, namely the proof itself. A program might succeed in a variety of ways. The following example of a proof search is inspired by Chris Martens talk about linear logic programming[6].

3.2 Proof Search Strategies

There are two complementary, but not mutually exclusive, strategies for implementing proof search strategies using queries; backward chaining and forward chaining[6]. Backwards chaining will take the goal stated in the query and start looking for rules in the program that matches this

goal. This will give a new set of subgoals the algorithm will try to solve recursively until it reaches axioms. This is the default way Prolog does proof searches.

In forward chaining, the algorithm starts by looking at the program's axioms and then propagates forward. The program iteratively applies every possible rule in the rulebase to the axioms and then their new subgoals, until there are no more alternatives and applying rules no longer gives new information. Then the algorithm stops and presents all cases in which an axiom led to the goal state from the query. This proof search strategy is the one we will focus on in this paper.

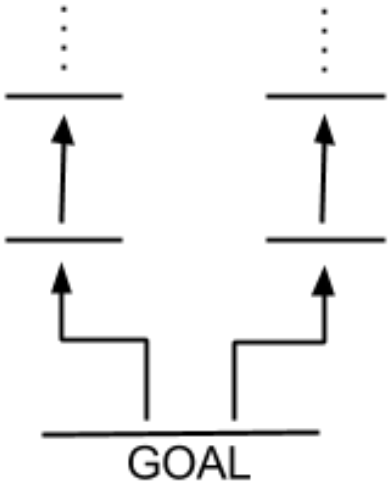

Figure 1: Backward chaining

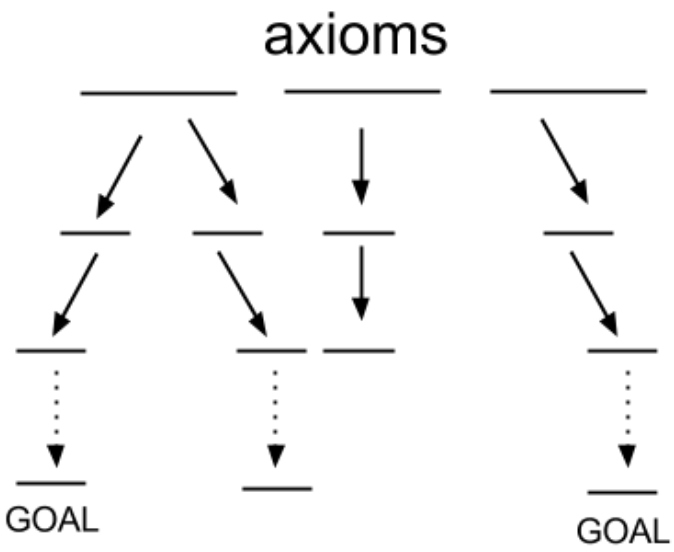

Figure 2: Forward channing

3.2.1 Forward Chaining Example

To give a high level understanding of how proof searches and forward chaining works, we have provided an example of a program that computes graph connectivity. The rules are as follows for edges and nodes:

1. If there is an edge from X to Y, then there is an path from X to Y.
2. If there is an edge from X to Y and an edge from Y to Z then there is a composed path from X to Z (transactivity rule).

The query in our example is searching for the path (a,X). This is supposed to return all of the paths that are reachable from a.

Query	?- path(a, X)	

First, all of the edge facts are put into a database.

Database	edge(a,b) edge(b,c) edge(c,d)	

Then, all applicable rules are applied to the edge facts. Here; paths correspond to the edges.

Apply every rule to database	path(a,b) path(b,c) path(c,d)	**X = b**

This is repeated where we generate the path facts that now can be generated from the previous edge facts and the new path facts. Which gives us the two hop away connectivity.

Again, apply rule to database	path(a,c) path(b,d)	**X = c**

This process is repeated until *saturation*. Saturation is the point at which applying another rule in our system will not give any new information.

Saturation	path(a,d)	**X = d**

At saturation we can go back and look through. We can notice that b,c and d are all values of X.

4. Narrative Examples

As previously stated, narratives can be both interactive (non-linear) or pre-defined with a plot (linear). We chose to focus on non-linearity as it has several possible application areas, such as interactive books and games. Recently, interactive storytelling has become a subject of interest within Artificial Intelligence, as researches recognize its components as compatible with their research. Storytelling systems typically use a *planner*[5] to generate a sequence of actions dynamically, which is a commonplace problem in AI. The goal of a planner in AI is to determine

whether a goal state can be achieved given an initial state and set of actions that can be performed. By now this may sound familiar, and shows how linear logic programming can be applied to problems even within Artificial Intelligence. Below is a short example of how linear logic can be used to represent a non-linear narrative.

4.1.1 Non-linear example: Adventurer in the woods

In our example we have taken a possible excerpt from a fictional interactive game where the player is an adventurer who has entered a forest. He has a choice to either follow the road in front of him leading to the right, or take a path to his left. If he goes to the left, he will encounter a monster that will kill him. If he follows the road, he will be told of a treasure inside a large castle that can only be obtained if he finds the answer to a riddle. The road will lead to the castle which he will have to enter either by using his crowbar to break in, or try a key he found earlier in the game, which turns out to unlock the door. He must also get the answer to the riddle he was told of and say out loud for the treasure chest to open. The player can get this answer by summoning an all-knowing wizard to help him any time during the game.

We have used a acyclic graph to illustrate our example. The init-state represents the initial state of the narrative. The other nodes are narrative actions and the edges represent inferred causality relationships. Resources the actions must consume are written by the arrows leading in to the action node. As we can see from the graph, the story can have two different outcomes; getting treasure or dying. There are also two possible ways of obtaining the treasure because of the different ways there are to enter the castle.

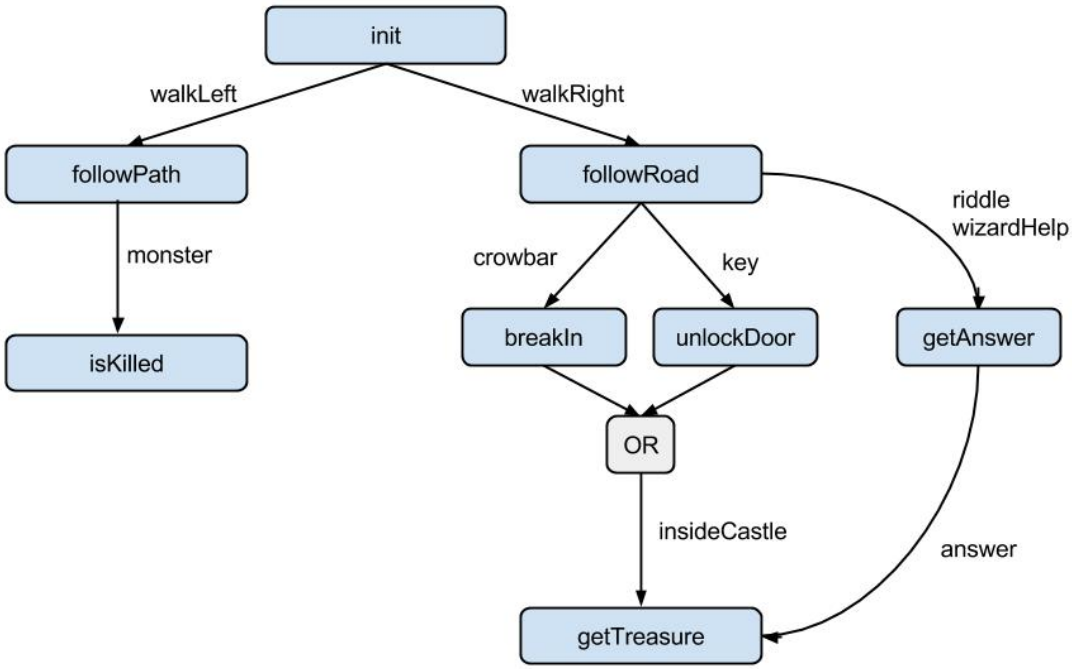

Figure 3: Non-linear example

4.1.2 Code Example

There are a few programming languages that are based on Linear Logic. Lolli [7] is one such language, and uses backward chaining. LolliMon [8] and Celf [9] are more recent languages that extend Lolli, where both the forward and backward chaining phases may be controlled by the programmer. In this paper we have chosen to use forward chaining for our proof searches. Figure 4 shows an example of a Celf program for the story of the adventurer in the acyclic graph from Figure 3.

```
1 % Narrative resources
2  walkLeft      : type.
3  walkRight     : type.
4  key           : type.
5  crowbar       : type.
6  door          : type.
7  insideCastle  : type.
8  wizardHelp    : type.
9  riddle        : type.
10 answer        : type.
11 monster       : type.
12 treasure      : type.
13 dead          : type.
14
15 % Actions
16 isKilled    : type =   monster -o { @dead }.
17 getTreasure : type =   insideCastle * answer -o { @treasure }.
18 unlockDoor  : type =   key * door -o { insideCastle * @key * @getTreasure }.
19 breakIn     : type =   crowbar * door -o { insideCastle * @crowbar * @getTreasure }.
20 getAnswer   : type =   riddle * wizardHelp -o { @answer }.
21 followRoad  : type =   walkRight -o { door * riddle * @breakIn * @unlockDoor }.
22 followPath  : type =   walkLeft -o { monster }.
23
24 % Initial Environment
25 init : type = { (walkLeft & walkRight) * @key * @crowbar * @followRoad *
26                @followPath * @wizardHelp * @getAnswer * @isKilled }.
27
28 % Query
29 #query * * * 20 (init -o { treasure }).
```

Figure 4: Celf code

The lines 1-13 is the atomic types that correspond to to the atomic resources in the narrative. Line 15-22 describes narrative actions and what resources they consume and produce. Line 25-26 shows the initial state. As mentioned in the previous section the initial state represents the resources available at the starting point of the game and the available actions. The query in line 29 is defining the problem which Celf will try to find a solution to, in this case; find out if there is a way to get from the initial state to getTreasure.

From the code and the graph in Figure 3 one can see that getting from init to getTreasure is possible as long as the player chooses to go right, has the answer to the riddle and gets into the castle in one of two ways. There are no linear resources left unconsumed at this point, only the resources with a bang in front of them, and perhaps affine resources. This is fine as the bang-rules can not be removed by consumption, and the affine rules do not *have* to be consumed, so the proof holds. In fact, if the query were to look for a way from init to isKilled, this also turns out to be a valid goal state with no unconsumed linear resources.

We want to emphasize that when the lollipop is used in the narrative actions (line 15-22), it is in a *rule*, and represents *consumption* in the same way as we described it in section 2. In the query (line 29) however, the lollipop is used in a goal, and indicates that the query is asking *if* it is possible to obtain treasure given the init state.

5. Conclusion

Narratives are stories; a continuous chain of events that have consequences on the world around them. Certain actions are mutually exclusive, require certain other events or environmental factors in order to occur or have unique consequences. Even though programming doesn't represent time in the same sense as we know it, it can be represented by sequences of events and the world and environment can be represented as states and attributes. Incorporating linear logic when programing narratives gives an easy way to control the restrictions and causality that actions have on the environment, due to its resource focused nature.

7. References

[1] Frank Pfenning, Linear Logic, draft 2002, NSF Grants CCR-9303383 and CCR-9619684

[2] Philip Wadler, A taste of linear logic, University of Glasgow, G12 8QQ, Scotland

[3] Anne-Gwenn Bosser, Teesside University João F. Ferreira, Teesside University Teesside University, Marc Cavazza, Teesside University Chris Martens, Carnegie Mellon University, A taste of linear logic

[4] Anne-Gwenn Bosser, Teesside University, Marc Cavazza, Teesside University, Ronan Champagnat, La Rochelle University, Linear Logic for Non-Linear Storytelling

[5] Craig Boutilier, University of Toronto, Logical Representations and Computational Methods for Markov Decision Processes, 2002, NASSLI Lecture Slides

[6] Chris Martens, Carnegie Mellon University, Linear Logic Programming, Strange Loop 2013

[7] Hodas, J.S., Miller, D.: Logic programming in a fragment of Intuitionistic Linear Logic. Information and Computation 110(2) (1994) 327–365

[8] López, P., Pfenning, F., Polakow, J., Watkins, K.: Monadic concurrent Linear Logic programming. In: Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming. (2005)

[9] Schack-Nielsen, A., Schürmann, C.: Celf — a logical framework for deductive and concurrent systems (system description). In: Automated Reasoning. Springer (2008) 320–326

Attachment 1: Introduction to Celf

To help our fellow classmates we have provided a basic explanation on how Celf works. In Celf, forward and backward chaining phases can be controlled by the programmer. The language can be found and downloaded here: http://clf.github.io/celf/. Celf uses a different notation on some of the connectives. The table under provides an overview of the changes:

Abstract syntax	Concrete syntax	Meaning
$A \multimap B$	A -o {B}	replacement
$A \oplus B$	A * B	conjunction of resources

Recall that a Celf program is structured as follows:

- **Atomic types** (lines 2-13 in Celf example) correspond to the atomic resources in the narrative.
- **Asynchronous types** (lines 16-22) consist of linear implications. This is expressed in Celf as A1 * ... * An -o {B1 * ... * Bn}. Resources on the left are consumed and resources on the right produced.
- **The initial state** represents the resources available at the starting point of the game and the available actions.
- **The query** line is defining the problem which Celf will try to find a solution to. A standard query is set up in this format: `#query d e l a ty` where *d, e, l, a, ty* are its arguments. *d* is the let-depth-bound of the proof search, *e* is expected number of solutions the search will give, *l* is number of solutions to look for, *a* is the number of times to execute the query and *ty* is the goal which Celf is going to find a solution to. Not all the arguments need to be given a value. If you want to omit one or more, the argument can be written with a star, *, as its value instead. The reason for specifying the number of iterations of the query, *a*, is because in forward chaining, there is no backtracking if the program does not find a solution on a given path. Instead the programmer must specify it to execute again.

Interpreting the output

To explain how to interpret the output of running a Celf program, we continue with the example shown in Figure 4. Once you run the program, most likely in your terminal, the output will look something like this:

```
init: Type = {(walkLeft & walkRight) * (@key * (@crowbar * (@followRoad *
(@followPath * (@wizardHelp * (@getAnswer * @isKilled))))))}.
Query (*, *, *, 20) init -o {treasure}.
Iteration 1
Iteration 2
Iteration 3
Solution: \X1. {
      let {[X2, [@X3, [@X4, [@X5, [@X6, [@X7, [@X8, @X9]]]]]]]} = X1 in
      let {[X10, [X11, [@X12, @X13]]]} = X5 (X2 #2) in
      let {[X14, [@X15, @X16]]} = X12 [X4, X10] in
      let {@X17} = X8 [X11, X7] in
      let {@X18} = X16 [X14, X17] in X18}
Iteration 4
Solution: \X1. {
      let {[X2, [@X3, [@X4, [@X5, [@X6, [@X7, [@X8, @X9]]]]]]]} = X1 in
      let {[X10, [X11, [@X12, @X13]]]} = X5 (X2 #2) in
      let {[X14, [@X15, @X16]]} = X13 [X3, X10] in
      let {@X17} = X8 [X11, X7] in
      let {@X18} = X16 [X14, X17] in X18}
Iteration 5
Solution: \X1. {
      let {[X2, [@X3, [@X4, [@X5, [@X6, [@X7, [@X8, @X9]]]]]]]} = X1 in
      let {[X10, [X11, [@X12, @X13]]]} = X5 (X2 #2) in
      let {[X14, [@X15, @X16]]} = X13 [X3, X10] in
      let {@X17} = X8 [X11, X7] in
      let {@X18} = X16 [X14, X17] in X18}
                                      ...
                                      ...
                                      ...
                                      ...
                                      ...
```

First, observe that the first two iterations do not result in a solution, but that is no indication of lack of possible solutions as we can see from iteration 3, 4 and 5. The way to read this may seem complicated at first glance, but there is an order to the chaos.

Every `X`-value uniquely represents a resource or rule. We will look at the solution from the third iteration for this explanation. The first `X`-value, here `X1`, always represents the initial state. Let's review the first line of the solution:

```
  let {[X2, [@X3, [@X4, [@X5, [@X6, [@X7, [@X8, @X9]]]]]]]} = X1 in
```

As we know, `X1` is the initial state. The arguments within the curly braces represent the resources and actions defined in this initial state, in the same order as they were listed. Currently, this gives us these mappings of `X`-values to resources/states:

```
X1: init            X2:!(walkLeft & walkRight)    X3: key
X4: crowbar         X5: followRoad           X6: followPath
X7: wizardHelp      X8: getAnswer            X9: isKilled
```

Note that `!(walkLeft & walkRight)` is considered one resource for now. In the following line:

```
let {[X10, [X11, [@X12, @X13]]]} = X5 (X2 #2) in
```

and all the following lines, read the `X`-value directly to the right of the "=" as an action that is performed; here it is `X5`, which we now know is `followRoad`. `X`-values following the action variable, (`(X2 #2)` in this case), represent the resources it consumes. The values to the left of the "=" represent the resources that are produced by the action. Because we know that `X5: followRoad`, we can look at the program's code to map the input and output variables:

```
(X2 #2): walkRight       X10: door      X11: riddle
X12: breakIn             X13: unlockDoor
```

Notice how `(X2 #2)` shows how choices are made from an `X`-value containing an "&". Repeat this process for the following line

```
let {[X14, [@X15, @X16]]} = X12 [X4, X10] in
```

gives us (because we know `X12: breakIn` is the action performed and we know its in- and outputs):

```
X14: insideCastle        X15: crowbar
X16: getTreasure
```

Why did `breakIn` give `X15` as `crowbar` when `crowbar` was already defined as `X4`? Because no two linear resources are the same. `X4` was consumed by `breakIn` and ceased to exist, but even if it hadn't, and only had *produced* a `crowbar`, this `crowbar` would not have been the same as the one represented by `X4`.

Repeating this process on all the lines in iteration 3 gives us the following actions that lead from `init` to `treasure` in chronological order:

```
followRoad, breakIn, getAnswer, getTreasure.
```

Sure enough, by reviewing the diagram in Figure 3, we see that these actions in this order lead from the initial state to obtaining the treasure. The other iterations may find different solutions, but are all read the same way.