Linear Logic Programming for Narrative Generation

Chris Martens[1], Anne-Gwenn Bosser[2], João F. Ferreira[2], and Marc Cavazza[2]

[1] Carnegie Mellon University
[2] Teesside University

Abstract. In this paper, we explore the use of Linear Logic programming for story generation. We use the language Celf to represent narrative knowledge, and its own querying mechanism to generate story instances, through a number of proof terms. Each proof term obtained is used, through a resource-flow analysis, to build a directed graph where nodes are narrative actions and edges represent inferred causality relationships. Such graphs represent narrative plots structured by narrative causality. This approach is a candidate technique for narrative generation which unifies declarative representations and generation via query and deduction mechanisms.

Keywords: Linear Logic Programming, Narrative Modelling, Celf

1 Introduction

Linear Logic [5] has recently been proposed as a suitable representational model for narratives [2]: its resource-sensitive nature allows to naturally reason about narrative actions and the changes they cause in the environment. In this paper, we explore Linear Logic programming as a tool for narrative representation and narrative generation. We describe how initial circumstances and narrative actions can be declared in the Linear Logic programming language Celf [11] and how using Celf's search mechanism allows the generation of proof terms which can be interpreted as causally structured narrative plots. To improve narrative analysis, we developed a prototype front-end to Celf. We illustrate how to use story material to program and generate a variety of plots using the novel *Madame Bovary* [4]: its narrative causal structure has been emphasized in Flaubert's working material [8]. Preliminary results are encouraging, allowing the generation of story variants through a methodical programming approach.

2 Related Works

Narratives have always been an important topic for research in AI for their role as knowledge structures [12] and recent years have seen the widespread adoption of planning techniques for the construction of narrative generation systems [14], mostly because they support the representation of causality. Linear

Logic provides an expressive model of action and change (information revision is dealt with at the level of the logical rules through the linear implication) which has led previous work to explore its suitability for narrative representation using a story-as-proof analogy [2]. The intractability of proof search in expressive fragments of Linear Logic has led to the use of a proof assistant [3] for story generation and for evidencing properties transcending all narratives in a semi-automated manner. Support of narrative causality at the logical level is also an advantage when compared with standard logic programming approaches to narrative generation [13]. LolliMon [9] and Celf [11] are recent systems that have extended Lolli [7] (which follows a goal-directed backward proof-search interpretation in the intuitionistic fragment of Linear Logic) and where forward and backward chaining phases may be controlled by the programmer using a monad. We refer the reader to [10] for an overview and application survey.

3 Programming a Narrative

3.1 A Celf Program describing a narrative

Celf[3] [11] uses dependent types for the representation of logical predicates; this approach to logic programming means that the result of a query is a *term* of the corresponding *type*, which can be analysed as a computational artefact. Celf programs are normally divided into two main parts: **a signature**, which is a declaration of type and terms constants describing data and transitions, and **query directives**, defining the problem for which Celf will try to find solutions (proof terms showing that a given type is inhabited).

The technique used by Celf to compute proof terms is called focusing, based on the foundations of *Focused Linear Logic* [1] interpreted as Monadic Concurrent Logic Programming [9]: Celf gives the programmer control over when to enter a forward-chaining phase, which may use synchronous connectives, through the use of a monad (denoted using curly brackets {...}). The search triggered by a query in Celf begins in a backward-chaining phase using the query type as its goal, and if that type includes a monadic expression, it will enter a forward-chaining phase. This phase is implemented with a committed choice semantics, backtracking over the selection of a rule only when its antecedents cannot be met—effectively inducing a random choice between all fireable rules on each forward chaining step. This built-in nondeterminism lets us go automatically from a *specification* of a narrative structure to the automatic generation of stories.

3.2 Identification of Narrative Elements

The process of programming a narrative is that of describing circumstances that can, by execution of the program, generate one or many stories. Following a widespread paradigm in narrative generation research, we use an existing, linear, baseline story to support our experiments. Identifying the circumstances

---

[3] The Celf system can be obtained from https://github.com/clf/celf

```
 1  emma : type.
 2  emmaCharlesMarried : type.
 3  <............>
 4  arsenic : type.
 5  emmaIsDead : type.
 6  emmaSpendsYearsInConvent : type = emma * convent -o {!novels * !grace * !
        education * @emma}.
 7  emmaMarriesCharles  : type = emma * escapism * grace * charles -o {
        emmaIsBored * @emma *!emmaCharlesMarried}.
 8  emmaDoesNotGoToBall : type = emma * ball -o {emmaIsBored * @emma}.
 9  < ......>
10  emmaContractsDebts : type = emma * emmaIsBored -o {@debt * @emma}.
11  emmaGetsSick : type = emma * emmaIsDespaired -o {@debt * @debt * @debt *
        @debt * !charlesIsConcerned * @emma}.
12  emmaJumpsThroughWindow : type = emma * emmaIsDespaired * emmaRebels -o {
        @emmaIsDead}.
13  emmaLearnsBovaryFatherDeath : type = emma * leonEmmaTogether *
        charlesIsConcerned * homais -o {@arsenic * @inheritance *
        @leonEmmaTogether * @emma}.
14  emmaCommitsSuicide : type = emma * ruin * arsenic * emmaRebels -o {
        @emmaIsDead}.
15  init : type =
16   { convent * @emma * @leonIsBored * !charles * !rodolphePastLoveLife * !
        homais
17          * @emmaSpendsYearsInConvent
18          * @(emmaGoesToBall & emmaDoesNotGoToBall)
19          <.......>
20          * !emmaContractsDebts
21  }.
22  #query * * * 100 (init -o {emmaIsDead}).
```

Fig. 1. Celf excerpt for a fragment of *Madame Bovary*. Atomic types (narrative resources) are followed by types describing narrative actions, the initial environment declaration and a query of 100 attempts to generate stories ending with Emma's death. The complete file (105 lines of code) is available on https://github.com/jff/TeLLer.)

within a static story such as Madame Bovary [4] is a human activity that can be assisted by companion works [8]. Figure 1 shows an example of a Celf program representative of the form we use to model narratives, and composed of a signature and a query (line 22).

The narrative elements we identify and model fall into two main categories. **Narrative resources** are available story elements (including characters) as well as states of the story, which may be related to characters and motives. In the present example, we model them using **atomic types** (lines 1–5). **Narrative actions** are transforming events occurring in the narrative. We model the impact they have on the narrative, in terms of resource creation and consumption using asynchronous types (lines 6–14), here linear implication formulae.

The type `init` on line 15 describes the initial narrative environment. Resources can be introduced as a) linear (default): there is one copy in the initial environment, and it will need to be consumed for any computation to terminate successfully. Emma's boredom is modelled as linear, since one of the driving force for her actions in the story is to escape this state; b) affine (using `@`): there is initially one copy in the initial environment and it may or may not be consumed by a successful computation. Because Emma may die in the story, the corresponding resource is introduced as persistent; c) persistent, (using `!`): there are arbitrarily many copies in the initial environment and any number of them

may be consumed by a successful computation. We use this to denote immutable facts and hard rules, such as Emma and Charles married status for instance.

In addition to the author's notes [8] for filtering through story events irrelevant for the modelled narrative structure, we proceed iteratively, and lazily model a new resource when we model a narrative action involving it. The narrative action corresponding to Emma taking arsenic to poison herself illustrates this process: Emma (returning late from a date with Leon) learns about her father's death from Homais because Charles is afraid to upset her. She learns about inheritance. We first model:

```
emmaLearnsBovaryFatherDeath : type = emma * leonEmmaTogether *
    charlesIsConcerned * homais -o {@inheritance * @leonEmmaTogether * @emma}
```

During the same event, a side conversation occurs between two of the characters present during which Emma incidentally learns where to find arsenic. The importance of this knowledge becomes apparent only when we model the narrative action corresponding to Emma's death. We then modify the code so that the action adds the corresponding resource to the environment and obtain the code on lines 13 and 14.

Mutually exclusive narrative actions can be explicitly suggested using the choice connective & in the declaration of the initial conditions. These can be used to encode key turning points in the narrative that are broadly recognized as such, which is frequently the case when using existing stories as a baseline. We use this connective to model Emma's choice to attend the ball (see line 18).

Once the narrative is modelled, it is run using Celf and the proof-terms obtained are post-processed for causality analysis. Following a long tradition of analysing causality via graphs [6], we developed a prototype tool, *CelfToGraph*[4], that automatically transforms proof terms generated by Celf into directed acyclic graphs. Such graphs represent narrative plots, structured by narrative causality, where nodes are narrative actions and edges represent inferred causality relationships.

One advantage of modelling narratives using a programming language is the ability to iteratively fine tune the model: a programmer alternates between coding and testing phases, which is facilitated by the frontend that we developed: in addition to the generation of causal graphs representing narratives, *CelfToGraph* queries can exhibit plots with specific characteristics. One can also verify if the generated set has a varied output (differing significantly from the original plot), test the impact of more **narrative drive** on the generation (for instance by comparing the effect of affine vs. linear models of narrative actions), or fine-tune resource threshold quantities.

3.3 Generated Plots

The entire code corresponding to the excerpt on Figure 1 consists of a total of 105 lines, including 31 narrative action descriptions. As we have only explicitly

---

[4] *CelfToGraph* requires Celf v2.9 and is available at https://github.com/jff/TeLLer
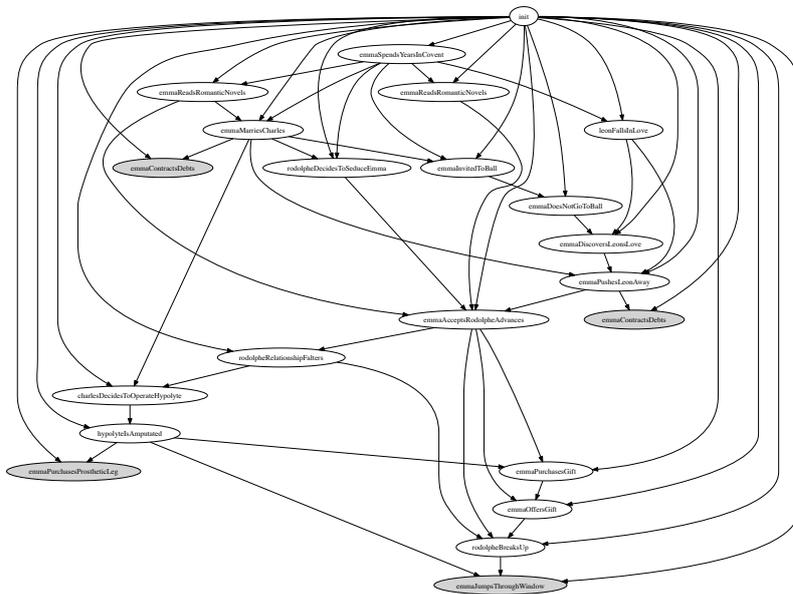
Fig. 2. One of 41 causally structured generated plots exhibited using *CelfToGraph*. In this variant, Emma does not attend the ball and defenestrates when left by Rodolphe.

encoded one branching choice, the variety of outputs is due to the linear semantics of narrative actions (producing resources that may be contended) and forward chaining variability.

The code described allows to **generate** 72 different narrative sequences for 100 attempts. After an automatic comparison of the corresponding plots using *CelfToGraph*, , we can exhibit 41 different plots (characterised by different generated causal structures), meaning that a number of different narrative sequences share the same causal structures. This allows the characterisation of classes of true **story variants**. Figure 2 shows a story variant among those generated, which has been exhibited by the tool: in this story, Emma jumps through the window following the departure of Rodolphe. If we look at the code in Figure 1 (l. 11 and l. 12) two narrative actions consume the resource `emmaIsDespaired`. When the first is triggered by the forward chaining mechanism, we obtain a story ending with Emma jumping through the window. When requesting 1000 query attempts, we obtain 747 solutions, among which 697 are different narrative sequences, and 226 true story variants.

4 Conclusion

There has been much interest in the use of Linear Logic to represent natural language semantics and the semantics of action and change. Narrative struc-

tures are based on the integration of the above phenomena, and Linear Logic programming provides a direct mechanism to operationalize these descriptions.

Our first results reported here are clearly encouraging, offering all the benefits of a declarative representation. This opens perspectives for applications such as Interactive Storytelling, where narrative generation is a default interaction paradigm, allowing narratives to adapt to changes in the environment.

In future work, we intend to develop this approach with the definition of an interaction paradigm using Linear Logic's choice connectives and on-the-fly environment modifications. Another interesting line of inquiry would be to explore the possible definition of normal forms for stories generated.

References

1. Andreoli, J.: Logic programming with focusing proofs in Linear Logic. Journal of Logic and Computation 2, 297–347 (1992)
2. Bosser, A.G., Cavazza, M., Champagnat, R.: Linear Logic for non-linear storytelling. In: ECAI 2010. Frontiers in Artificial Intelligence and Applications, vol. 215. IOS Press (2010)
3. Bosser, A.G., Courtieu, P., Forest, J., Cavazza, M.: Structural analysis of narratives with the Coq proof assistant. In: ITP (2011)
4. Flaubert, G.: Madame Bovary. Revue de Paris (1857), edition 2001 Collection Folio Classiques, ISBN 9782070413119.
5. Girard, J.Y.: Linear Logic. Theoretical Computer Science 50(1), 1–102 (1987)
6. Greenland, S., Pearl, J., Robins, J.: Causal diagrams for epidemiologic research. Epidemiology pp. 37–48 (1999)
7. Hodas, J.S., Miller, D.: Logic programming in a fragment of Intuitionistic Linear Logic. Information and Computation 110(2), 327–365 (1994)
8. Leclerc, Y.: Flaubert, Plans et Scénarios de Madame Bovary. Zuma, Cadeilhan (1995)
9. 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)
10. Miller, D.: Overview of Linear Logic programming. Linear Logic in Computer Science 316, 119–150 (2004)
11. Schack-Nielsen, A., Schürmann, C.: Celf — a logical framework for deductive and concurrent systems (system description). In: Automated Reasoning, pp. 320–326. Springer (2008)
12. Schank, R., Abelson, R.: Scripts, plans, goals and understanding: An inquiry into human knowledge structures. Psychology Press (1977)
13. Schroeder, M.: How to tell a logical story. In: Narrative Intelligence: Papers from the AAAI Fall Symposium. AAAI Press (1999)
14. Young, R.M.: Notes on the use of plan structures in the creation of interactive plot. In: Narrative Intelligence: Papers from the AAAI Fall Symposium. AAAI Press (1999)