Languages for Computational Creativity

Generative Art and Interactive Worlds (Full Presentation)

Chris Martens

Carnegie Mellon University

cmartens@cs.cmu.edu

Abstract

I propose that programming language researchers join me in exploring what our tools have to offer the practice of *programming creativity*, i.e. writing programs that aim to be interesting, surprising, and artful. One incarnation of this idea is *simulation* wherein the programmer may describe a world and multiple agents that interact within it. A next step is adding *interactivity*, wherein humans and computers collaborate to form a creative work or a dialogue, e.g. by the human participant manually introducing new constraints or goals into a simulation. In the former setting we are more interested in the artifact whereas in the latter we are more interested in the process. Both of these settings ask questions about *knowledge representation* and *conditional actions*, for which I believe the PL formalisms have been underexplored.

This proposal consists of (1) an argument for using linear logic to describe generative systems; (2) a discussion of linear logic's applicability to describing human interaction; (3) motivation for looking beyond linear logic for alternative "action languages."

1. Generative Media

One role that computation can play in the creation of art is by generating it systematically. Markov models and context-free languages have enjoyed wide use as *generative grammars* for poetry, amusement, and experimentation in addition to more pragmatic tasks in computational linguistics. In 1968, Vladimir Propp proposed a grammar for Russian folktales [4], which researchers have since explored computationally (e.g. [2]).

A step beyond these simple, stateless "choice tree" models of generativity lies multiset rewriting systems [1], which can be embodied and animated by a linear logic programming system. I will sketch how linear logic programming can be used to generate media.

1.1 Proofs as Stories

The author has been investigating the use of Celf [5], an implementation of linear logic programming, toward generative media, specifically for generating *stories* from an establishment of narrative elements such as characters and setting, along with rules for how those elements may interact.

We can write general rules for how characters may interact with each other in a story world, such as one character stealing from another:

```
do/steal
  : actor C' * actor C *
    at C L * at C' L * has C O * wants C' O
         -o {at C L * at C' L * has C' O * anger C C'}.
```

This rule permits a part of the story state represented by the antecedent (facts before the `-o`) to be replaced by the state represented by the succedent (facts after the `-o`). Importantly, rules written in this system obey a *frame rule*, meaning that whatever else holds true in the story–other characters' affections or dislikes for one another, or perhaps state pertaining to entirely separate storylines–does not change when the rule is applied.

Using a nondeterministic semantics for rule selection, we can run this set of rules by querying a final state matching all possible outcomes, and because the system is based on constructive logic, we wind up with a *proof* of the query proposition corresponding to a *story* generated by the system.

Interest in generated stories seems alive and well among creative programmers, what with the 2013 creation of NaNoGenMo,[1] a spinoff of NaNoWriMo (National Novel Writing Month) in which participants must programmatically *generate* 50,000 words of story.

Reviewers are encouraged to look at a Celf program written by the author[2] which generates the skeletal structure of stories atop which one could apply natural-language rendering of events toward this end. These skeletons can nonetheless be read by a human; sample output for this generator can be found in the same project repository.[3]

1.2 Games: Levels, Characters, and Puzzles

Generativity holds particular interest for game designers who aim to create a sense of *rule-based surprise* in their gameplay. Suppose a game consists of levels that take place on a 2D grid, as do many classics. A designer may lovingly craft every level by hand; or she may set constraints (e.g., which kinds of tiles may go next to each other, or that an exit must always be reachable) and let a program randomly generate levels; or she may combine these approaches. Games in the "roguelike" genre embrace this idea, presenting the player with entirely randomly generated levels, including placement of enemies, treasure, and level exits (again, according to some constraints).

Apart from level generation, one may want to generate characters in a roleplaying game or puzzles in a strategy game. The opportunities for programmatic content generation in games settings abound.

1.3 Other Media

Finally, traditional media such as poetry, illustration, and music can be automatically generated using these techniques. The rules–the constraints–that the program author chooses to codify still represent human choices about the

[1] `https://github.com/dariusk/NaNoGenMo`

[2] `https://github.com/chrisamaphone/interactive-lp/blob/master/examples/romeo.clf`

[3] Look at the part beginning with "Iteration 1": `https://github.com/chrisamaphone/interactive-lp/blob/master/examples/romeo.out`

structure of the composition. For example, a haiku has some amount of creative choice in terms of specific words and subject matter, but it also has formal constraints about the syllabic content of words. In music, consider the twelve-tone row, in which every note in a chromatic scale is used once before repeating any such note, with exceptions for adjacent notes in the composition.

The French intellectual community known as Oulipo[4] famously created constrained works with mathematical structure. With compuational approaches to creating and animating such structures, which programming language researchers have been doing outside of creative domains for decades, I believe we can only create more dynamic and interesting artworks.

2. Interactive Worlds

Once we have established a rich environment in which programmed entities may interact, a natural direction to take (especially in the design of digital games) is to allow the player to control one or more of these entities. This idea turns out to be easier to conceive than to implement. For one thing, at which points in the program's execution ought we allow human intervention?

One answer is that we introduce a linear resource at the beginning of every rule corresponding to a player action. For this to make sense, we have to decide what counts as an action. We can modify the previous rule example, replacing the two `actor` premises with a specific player action:

```
do/steal
  : player_action(Player, steal_from C O) *
    at C L * at Player L * has C O
  -o {at C L * at Player L * has Player O * anger C Player}.
```

Note that we no longer need to codify the motivation `wants Player O` as a premise to the rule, since we rely on the player herself to provide that impetus. The author's thesis proposal [3] aims to introduce this form of interactivity into the Celf system via linguistic mechanisms added around the scaffolding of linear logic.

Outside the realm of programming languages research lie systems like the Versu environment for interactive fiction.[5] In Versu, characters have interiority (personalities, motivations, sentiments) that can be exposed through *dialogue actions* constrained by other elements of the state. The player may select any character to control, and she may simply observe the conversation or interject by choosing from a range of context-dependent verbalizations. Each scenario in Versu is at once an open-ended simulation and a carefully-crafted narrative; the author's voice emerges from both the particular rendering of dialogue (to which the underlying system is agnostic) and the systematic constraints that she has imposed on the story.[6]

In domains other than games, character improvisation has obvious applications to theater, and Continuing this thought in the domain of generative music suggests the possibility of a "jam band" improvisational score that reacts to the sounds in the environment. And in the games domain, one could consider the design of levels that would adapt (either helpfully or adversarially) around the player.

[4] `http://en.wikipedia.org/wiki/Oulipo`

[5] `http://www.versu.com/`

[6] `http://emshort.wordpress.com/2013/02/26/versu-conversation-implementation/`

3. Action Languages

Despite the idealism permitted by the vagueness of the previous sections, linear logic does not present an ideal formalism for describing interaction with generative worlds. Some common patterns that emerge are:

- **Negation.** Purely proof-theoretic systems do not easily account for the negation of formula, i.e. testing for the absence of a particular fact in the state. One can easily imagine use cases for such a thing.
- **Comprehensive production (broadcast) or consumption.** Consider that in a conversation simulation, we may want a rule that, whenever a character speaks, causes every character who can hear her to be affected (e.g. by forming an opinion of the speaker). In linear logic the natural possibility would be

  ```
  speaks(C) -o {ForAll x.inEarshot x C -o {formOpinion x C}}.
  ```

  However, the precise semantics of the universal quantifier in Celf permits only one instantiate for the variable– and worse, it might apply the rule (by nondeterministic choice) to a character for whom the `inEarshot` premise does not hold.

These patterns emerge commonly enough, and are painful enough to specify in linear logic-based systems, that I claim we need to look beyond linear logic for the design of action languages. Systems with temporal indices such as event calculus[7] may serve as inspiration, though ideally one would like to preserve the frame property afforded by linear logic over such systems.

One reason to stay within the domain of strictly proof-theoretical systems is that they give us a notion of structural execution trace for free. Such an artifact may be used for analysis, i.e. for generating visual feedback representing the structure of actions and their dependencies between each other. I believe that programming languages researchers, especially with recent advents in mechanized reasoning, are uniquely positioned to contribute reasoning tools for programmatic media.

Artists want to use code in their art, and communities of such artists have developed their own languages and environments (such as Processing (`http://processing.org/`) and Twine (`http://twinery.org/`) to fill voids where no tools existed. Programming languages researchers can help attend to these needs with our well-established techniques for building high-level representations and development environments.

References

[1] Cervesato, I., and Scedrov, A. Relating state-based and process-based concurrency through linear logic (full-version). *Information and Computation 207*, 10 (2009), 1044–1077.

[2] Kakas, A., and Miller, R. A simple declarative language for describing narratives with actions. *Journal of Logic Programming 31* (1997), 157–200.

[3] Martens, C. Thesis proposal: Logical interactive programming for narrative worlds, 2013.

[4] Propp, V. *Morphology of the Folktale*. University of Texas Press, 1968.

[5] Schack-Nielsen, A., and Schürmann, C. Celf — a logical framework for deductive and concurrent systems (system description). In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'08)* (2008), Springer LNCS 5195, pp. 320–326.

[7] `http://www.doc.ic.ac.uk/~mpsha/ECExplained.pdf`