Rule-Based Interactive Fiction (Full Presentation)

Chris R. Martens Zachary A. Sparks Claire E. Alvis William E. Byrd
Carnegie Mellon University, Indiana University
{cmartens}@cs.cmu.edu,{zasparks,calvis,webyrd}@cs.indiana.edu

1. Interactive Fiction

Interactive fiction (hereafter IF) is a genre of game in which players interact in a text loop not unlike a REPL [7]. The game provides a prompt, the player enters a command, and the game responds with the next prompt, possibly changing some internal state.

Typically, the game's state includes a map of a physical space that the player navigates with directional commands. The space consists of connected rooms, and within the rooms are objects and characters that the player can interact with, e.g. by *looking* around, *examining* something, *taking* something, or *talking* to someone. The player also typically has an inventory to store taken objects, which can be used in various ways.

The author of a piece of interactive fiction is tasked with describing a rich setting and anticipating the player's actions. With each new interactable object introduced, the space of game play explodes combinatorially. Of course, in order for the game to feel interactive, we cannot anticipate and script a response to every action; we need to set up simple rules that *emergently generate* game content. (In the game design world, this notion of generativity is referred to as *procedural.*) Most frameworks for writing interactive fiction deal with this combinatorial explosion by establishing broad defaults for every command; the IF author needs only to override default game responses to selected actions upon a new object she introduces.

A key challenge in language design for any kind of game programming is enabling such a rich space of possible games that feel generative and interactive rather than canned. IF in particular is a ripe domain for programming languages research because it introduces the richness of game design without the extra cruft of rendering. Everything is a turn-based, discrete state transition. We can imagine turning this crank on an extralinguistic interpreter; the language, then, need only be for *specifying game logic.* A programming language for interactive fiction could easily extend to prototyping games with fancier rendering systems, and indeed some like-minded game programmers have done this [11].

The majority of IF has been developed in the systems Inform [8], TADS, and ADRIFT. The authors of this work are primarily familiar with Inform, specifically Inform7. In Inform7, the author specifies game behavior by matching on a player action (possibly guarded by some condition) and specifying a state change. We hope to base our design on this idea but using concepts from logic programming and substructural logics to make games easier to specify and reason about.

What follows is a sketch of ideas for describing pieces of interactive fiction as sets of rules and some speculation towards using such descriptions as a programming language.

2. Logic Programming and IF

In functional programming, we like to think of propositions as types and proofs as programs. In what the authors coin the Miller Correspondence [6], logic *programs* correspond to propositions and *traces* correspond to proofs. If we want to write a game as an *interactive logic program*, then running that program (playing that game) should somehow correspond to *interactive proof search*, or interactive theorem proving, not terribly unlike what many PL researchers do regularly in systems like Coq, Isabelle, and Agda. Important questions to ask in ITP come up in IF as well: at what point should reasoning be filled in automatically? What proof strategy should that automatic reasoning employ? In the talk we will argue that the proof theoretic ideas of *forward chaining* and *left focus* play a key role in the IF setting.

A logic program is a set of initial facts (typically general and hypothetical) from which new facts can be generated and queries on the space of facts can be made. It can be viewed as an inference system and its execution as proof search.

As a small example, consider the following Prolog [2] program to find paths between nodes in a graph:

```
path X X.
path X Z :- edge X Y, path Y Z.
```

Each block of code preceding a period is a *clause* of the program, and the capital-letter variables are implicitly universally quantified. The logical interpretation of this program is one in which there are atomic predicates `path(-,-)` and `edge(-,-)`, and two axiomatic propositions,

$$\forall x.\mathtt{path}(x, x)$$

and

$$\forall x, y, z.\mathtt{edge}(x, y) \land \mathtt{path}(y, z) \supset \mathtt{path}(x, z)$$

We can create a graph by adding an additional clause for each edge (with lowercase identifiers for specific nodes), then run a query such as `path a X` to get the set of all the nodes reachable from a. The execution of that program resembles constructing derivations of facts like `path a b` from rules in the program along with its core rules for implication and quantification.

We can imagine beginning to design a world with the basic logic programming toolset: we would introduce predicates (or types) for each kind of thing in the world, like rooms and objects and people, and we could write predicates like `visible` to determine whether the player can see an object. But as soon as we want to talk about state change, standard logic programming fails us. For example, if we want to turn off a light, the visibility of certain objects goes away—but treating the action as an ordinary implication, the old facts stay around. Ordinary propositional logics (such as the basis of Prolog) are *monotonic*: learning new facts can never erase old ones. This brings us to linear logic.

3. Linear Logic Programming

Linear logic [4] lets us reason locally about state. Hypotheses must be used exactly once, though they can be reordered. For example, using $\otimes$ as linear conjunction and $\multimap$ as linear implication, the proposition $A \otimes B \multimap B \otimes A$ is derivable, but neither $A \otimes B \multimap A$ nor $A \multimap A \otimes A$ is derivable.

A minimal example of an interactable object encoded as a linear logic program is toggling a switch:

```
flip on off.
flip off on.
toggle : sw v * flip v u -o sw u.
```

Implicitly, we have two contexts: δ, the ephemeral context, is where we put resources that can be consumed, whereas Γ, the persistent context, is where we put facts such as the rules defined above.

Supposing a distinguished proposition `end`, a standard linear sequent calculus for DILL (linear logic with !), and a desire to toggle the switch from on to off, we could construct the following derivation corresponding to a trace of the program:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma; sw\ on \longrightarrow sw\ on} \qquad \dfrac{}{\Gamma; \cdot \longrightarrow flip\ on\ off}}{\Gamma; sw\ on \longrightarrow (sw\ on) \otimes (flip\ on\ off)} \qquad \dfrac{\vdots}{\Gamma; sw\ off \longrightarrow end}}{\Gamma; \Delta, sw\ on, (sw\ on) \otimes (flip\ on\ off) \multimap sw\ off \longrightarrow end}}{\dfrac{\Gamma; \Delta, sw\ on, \forall(v,u).((sw\ v) \otimes (flip\ v\ u) \multimap sw\ u) \longrightarrow end}{\Gamma; \Delta, sw\ on \longrightarrow end}}$$

The unfinished derivation is the continuation of the program after toggling the switch. It represents a new choice point in the program where the human part of the interactive theorem prover could select a persistent rule from Γ. Supposing we had many switches, levers, and knobs, this component of choice could be a lot like a gameplay interaction.

3.1 Known issues

An important aspect of this problem arises when you want to add *persistently learnable facts* over ephemeral facts. In the previous section, we mentioned the possibility of having a *visibility* predicate, which is clearly ephemeral but which we may wish to use as a precondition to a rule without consuming it. In the talk we may go into further detail.

4. Defeasible Logic

Another weakness of vanilla logic programming is its inability to easily reason about exceptions to rules. A simple example is when determining whether or not a room is well-lit. By default, a room is lit, unless the lights are out, unless the player has a flashlight, unless the flashlight broke, unless the player fixed it...the list goes on and on.

Some of these are subsumed by linear logic, but linear logic still does not provide a general account of writing rules and their exceptions. For this, we need a different logic, called defeasible logic [9, 2], which innately allows reasoning about exceptions to rules.

For example, suppose we have a simpler case of the above example, in which a room is lit by default, unless the lights are out, unless the player has a flashlight. We could write this in d-Prolog as follows:

```
RD : {isRoom(X)} => lit(X).
RL : dark(X) => ~lit(X).
RF : {inRoom(X), has(flashlight)} => lit(X).

RF > RL.
RL > RD.
```

Each of the three rules indicates one of these cases. In addition to the rules themselves, we define an ordering on the rules stating their precedence. Intuitively, the lights being out in a room override their default illuminated status, and having a flashlight should illuminate the room even if the lights are out. Defeasible logic is a clean and elegant way of specifying rules such as these (and more complicated ones). It remains to be seen whether or not it would be subsumed by linear logic, but either way, this is a use case that we want to understand more thoroughly.

Acknowledgments

We thank Andrew “Zarf” Plotkin for his original presentation on rule-based interactive fiction, for his participation in the Iron Geek Language Design experiment, and for providing IF examples for testing our language design. We also thank Chung-chieh Shan for introducing us to both Andrew and this research problem, and for suggesting the use of defeasible logic in interactive fiction. Donald Nute has graciously answered our questions about defeasible logic and d-Prolog. We thank Cassie Orr and Doug Orleans for their enthusiastic participation in the Iron Geek Language Design marathon.

References

[1] Apt, K. R. *Principles of Constraint Programming.* Cambridge University Press, 2003.

[2] Covington, M. A., Nute, D., and Vellino, A. *Prolog programming in depth.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[3] Friedman, D. P., Byrd, W. E., and Kiselyov, O. *The Reasoned Schemer.* The MIT Press, 2005.

[4] Girard, J.-Y. Linear logic. *Theor. Comput. Sci. 50* (1987), 1–102.

[5] Marriott, K., and Stuckey, P. J. *Programming with Constraints. An Introduction.* The MIT Press, 1998.

[6] Miller, D. Proof search foundations for logic programming. `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/wollic03.pdf`, July 2003.

[7] Montfort, N. *Twisty Little Passages: An Approach to Interactive Fiction.* MIT Press, Cambridge, MA, USA, 2004.

[8] Nelson, G. Natural language, semantic analysis and interactive fiction. In *IF Theory.* Apr. 2005.

[9] Nute, D. *Defeasible logic.* Oxford University Press, Inc., New York, NY, USA, 1994, pp. 353–395.

[10] Plotkin, A. Rule-based programming. `http://eblong.com/zarf/rule-language.html`, June 2010.

[11] Smith, A. M. You have to mine the ore. `http://eis-blog.ucsc.edu/2009/06/you-have-to-mine-the-ore/`, June 2009.