Thesis Proposal: Logical Interactive Programming for Narrative Worlds

Chris Martens

November 11, 2013

Abstract

We present a logical theory of specifying interactive and reactive systems, particularly as they pertain to world-building and narrative construction.

Systems for executing, animating, and verifying sets of rules have been used ad nauseum for programming languages, which are typically expected to receive all input (the program) at once and execute monolithically. *Games* that humans play are also systems of rules, also amenable to treatment in these systems—yet rarely are they considered more than in scaled-down form as pedagogical examples. The richness of these rulesets emerges when they are seen as dynamic systems that evolve with the *choices* that agents make. These agents can be human or programmatic. This research endeavors to bridge the gap from existing logical frameworks to the ability to specify *dynamic, narrative worlds*.

Some game design scholars use the term *ludonarrative dissonance* (*ludo* being the Latin prefix for "play", thus used in game-related neologisms) to refer to the frequent disconnect between game mechanics and story intent—e.g. a game ostensibly "about" friendship and trust may offer interaction only in the form of combat and movement, with character and plot development relegated to (noninteractive) cutscenes. This work attempts to understand the structure of both *story* and *game mechanic* in the same framework, thus offering a basis for deeper ludonarrative harmony.

Thesis Statement: Phase-structured linear logic programming can form the basis of a framework for specifying, testing, and inventing ludonarrative mechanics.

Contents

1 Introduction

The state of the art in programming interactive media is more accessible than it has ever been: no longer does one need to learn sophisticated low-level programming skills to create games and share them widely. This is thanks in part to the development of tools such as Twine[1], GameMaker[2], and Unity[3], designed specifically for game designers and authors with little programming experience. It is also thanks to a return of contemporary fascination to *interactive fiction*, i.e. games made out of text, which require less asset-construction overhead than graphical games. A blurry line emerges between *branching story* and *text-based game*, the main difference being the way the authors and players think about what they are doing: interactive storytelling leans on narratology and literary theory, whereas games with text interfaces, such as "roguelikes" and "text adventures," have more focus on creating a dense space of interactions through techniques such as nonlinear navigation of the setting (spatial movement) and composable actions that are not situationally constrained (e.g. "take" has an action, possibly failure, on any object in any part of the game, rather than presenting itself as a choice only when relevant).

Meanwhile, an orthogonal line of development has been pursuing the use of *logical frameworks* for the sake of expressing and proving theorems about deductive systems (including programming languages, logical formalisms, and computational models). One such system, based on a logic programming interpretation of the LF type theory [13], is called Twelf [19]. Twelf has been used extensively in the field of programming languages research. [14, 11, 3, 6] More recently, the use of *substructural logic* as the basis for encoding concurrent, stateful, and distributed systems has motivated the development of CLF [24] and its implementation Celf. [21]

Celf's use of substructural logic, specifically *linear* logic, brings it into similar territory to *planning systems*, which have been used in the field of Artificial Intelligence (AI) for decades. [17, 18] Games research takes a lot of its foundations from AI studies, and interactive storytelling in particular has been used as a motivating application domain for planning. [20, 25]

Thus, the first step of this project is making the transitive connection between linear logic programming and interactive or branching narratives. The representation scheme discussed by Bosser et. al. [4] suggests that linear logic forms a suitable alternative basis to planning for interactive storytelling, and our preliminary mechanization results [16] suggest that the Celf framework in particular makes sense as a starting point for a logic-based approach to interactive narrative construction.

Encoding narrative branching and choice in linear logic basically presents the interactor with a finite enumeration of decisions at predetermined points. While this model can still capture the important structural property of *narrative causality*, it fails to offer the rich form of interactivity that a more generative "language of actions" found in many games can provide. In a text adventure that receives typed commands, for instance, we may always try to *take*

[1] `http://www.gimcrackd.com/etc/src/`
[2] `http://www.yoyogames.com/studio`
[3] `http://unity3d.com/`

or *eat* or *open* any object in the game, even if that is not a significant narrative action – many designers contend that this capability helps them create more convincing settings and that the player feels more agency.[4] Conversely, this makes more work for the game programmer, who needs a way to systematically account for and produce responses to each possible action from a combinatorially large set.

To create suitably rich and interactable worlds in this sense, we need to reconsider the architecture of interactive systems. The object-oriented programming community invented a good starting point for thinking about the design of such systems as having three interoperating components: the *model*, *view*, and *controller* (MVC). [5] These can be considered processes with distinct jobs that hand distinct kinds of data to one another in a loop (sometimes referred to as the *event loop*).

I will explain these components (as I consider the terms useful) through two examples: a calculator program and a turn-based 2D tile game.

The **m**odel is the internal logic of a process. In the calculator program, this is the functionality of arithmetic operations. In the game example, this is the processing of player/NPC actions and the consequent modification of the internal representation of the playing field.

The **v**iew is the outward-facing rendering of the result computed within the model. In the calculator example, this might just be a value printed at the console or in a text field. In the game example, this is the redrawing of the tilemap to reflect the updated positions of interactable objects.

The **c**ontroller is the set of choices available to the external agent (in our examples, a human user/player). For the calculator, this is the buttons for keying in numeric values and arithmetic operations. For the game, this would be any movement mechanism (arrow keys, mouse, joystick, dpad) and other game controls for interacting with tiles. Crucially, the controller typically includes "event handling" logic, for which the OO setting employs callbacks that fire whenever a particular event is triggered. Callbacks make for program structures that are notoriously difficult to reason about due to unpredictable transfers of control.

What I aim to do in this thesis is present a concise, declarative, proof-theoretic understanding of MVC-style interactivity that allows for to an extent sufficient for encoding, experimenting with, and inventing game mechanics. I aim to place most of the developer's focus on the *model*, or the rules of the game, while also accounting for descriptions of the view and controller, which could be the same for many different games. The same game (model) could even make sense with multiple different views or controllers, such as an interactive fiction piece with both textual and graphical rendering mechanisms. The idea is just that these should be separable and composable parts.

It is not my aim to create an end-to-end games production system, but rather to architect a *ludonarrative laboratory* in which authors and game designers can rapidly put their ideas into executable form so as to better understand them, debug them, and iterate their designs.

[4] A detailed discussion of the axes of variation in interactive fiction by one of its prominent authors can be found at `http://eblong.com/zarf/essays/ifdef.html`

[5] http://heim.ifi.uio.no/ trygver/themes/mvc/mvc-index.html

1.1 Logical Interactivity

The motivation for this work is *creative fluency*—I want to enable creators to run their ideas shortly after coming up with them without limiting the *language* of ideas to existing tropes and clichés. There is nothing inherent to (my notion of) games about the concept of an *enemy* or a *platform*, for example, so these should not be built-in constructs to the language (though certainly they should be representable). Seemingly, the current approach to making game-writing languages accessible involves limiting the ludolinguistics in this way: Twine says the linguistic constructs are "passages" and "links", letting you create webs of text; GameMaker says they are rooms containing objects with any of a finite list of pre-defined behaviors; PuzzleScript[6] says they are transformations of 2-dimensional adjacency relationships in a grid; Unity says they are surfaces with realistic physical properties. Beyond these thin veneers, to develop anything that feels, in terms of game mechanics, like more than just a collage of re-skinned instantiations of default components, these "programming-free" tools release you to the wilds of C#, CSS, Python, and JavaScript. Given that most people seem attracted to these tools *because of* their advertised scope being limited to game programming, as opposed to general-purpose programming languages, I feel we can better deliver on this promise.

Thus I position the language in my proposal as something of a delicate balance, and hopefully not a contradiction, between "general-purpose" (in the sense of non-genre-specific) and "domain-specific" (in the sense of aiming specifically at the domain of game design).

Why should we consider games from a logic-based point of view? I take a broad and inclusive definition of "game" in the interest of not inhibiting what is still a young and growing artform, and as a matter of personal opinion, I find much of the most compelling modern game design work to be exactly that of which critics question "gameness." But there is a common strand through all things I've seen described as games, which is that they are defined by *rules*, and if there is a task that logic is unquestionably useful for, it's describing rules. A computer may allow you to write down rules and check that they are syntactically well-formed, logically consistent, and perhaps that they allow or do not allow some particular emergent behavior—just as logical frameworks have been used for programming languages research. In this sense, a game can be understood as a kind of language—perhaps a language of actions and interactions rather than a language of programming. For instance, consider the following rule of Tag.

$$\frac{It \text{ tags you}}{\text{You become } It}$$

Read as "if-then" relationship between what's above the line and what's below it, and combined with the initial condition that one person is named It, the game has an invariant that only one person is It during the entire run of the game. This behavior can be observed empirically by *running* the rules of the game or by *describing the invariant* in logic and asking the computer to prove for you that it holds. This is the sort of thing we should be able to do

[6] http://www.puzzlescript.net/

by explaining the rules of Tag to a computer in as direct a manner as we would describe them to a five-year-old.

1.2 Project Outline

In the remaining sections of my proposal, I will describe (1) the extent to which we can already describe narratives games with linear logic programming; (2) the limitations of this approach; (3) a proposal for a language that addresses these limitations; (4) additional work required to support my thesis statement; and (5) a plan to carry out that work. Here is a quick summary of each of these parts:

- **Preliminary results:** With Anne-Gwen Bosser and João Ferreira, I have encoded branching narratives in Celf and shown how to use the logic programming engine to generate several distinct stories from the same "story world" or narrative setup. We have also developed an analysis tool to understand the causality structure of narrative events. Additionally, I have worked through several examples (some failures) of representing even less linear structure, such as games with combat rounds and two-dimensional movement.
- **Limitations:** The primary limitations of the existing logical framework are (1) an inability to control the order of rules firing and (2) an inability to group sets of rules together by their functionality. (1) means that we can execute randomly generated "playthroughs" of games, but we cannot meaningfully insert a point in execution at which human-controlled player input is accepted. (2) means that our programs are difficult to reason about, because every rule we write must be considered in interaction with every other rule of the game, even when it is in a separate "level" or "screen" or other phase of play.
- **Language proposal:** These limitations motivate the introduction of *phase* as a programming construct. Like a vertical stripe down a musical score or a collection of available actions during a player's turn in a board game, a phase is a possibly-repeated section of code that may be given meaning in isolation, and that meaning may change what occurs after it. We sketch a languge design for linear logic programming with phases and describe its semantics.
- **Current issues:** I need to resolve the semantics of my user-facing language and prove it sound, although I confidently conjecture that there is *a* sound semantics with respect to CLF's implementation. I also need to ensure that my current proposal for the statement of invariants and phase behaviors is simple enough to be mechanically checked but expressive enough to capture relevant examples. The main obstacle between current results and a justification of my thesis statement is that I have yet to establish a productive feedback loop between the artifacts creatable with this language and the refinement of a game's design.

- **Plan for proposed work:** To address the above issues, I will continue working on the theoretical components of the language. I will then need to put in substantial implementation effort (of the language prototype, its proof checking mechanisms, and its visualization/analysis tooling) and construct several examples using that implementation. I will need to make a thorough qualitative, if not quantitative, comparison of my prototype against other tools discussed in this overview, and perhaps sketch how I would implement my framework as a scripting language for such a tool.

Again, my thesis statement is that *phase-structured linear logic programming can form the basis of a framework for specifying, testing, and inventing ludonarrative mechanics.* I will revisit this statement once more at the end of the proposal, once I have made precise what I mean by phase-structured linear logic programming. At that point, I can be more precise about what I mean by specifying, testing, and inventing ludonarrative mechanics.

2 Background

Imagine specifying the rules for movement on a grid controlled by up, down, left, and right arrows.

If the player presses the arrow key for direction D and the cell in direction D is empty, move to that cell.

We can think of this in slighly more formal notation as a pictorial logical inference rule (or collection thereof, one for each movement direction):

```
                                             [   ]             [ @ ]
[>]  [ @ |  ]       [<]   [   | @ ]    [^]   [ @ ]       [v]   [   ]
---------------     ----------------  ---------------  ----------------
  [  | @ ]               [ @ |  ]            [ @ ]             [   ]
                                             [   ]             [ @ ]

Legend:
 @                   player
 [>], [<], [^], [v]  key right, left, up, down
 [ X | Y ]           X is adjacent to Y
```

If we use mathematical predicates to express location and adjacency, we can write this in slightly more compact, abstract notation:

$$\frac{\text{key}(\text{arrow}(D)) \quad \text{player-location}(L) \quad \text{in-direction}(D, L, L') \quad \text{empty}(L')}{\text{player-location}(L') \qquad \text{empty}(L)}$$

The meaning of these rules is, informally, just that if whatever is above the horizontal line holds, we change it to whatever is below it. When implementing this syntax as a programming language, we need to be more precise about what we mean by "holds" and "changes," and these ideas turn out to have a basis in logical principles that can help us better understand them.

The actual programs I will show in my examples look very similar to this if-then notation; for example:

```
key (arrow D) * player_location L * in_direction D L L' * empty L'
  -o {player_location L' * empty L}.
```

In the rest of this section I will describe how to write and run a program using replacement rules, describe a justification of these rules based on *linear logic*, and finally use that justification to motivate the more sophisticated language constructs in Celf.

2.1 Programming With Replacement Rules

To start out, I define a *logic program* as a collection of *replacement rules*, where replacement rules have the form

$$\frac{A_1 \dots A_n}{B_1 \dots B_m}\ r$$

The symbols A_i and B_i stand for *logical predicates* and the symbol r stands for the name of the rule.

Let us consider as a first example a simulation of the Blocks World scenario commonly used as an introductory AI problem. The setup is that we have distinguishable blocks that can be set on a table or on another block. We also have a robotic arm that can pick up the top block from any stack, or if it is holding a block, set it down on top of the top block of any stack.

First, we choose predicates to represent the components of this scenario's state.

- $\mathrm{arm\text{-}free}$: Denotes when the arm is free to pick up a block.
- $\mathrm{arm\text{-}holding}(X)$: Denotes that the arm is holding the block X.
- $\mathrm{on\text{-}table}(X)$: Denotes that the block X is on the table.
- $\mathrm{on}(X, Y)$: Denotes that the block X is stacked atop block Y.
- $\mathrm{clear}(X)$: Denotes that the block X doesn't have any blocks atop it and so may be picked up.

Using these predicates, we can describe a scenario like the following, with the robot arm free and three blocks, one on the table and the other two stacked.

```
  _|_
 |   |


    [c]
[a] [b]
```

We can represent this logically as the following *state*, or collection of predicates:

$$\{\text{arm-free}, \text{ontable}(a), \text{clear}(a), \text{ontable}(b), \text{on}(c, b), \text{clear}(c)\}$$

Sometimes we will abbreviate this state with Δ. Now we need to describe the rules for such a world. We will need four rules: two each for picking up and putting down, one for when the target is on the table and another for when it rests instead on another block.

$$\frac{\text{on}(X,Y) \qquad \text{clear}(X) \qquad \text{arm-free}}{\text{clear}(Y) \qquad \text{arm-holding}(X)}\ \text{pickup-b}$$

$$\frac{\text{on-table}(X) \qquad \text{clear}(X) \qquad \text{arm-free}}{\text{arm-holding}(X)}\ \text{pickup-t}$$

$$\frac{\text{arm-holding}(X) \qquad \text{clear}(X)}{\text{on}(X,Y) \qquad \text{clear}(X) \qquad \text{arm-free}}\ \text{putdown-b}$$

$$\frac{\text{arm-holding}(X)}{\text{on-table}(X) \qquad \text{clear}(X) \qquad \text{arm-free}}\ \text{putdown-t}$$

We have thought about the meaning of each individual rule, but we should pause for a moment to consider the meaning of this program taken as a whole. In particular, if you are used to programming in other languages, you may be accustomed to specifying a sequential order in which pieces of code execute. Here, we have not explicitly told the computer anything about ordering: we have only said, “whenever this rule applies, apply it.”

What does it mean for a rule to *apply*? The rule’s premise (part above the line) only describes the part of the state we wish to change. The entire program state might consist of many more parts, which we leave untouched.

For example, if our blocks world state looks like this:

```
 _|_
|   |

    [c]
[a] [b]
```

then the pickup-t rule applies, which would change the state to look like this:

```
 _|_
|[a]|

    [c]
    [b]
```

The rule need not mention the blocks `b` and `c`, because they stay the same.
In terms of the logical representation, we've rewritten our state

$$\{\text{arm-free}, \text{ontable}(a), \text{clear}(a), \text{ontable}(b), \text{on}(c, b), \text{clear}(c)\}$$

to

$$\{\text{arm-holding}(a), \text{ontable}(b), \text{on}(c, b), \text{clear}(c)\}$$

This suggests something about the *algebraic structure* of states. Specifically, we can say:

- Order doesn't matter: if a rule applies to a state Δ, A, B, Δ', it applies to a state Δ, B, A, Δ'.
- Multiple copies of a predicate mean something different from a single copy, i.e. it is not the case that $\Delta, A, A = \Delta, A$
- When we fire a rule replacing predicates A with B on a state Δ, A, the resulting state is Δ, B.

Logically, this notion of state can be identified with a *context* in linear logic, each of those properties embodied as a logical principle. [12] Algebraically this structure is a *commutative monoid*. Set-theoretically, it is a multiset.

We can now describe when a rule r replacing A with B applies in each of those terms:

- Logic: the rule applies to state Δ if $\Delta = \Delta', \Delta_A$ and $\Delta_A \vdash A$.
- Algebra: if Δ taken as a monoidal structure with operator $*$ is equivalent to $\Delta' * A$, then the rule applies.
- Set theory: if Δ is treated as a multiset and $A \subseteq \Delta$.

For the final reason in that list, sometimes this manner of program execution is called *multiset rewriting*.

Notice that this definition of when a rule can apply does not rule out the possibility that *multiple rules may apply*—in fact, in the example given above, the arm may also pick up the block c. Which rule it will actually pick depends on the semantics of the inference engine. Celf picks nondeterministically among all rules that apply, which means that different runs of the program may do different things! I will revisit the nondeterminism in these programs at numerous points throughout the document. In some scenarios it serves as a handy expressive tool for making arbitrary decisions, and it is compatible with alternate modes of computation for which nondeterminism is undesireable.

Finally, before moving on, I will introduce a bit more new notation. While our current syntax for rules is fairly clear, as we introduce more advanced constructs, it will begin to fail us. Also sometimes it is convenient to have a less vertical notation for rules.

Because what we have described coincides with linear logic, I will borrow linear logic notation to write them in a more compact form. Specifically, I will conjoin multiple premises and conclusions with the $\otimes$ symbol, and I will write the horizontal line between them as $\multimap$, such that a rule

$$\frac{A_1 \ldots A_n}{B_1 \ldots B_m}\ r$$

can be alternately written

$$r : A_1 \otimes \ldots \otimes A_n \multimap B_1 \otimes \ldots \otimes B_n$$

2.2 Initial States

We've almost completed our blocks world simulation program, but to actually run this program, we need to give it an initial state. Suppose we want to start with the arm free and three blocks (a, b, and c), each independently on the table and clear.

Can this initial state be specified as a rule in the program? We might think it could be introduced as a rule with no premises:

$$\frac{}{\text{arm-free} \quad \text{on-table}(a) \quad \text{clear}(a) \quad \text{on-table}(b) \quad \text{clear}(b) \quad \text{on-table}(c) \quad \text{clear}(c)}\ \text{init}$$

But what the absence of any premises would mean according to our interpretation is *replace nothing with the stuff below the line*. What does it mean to replace nothing? When can we expect the rule to apply?

The answer is *always*, meaning that this is not the program we wanted to write. To see why, again consider the semantics of replacement rules in a state Δ. By writing nothing above the line in the rule, we mean "if 'nothing' is part of our state, the rule can fire." But "nothing" is a part of any state! Algebraically speaking, the null resource is the *unit*: it's exactly that which when combined with Δ yields Δ (or more symbollically, if 1 is the unit, $\Delta * 1 = \Delta$). So if we have a rule allowing us to replace the null resource with A, we're actually allowing arbitrary copies of A to be added to Δ.

To get around this, we can specify in our programming language that there is an init atom supplied by the inference engine at the beginning of execution, such that we could give the rule as

$$\frac{\text{init}}{\text{arm-free} \quad \text{on-table}(a) \quad \text{clear}(a) \quad \text{on-table}(b) \quad \text{clear}(b) \quad \text{on-table}(c) \quad \text{clear}(c)}\ \text{init-rule}$$

After this rule fires, the special init atom will be consumed, so the rule may not fire again.

2.3 Persistence

By now we have seen a need for three logical connectives: $\otimes$, $\multimap$, and 1. We have discussed what a rule using 1 to the left of a $\multimap$ does: $1 \multimap A$ creates arbitrarily many copies of A.

For completeness' sake, we should consider the meaning of $A \multimap 1$. Such a rule can be read "replace A with nothing," and that's exactly what it does: a program with such a rule would delete copies of A that occurred in the state.

Let's reconsider our grid movement example from previously and see if these connectives are general enough. Written using linear logic notation, our movement rule was:

$$\begin{aligned}\mathrm{key}(\mathrm{arrow}(D)) \otimes \ \mathrm{player\text{-}location}(L) \otimes \ \mathrm{in\text{-}direction}(D, L, L') \otimes \ \mathrm{empty}(L') \\ \multimap \mathrm{player\text{-}location}(L') \otimes \ \mathrm{empty}(L)\end{aligned}$$

But if we interpret this rule as we have been interpreting the blocks world rules, the $\mathrm{in\text{-}direction}$ predicate disappears after its application. Instead, we meant for it to be a constant, immutable fact about grid cells.

Thus we introduce the logical operator ! for refering to constant, immutable facts. We can write the rule instead

$$\begin{aligned}\mathrm{key}(\mathrm{arrow}(D)) \otimes \ \mathrm{player\text{-}location}(L) \otimes \ !\mathrm{in\text{-}direction}(D, L, L') \otimes \ \mathrm{empty}(L') \\ \multimap \mathrm{player\text{-}location}(L') \otimes \ \mathrm{empty}(L)\end{aligned}$$

This is called a *persistent* use of the $\mathrm{in\text{-}direction}$ resource. When we mark something persistent, we can be sure that it will never go away.

We might define it as follows:

$$\frac{}{\mathrm{in\text{-}direction}(\mathrm{north}, \mathrm{cell}(X, Y), \mathrm{cell}(X, Y+1))}\ n \qquad \frac{}{\mathrm{in\text{-}direction}(\mathrm{south}, \mathrm{cell}(X, Y+1), \mathrm{cell}(X, Y))}\ s$$

$$\frac{}{\mathrm{in\text{-}direction}(\mathrm{east}, \mathrm{cell}(X, Y), \mathrm{cell}(X+1, Y))}\ e \qquad \frac{}{\mathrm{in\text{-}direction}(\mathrm{west}, \mathrm{cell}(X+1, Y), \mathrm{cell}(X, Y))}\ w$$

Although it is not necessary, we can simplify our presentation by saying that a predicate is always written with ! or without, and that predicates written with ! are not generated unless needed. This means that we can write each of the above rules as

$$1 \multimap !\mathrm{in\text{-}direction}(\dots)$$

Because of the unusual meaning of $A \multimap !B$, which in fact coincides more with traditional logic programming's *implication* than with the replacement semantics with have given $\multimap$, in general we will write such formulae $A \to B$. In particular when $A = 1$, we will simply write B. Such a "rule" (thought we might call it a "fact") just means that B may always be inferred when needed. As such, the above four rules will simply appear in our program as:

$$\begin{aligned} n &: \mathrm{in\text{-}direction}(\mathrm{north}, \mathrm{cell}(X, Y), \mathrm{cell}(X, Y+1)) \\ s &: \mathrm{in\text{-}direction}(\mathrm{south}, \mathrm{cell}(X, Y+1), \mathrm{cell}(X, Y)) \\ e &: \mathrm{in\text{-}direction}(\mathrm{east}, \mathrm{cell}(X, Y), \mathrm{cell}(X+1, Y)) \\ w &: \mathrm{in\text{-}direction}(\mathrm{west}, \mathrm{cell}(X+1, Y), \mathrm{cell}(X, Y)) \end{aligned}$$

2.4 Backward Chaining and Monadic Notation

While the primary different between the in-direction predicate and the other predicates we have seen so far is that it is *persistent* rather than linear, there is another difference.

This difference lies in the reading of the rules. The current presentation assumes a *replacement* semantics for the rules, meaning we read them from top to bottom. We could instead, at least for the persistent predicates, read them with *inference* semantics. Inference semantics assumes that there is a particular goal formula G in mind, and we may look at rules of the form $A_1 \otimes \ldots \otimes A_n \multimap G$ in order to proceed. Applying such a rule then means that we recursively try to infer each of $A_1, \ldots, A_n$ as goals. This reading makes sense for a predicate like in-direction: we do not want to eagerly apply the rules that generate in-direction facts; we simply want to consult those facts in the premises of rules like move.

What I have termed *replacement semantics* and *inference semantics* have more standard terms, *forward chaining* and *backward chaining*. While I have taken forward chaining to be primary, most studies of logic programming actually take backward chaining as primary.

Because Celf incorporates both forward- and backward-chaining into its semantics, it also introduces a syntactic separation between the two notions. For technical reasons it takes the *unmarked* rules to be backward-chaining, and forward chaining rules are written using the notation

$$A \multimap \{B\}$$

So, in fact, almost all of the rules we have written so far would appear in Celf's notation with this marker around the predicates to the right of the $\multimap$.

The notation $\{A\}$ represents a *monad* algebraically, or a *lax proof* of A logically. For this reason we will call it "the monad" in Celf. Full elaboration on this terminology may be found in previously published material. [15]

By this point we have in our inventory the connectives $\multimap$, $\otimes$, 1, !, and $\{\}$.

2.5 Representing Data Structures

To program more complex examples, we may wish to have access to standard data types like numbers and lists, as well as operators over them. Fortunately these are encodable with backward-chaining, persistent predicates using standard logic programming techniques.

The notation we use is a subset of LF. [13] We can use it to declare new types and new inhabitants for those types, e.g.

```
nat : type.
z : nat.
s : nat -> nat.
```

This *signature*, or collection of rules, represents the natural numbers, which have a constructor z for zero and s for the successor. A term `s (s (s z))` can be written in this notation to represent the number 3 (there are three iterations of s) and be said to have type `nat`.

We also use the type keyword to declare predicates. For example, addition on natural numbers can be described as a three-place predicate, given the following type declaration and inhabitants. The inhabitants "define" addition by pattern-matching on the shape of the numbers (as given by the inhabitants of nat) and using induction (the second rule defining plus has an appeal to plus as a premise).

```
plus : nat -> nat -> nat -> type.
plus/z : plus z N N.
plus/s : plus N M P -> plus (s N) M P
```

Our predicates need not be functions: we can also give a two-place predicate classifying a less-than relationship between numbers.

```
lt : nat -> nat -> type.
lt/z : lt z N.
lt/s : lt N M -> lt (s N) (s M)
```

In some cases rules will be more readable if we write them "backwards," e.g.

```
lt/s : lt (s N) (s M)
     <- lt N M.
```

This way of writing it is equivalent to what we wrote previously and makes each line start with the pattern-match, making it clearer which case of the program it defines.

We can also code up lists (though we lack polymorphism, so we have to specify a particular element type):

```
list : type.
nil : list.
cons : nat -> list -> list.
```

Here is a predicate for appending two lists:

```
append : list -> list -> list -> type.
append/nil : append nil L L.
append/cons : append (cons N L1) L2 (cons N L)
            <- append L1 L2 L.
```

While this sort of logic programming is auxilliary to the main points I wish to make in this proposal, it is important that the reader be fluent in them. For a reader unfamiliar with "conventional" logic programming, I recommend working through a Prolog tutorial. [7]

Using these data structures, we can now write linear logic programs over them. A good first example is list permutation, which takes advantage of the nondeterministic committed choice semantics we specified for replacement rules. We will take a list as input, turn each of its elements into a linear predicate, then use another rule to reassemble a list out of these predicates.

This code uses the predicate explode to inductively decompose the list into a member for each of its members:

[7] e.g. http://www.lix.polytechnique.fr/ liberti/public/computing/prog/prolog/prolog-tutorial.html

```
explode/cons : explode (cons N L) -o {member N * explode L}.
explode/nil : explode nil -o {collect nil}.
```

This rule plucks a `member` out of the context along with the currently collected list and conses the member to the front:

```
add : member N * collect L -o {collect (cons N L)}.
```

2.6 Running from Start to Finish

We can specify a test list for input using a term definition:

```
ltest : list = (cons z (cons n1 (cons n2 (cons n3 (cons n4 nil))))).
```

Now the question is how to observe the effects of introducing `explode ltest` into the initial state.

In Celf we have two ways of doing this. Since our program is defined with forward-chaining rules, we can `trace` the initial state:

```
#trace * {explode ltest}.
```

When we load the file, this incantation prints out an iteration of the state after every rule that fires, for example

```
-- explode (cons !z !(cons !(s !z) !(cons !(s !(s !z)) !(cons !(s !(s !(s !z)))
           !(cons !(s !(s !(s !(s !z)))) !nil))))) lin
-- member z lin,
   explode (cons !(s !z) !(cons !(s !(s !z)) !(cons !(s !(s !(s !z)))
           !(cons !(s !(s !(s !(s !z)))) !nil)))) lin
-- member z lin, member (s !z) lin,
   explode (cons !(s !(s !z)) !(cons !(s !(s !(s !z)))
           !(cons !(s !(s !(s !(s !z)))) !nil))) lin
-- member z lin, member (s !z) lin, member (s !(s !z)) lin,
   explode (cons !(s !(s !(s !z))) !(cons !(s !(s !(s !(s !z)))) !nil)) lin
-- member z lin, member (s !z) lin, member (s !(s !z)) lin,
   member (s !(s !(s !z))) lin, explode (cons !(s !(s !(s !(s !z)))) !nil) lin
-- member z lin, member (s !z) lin, member (s !(s !z)) lin,
   member (s !(s !(s !z))) lin, member (s !(s !(s !(s !z)))) lin,
   explode nil lin
-- member z lin, member (s !z) lin, member (s !(s !z)) lin,
   member (s !(s !(s !z))) lin, member (s !(s !(s !(s !z)))) lin,
   collect nil lin
-- member (s !z) lin, member (s !(s !z)) lin, member (s !(s !(s !z))) lin,
   member (s !(s !(s !(s !z)))) lin, collect (cons !z !nil) lin
-- member (s !z) lin, member (s !(s !z)) lin, member (s !(s !(s !(s !z)))) lin,
   collect (cons !(s !(s !(s !z))) !(cons !z !nil)) lin
-- member (s !z) lin, member (s !(s !(s !(s !z)))) lin,
   collect (cons !(s !(s !z)) !(cons !(s !(s !(s !z))) !(cons !z !nil))) lin
-- member (s !(s !(s !(s !z)))) lin,
```

```
   collect (cons !(s !z) !(cons !(s !(s !z)) !(cons !(s !(s !(s !z)))
            !(cons !z !nil)))) lin
-- collect (cons !(s !(s !(s !(s !z)))) !(cons !(s !z) !(cons !(s !(s !z))
            !(cons !(s !(s !(s !z))) !(cons !z !nil))))) lin
```

We can also run a *query*, which treats its argument as a *goal* for backward chaining. As far as we are concerned so far, this means that if we give it the goal $A \multimap \{B\}$, it will create an initial state containing A, run forward-chaining rules until no more can fire, and then attempt to prove B from the resulting state. It will fail if it cannot do so.

For various reasons we will explore in the rest of the document, this particular query mechanism is not exactly what we want for most of our examples. But we can use it to inspect the permutation program just fine:

```
#query * * * 5 explode ltest -o {collect L}.
```

If we ignore the *s for now, this particular incantation asks for five repetitions of the query seeded with `explode ltest` and expecting a context of the form `{collect L}` at the end. The query output should tell us what value gets filled in for `L`.

Indeed, in the five iterations we observe several different values for the result list `L`:

```
Iteration 1
 #L = cons !(s !(s !(s !(s !z)))) !(cons !z !(cons !(s !(s !z)) !(cons !(s !z)
      !(cons !(s !(s !(s !z))) !nil))))

Iteration 2
 #L = cons !z !(cons !(s !(s !(s !z))) !(cons !(s !z) !(cons !(s !(s !z))
      !(cons !(s !(s !(s !(s !z)))) !nil))))

Iteration 3
 #L = cons !(s !(s !z)) !(cons !(s !(s !(s !(s !z)))) !(cons !(s !(s !(s !z)))
      !(cons !(s !z) !(cons !z !nil))))

Iteration 4
 #L = cons !(s !z) !(cons !(s !(s !(s !(s !z)))) !(cons !(s !(s !z)) !(cons !z
      !(cons !(s !(s !(s !z))) !nil))))

Iteration 5
 #L = cons !(s !(s !(s !(s !z)))) !(cons !(s !(s !(s !z))) !(cons !z !(cons !(s
      !(s !z)) !(cons !(s !z) !nil))))
```

Interestingly, though, that is not all the output we get. We also get something like a trace in more compact notation for each iteration. Let's narrow our focus to just one iteration:

```
Iteration 1
Solution: \X1. {
    let {[X2, X3]} = explode/cons X1 in
    let {[X4, X5]} = explode/cons X3 in
    let {[X6, X7]} = explode/cons X5 in
    let {[X8, X9]} = explode/cons X7 in
```

```
    let {[X10, X11]} = explode/cons X9 in
    let {X12} = explode/nil X11 in
    let {X13} = add [X8, X12] in
    let {X14} = add [X4, X13] in
    let {X15} = add [X6, X14] in
    let {X16} = add [X2, X15] in
    let {X17} = add [X10, X16] in X17}
 #L = cons !(s !(s !(s !(s !z)))) !(cons !z !(cons !(s !(s !z))
      !(cons !(s !z) !(cons !(s !(s !(s !z))) !nil))))
```

Reading down the right side of the =, we notice that we see the replacement rules in the order that they are applied. Furthermore, these rules are applied to variables of the form `Xn` (written to the right of the rule name) and bind new variables of the same form (written to the left of the = sign). These variables correspond to the resources to the left and right of the $\multimap$ in the rule, and they demonstrate a resource dependency in the computation. There is a lot to be said about this notion of dependency, but for now let it suffice to think of a query's output as just a record of the inference engine's work.

2.7 Linearity vs Affinity

So far, what we have observed about the linear function space $\multimap$ is that it treats resources as having *multiplicity*: the combination of resources $a \otimes a$ is stronger than just a single copy of a. What may not be apparent is that query's semantics also requires that we match *exactly* the goal described, i.e. we must not have any extra resources in the state for the goal to succeed. For example, if our goal were a but we had $a \otimes a$ in our state, the goal would not succeed.

For the purpose of describing situations where we would like it not to matter whether a resource exists in the state or not for a query to succeed, CLF incorporates the connective $@A$. The function is similar to $!A$, but rather than being an unrestricted use, it describes an A that may be used 0 or 1 times.

To see an example of how affinity can change the behavior of a program, consider our list permutation program. Suppose we wanted to change it to return a list containing a subset of the elements in the input list. We would need to change our rules for `collect` so that we can stop early with whatever list we've currently collected:

```
add : member N * collect L -o {collect (cons N L)}.
stop : collect L -o {return L}.
```

We will also need to query `return` rather than `collect`. But if we run the query now, several queries will fail. Whichever `members` we did not select for inclusion in the return list still linger in the state, and if treated as linear resources, that means failure.

We need to change the rule

```
explode/cons : explode (cons N L) -o {member N * explode L}.
```

to an *affine* use of `member` i.e.

```
explode/cons : explode (cons N L) -o {@member N * explode L}.
```

This means that we no longer require `member` predicates to be consumed. Now every query should succeed and produce output like the following.

```
Solution: \X1. {
    let {[@X2, X3]} = explode/cons X1 in
    let {[@X4, X5]} = explode/cons X3 in
    let {[@X6, X7]} = explode/cons X5 in
    let {[@X8, X9]} = explode/cons X7 in
    let {[@X10, X11]} = explode/cons X9 in
    let {X12} = explode/nil X11 in
    let {X13} = add [X2, X12] in
    let {X14} = add [X6, X13] in
    let {X15} = stop X14 in X15}
 #L = cons !(s !(s !z)) !(cons !z !nil)
```

2.8 Linear Backward Chaining

While linear forward chaining and persistent backward chaining make a well-behaved pair of computation modes, we will also make use of *linear backward chaining*.

At this point we will have to be much more honest about the roots in logic we have been drawing from all along. The logic program's execution is effectively *proof search in a sequent calculus*. A sequent calculus is a deductive system where we take a *sequent*

$$A_1 \dots A_n \vdash C$$

and attempt to prove it using rules in the calculus. Some of these rules are given by the definition of the logical connectives, for example

$$\frac{\Delta, A, B \vdash C}{\Delta, A \otimes B \vdash C}$$

which, if read bottom-up, decomposes a compound predicate $A \otimes B$ into two distinct components of the state A and B which can be used separately.

The collections of propositions to the left of the turnstile $\vdash$ correspond to states in the logic program, and in logic they are sometimes called *contexts*.

To account for this mode of computation we need to rewrite our *replacement rules*

$$\frac{A}{B}$$

as *inference* rules

$$\frac{\Delta, B \vdash C}{\Delta, A \vdash C}$$

where C is the overall goal of the program. (Incidentally, this formalism does not account well for programs that do not have overall goals.)

Consider the example of converting change (dimes, nickels, and quarters) into equivalent amounts of currency. One terminating strategy would be to use the fewest coins. We can represent that strategy as a forward-chaining program:

```
ddn-q : d * d * n -o {q}.
d-nn  : n * n -o {d}.
```

But if we were to run the query

```
d * n * n * d -o {q * d}
```

we might not get the right answer. If the program chooses the `d-nn` rule, it will commit to that choice and wind up with three dimes which cannot be converted. We need backward chaining's *backtracking* semantics to ensure that this query succeeds.

To convert the above program into a backward-chaining one we need to distinguish a "goal" atom. All rules will be written in terms of this atom.

The `goal` atom's base case is the actual goal of our query:

```
qd : goal o- q * d.
```

Now we can rewrite the rule in "paper notation" from

$$\frac{\mathrm{d} \quad \mathrm{d} \quad \mathrm{n}}{\mathrm{q}}$$

to

$$\frac{\mathrm{q} \vdash \mathrm{goal}}{\mathrm{d}, \mathrm{d}, \mathrm{n} \vdash \mathrm{goal}}$$

In concrete syntax this becomes

```
ddn-q : goal
        o- d * d * n
        o- (q -o goal).

nn-d : goal
        o- n * n
        o- (d -o goal).
```

In other words, we can internalize the $\Delta \vdash A$ judgment as $\bigotimes \Delta \multimap A$.

2.9 Choice

With the introduction of sequent notation $\Delta \vdash A$, there is one more logical connective it will be useful to discuss: the choice operator $A \& B$. The rules for decomposing such a resource in the state are

$$\frac{\Delta, A \vdash C}{\Delta, A \& B \vdash C} \qquad \frac{\Delta, B \vdash C}{\Delta, A \& B \vdash C}$$

In fact we can think of these rules with replacement semantics as well:

$$\frac{A\&B}{A} \qquad \frac{A\&B}{B}$$

In other words, if we have $A\&B$ we get to choose between two rules, one yielding A and one yielding B. We also need to account for what happens if $A\&B$ appears as the goal of the sequent:

$$\frac{\Delta \vdash A \quad \Delta \vdash B}{\Delta \vdash A\&B}$$

This rule says that A and B must be constructible from the *same* set of resources Δ in order to be constructible from Δ.

2.10 Notation Summary

To summarize, we have now accounted for the following notation:

- $A \multimap \{B\}$ is a *replacement* rule, used during forward chaining, whereas $A \multimap B$ without the monadic marker is an *inference* rule, used during backward chaining.
- $A \otimes B$ is the conjunction of two resources A and B which we have available simultaneously.
- 1 is the empty resource, and the unit of $\otimes$.
- $!A$ is a *persistent* fact that we may appeal to as a condition for the rule applying (without consuming it) or generate in the conclusion of a rule to introduce something permanent.
- $@A$ is an *affine* predicate that need not be consumed by the end of the program but may not be used more than once.
- $A\&B$ is a *choice* between resource A and resource B. We can construct $A\&B$ from Δ if we can construct A from Δ and B from Δ, and we can *use* $A\&B$ by selecting one of the two resources.

Again, each of these connectives has a formal logical justification, and the reader may refer to formal descriptions of the system [5, 24, 15, 22] for a full explanation.

3 Encoding narratives in Celf

Linear logic has recently been proposed as a suitable representation model for narratives [4]: its resource-sensitive nature allows natural reasoning about narrative actions and the changes they cause to the environment. Under a logic programming interpretation, we can animate these representations to generate new stories (and story variations) from the circumstances of a single, static narrative. We take this idea as a first step toward fully interactive systems. [16]

Celf's nondeterministic implementation combined with its proof term construction mechanism makes it especially suitable for *generating and analyzing* stories. The proof terms can be interpreted as causally structured narrative plots. To improve narrative analysis, we developed a prototype front-end which provides statistics and graphical representations of proof terms.

3.1 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 a existing, linear, baseline story to support our experiments. The detailed formalization of such a story will bring to existence the many decision points and opportunities for actions to succeed, fail or be deferred. Identifying the circumstances within a static story such as Madame Bovary [10] is a human activity that can be assisted by companion works [9].

The narrative elements we need to 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. **Narrative actions** are transforming events occurring in the narrative. We model the impact they have on the narrative, in terms of narrative resource creation and consumption.

Narrative actions conceptually have a lot in common with the replacement rules seen in section 2. However, recall that while those express the consumption and production of resources, the rules themselves are not consumed when used: the meaning we gave them (implicitly) said we could use a rule arbitrarily many times.

In the context of narratives, we may wish to have some rules describing the *story world* or setting that can be used arbitrarily often. However, for our purposes we want to define a *narrative action* as a *rule* that gets consumed when used. A rule-shaped proposition $A \multimap \{B\}$ is just another resource in CLF, so the framework supports this mode of use.

For this reason, instead of writing narrative actions as declared rules $a : A \multimap \{B\}$, we will give *type abbreviations* $a : type = A \multimap \{B\}$. This way, we can specify our initial state by bundling together all of our actions, i.e. writing $\bigotimes a_i$ to represent the bundle of actions $\bigotimes(A_i \multimap \{B_i\})$.

Consider the narrative action of Emma marrying Charles which will allow us to illustrate how we model resources. This action requires the presence of Emma and Charles, and some facts representing those characters' motivation for this action: Emma's grace and escapism. Their marriage results in the new fact that they are married and Emma gets bored. We write it in Celf as:

```
emmaMarriesCharles : type
  = emma * escapism * grace * charles
```

```
    -o {emmaIsBored * @emma *!emmaCharlesMarried}.
```

Immutable facts and hard rules are modelled as persistent: this is how we represent Emma and Charles married status. We declare the state representing Emma's boredom as linear, since one of the driving force for her actions in the story is to escape boredom. The resources corresponding to Emma and Charles are respectively declared in the initial environment as affine and persistent, because Emma may die in the story and Charles is a constant presence in the modelled fragment (line 38 of Figure 2). This is why the resource corresponding to Emma needs to be preserved by this narrative action. Note that because in this specific code example we are searching for stories where Emma dies, we could have used a linear resource as well.

In addition to the author's notes [9] 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 learns about her father's death from Homais (returning late from a date with Leon) because Charles is afraid to upset her. She learns about inheritance. We first model:

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

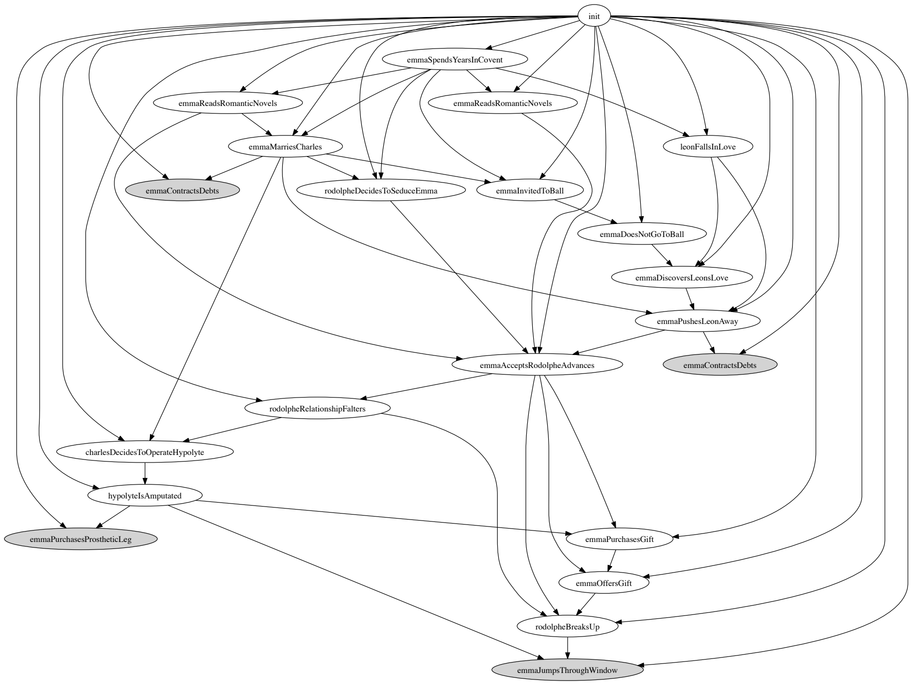
During the same event, a side conversation occurs between two of the characters present during which Emma incidentally learns where to find arsenic. The importance of this knowledge becomes only apparent when we model the narrative action corresponding to Emma's death. We then modify the code:

```
emmaLearnsBovaryFatherDeath : type
  = emma * leonEmmaTogether * charlesIsConcerned * homais
    -o {@arsenic * @inheritance * @leonEmmaTogether * @emma}.
emmaCommitsSuicide  : type
  = emma * ruin * arsenic * emmaRebels -o {@emmaIsDead}.
```

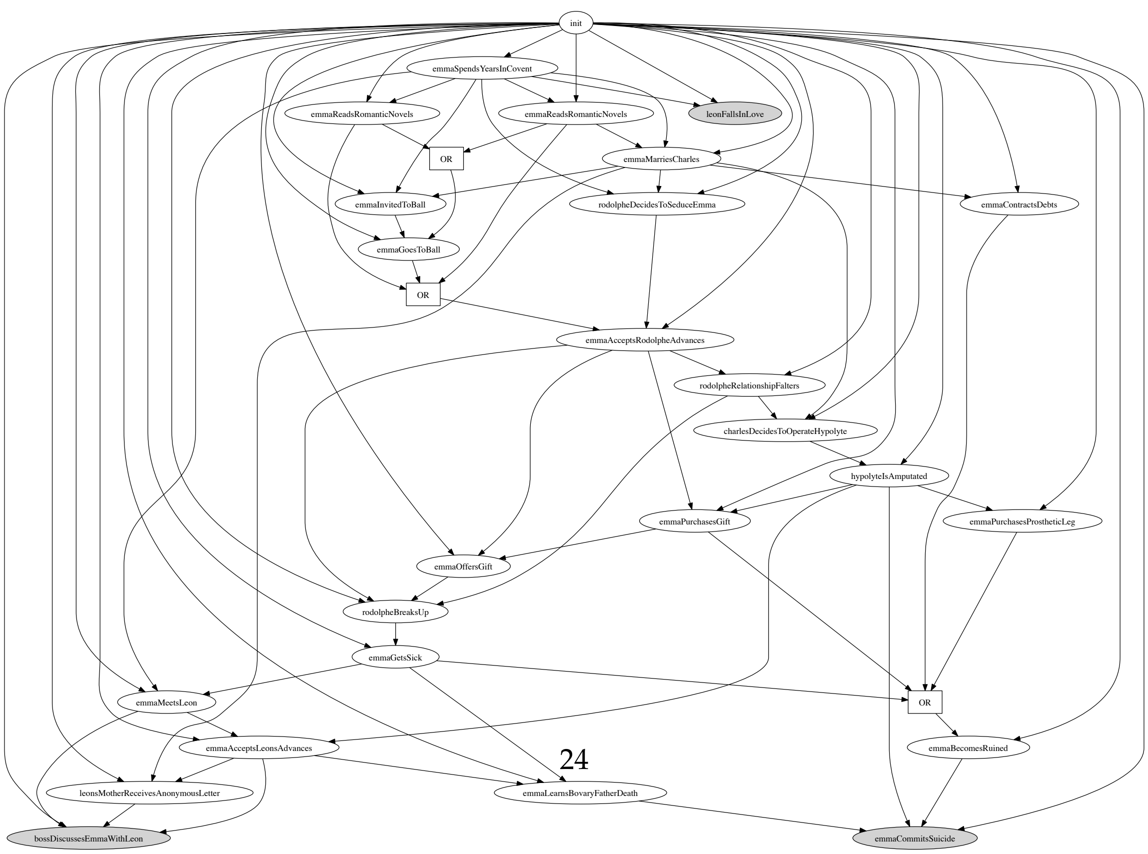
Mutually exclusive narrative actions can be explicitly suggested using the choice connective `&` in the declaration of the initial conditions. These can be used to encode key turning points in the narrative that are broadly recognized as such, which is frequently the case when using existing stories as a baseline. We use this connective to model Emma's choice to attend the ball (see line 44 in Figure 2). The use of the choice connective `&` causes variation in the outputs. However, the main mechanism for varied outputs remains the competition for the consumption of resources by different narrative actions.

3.2 Incremental Formalization of the Narrative

As the examples above demonstrate, an advantage of modelling narratives using a programming language is the ability to iteratively fine tune the model. Indeed, programming is an



(a) Emma does not attend the Vicomte's Ball but still wants to escape her life. She defenestrates when left by Rodolphe.



(b) As in the original story, Emma attends the Vicomte's Ball. Following Rodolphe's departure she becomes sick but later start another liaison with Leon. After being ruined, she ingests Arsenic.

iterative activity alternating between coding and testing phases, and the tool that we developed improves immensely the effectiveness of the testing phase.

Testing can exhibit plots with specific characteristics, as illustrated in Figure 3. We can also verify if the generation has a varied output (differing significantly from the original plot) by exhibiting corresponding plots such as in Figure 1. One can also test the impact of more *narrative drive* on the generation if enforcing an action is desired: by making emmaAcceptsLeonsAdvances linear, one can observe the effect on the number and variability of the stories generated (such a modification would generate a smaller number of stories per 100 queries, and all of them would end by Emma's death by poisoning).

Such fine-tuning can help setting up threshold values where levels or a certain amount of a given resource is needed to trigger various specific narrative actions.

3.3 Generated Plots

The complete code corresponding to the extract shown in Figure 2 consists of a total of 105 lines of code, including 31 narrative action descriptions (the rest of the code being mainly atomic declarations). As we have only explicitly encoded one branching choice, the variety of outputs is due to the narrative actions semantics (narrative action's competition for resources, and the fact that different actions can create the resource consumed by another one), and to the forward chaining variability.

The code described allows to generate 72 different narrative sequences for 100 attempts. After a comparison of the corresponding plots using the *CelfToGraph*'s command `stats`, 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 1 exhibits examples of story variants among those generated, which have been exhibited by the tool: the first tells a story where Emma jumps through the window following the departure of Rodolphe, and the other a story where she takes arsenic. If we look at the code Figure 2 l.31 32, two narrative actions `emmaJumpsThroughWindow` and `emmaGetsSick` 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 are different plots (i.e., 226 true story variants).

3.4 Interacting with narratives

Once we add the ability to transfer control to a human user to select a narrative action from among the actions that may fire, we are not too far off from the idea of a game. In the next section I'll describe how to interpret a branching narrative encoding as a "choose your own path" *interactive* narrative. This corresponds to the conception of interactive fiction known as "hypertext" because it can be rendered as a display of text with clickable links (a hypertext node) where clicking on the link renders a new node. I will then explain how to get to the

```
%% Encoding of an extract of the novel: Madame Bovary.
emma : type.
charles : type.
homais : type.
leon : type.
rodolphe : type.
emmaCharlesMarried : type.
convent : type.

<............>

emmaIsDespaired : type.
charlesIsConcerned : type.
emmaInLove : type.
leonEmmaTogether : type.
arsenic : type.
inheritance : type.
denounced : type.
ruin : type.
emmaIsDead : type.

emmaSpendsYearsInConvent : type = emma * convent -o {!novels * !grace * !education * @emma}.
emmaReadsRomanticNovels : type = emma * novels -o {@escapism * @escapism * @emma}.
emmaMarriesCharles : type
  = emma * escapism * grace * charles -o {emmaIsBored * @emma *!emmaCharlesMarried}.
emmaInvitedToBall : type = emma * emmaCharlesMarried * grace -o {@ball * @emma}.
emmaGoesToBall : type
  = emma * ball * escapism -o {@escapism * @escapism * @escapism * @escapism * @emma}.
emmaDoesNotGoToBall : type = emma * ball -o {emmaIsBored * @emma}.

<.....>

emmaJumpsThroughWindow : type = emma * emmaIsDespaired * emmaRebels -o {@emmaIsDead}.
emmaGetsSick : type
  = emma * emmaIsDespaired -o {@debt * @debt * @debt * @debt * !charlesIsConcerned * @emma}.
emmaLearnsBovaryFatherDeath : type
  = emma * leonEmmaTogether * charlesIsConcerned * homais
    -o {@arsenic * @inheritance * @leonEmmaTogether * @emma}.
emmasLoveForLeonFalters : type
  = emma * leonEmmaTogether * emmaInLove -o {@emmaIsBored * @emma * @leonEmmaTogether}.
emmaContractsDebts : type = emma * emmaIsBored -o {@debt * @emma}.
emmaCommitsSuicide : type = emma * ruin * arsenic * emmaRebels -o {@emmaIsDead}.
init : type =
 { convent * @emma * @leonIsBored * !charles * !rodolphePastLoveLife * !homais }.

#query * * * 100 (init -o {emmaIsDead}).
```

Figure 2: Celf Code excerpt of a narrative description inspired by a fragment of the novel *Madame Bovary* [10]. Atomic types corresponding to narrative resources are followed by types describing narrative actions, then the initial environment declaration and finally by the query of 100 attempts to generate stories ending by Emma's death. (The complete file contains 105 lines of code.)

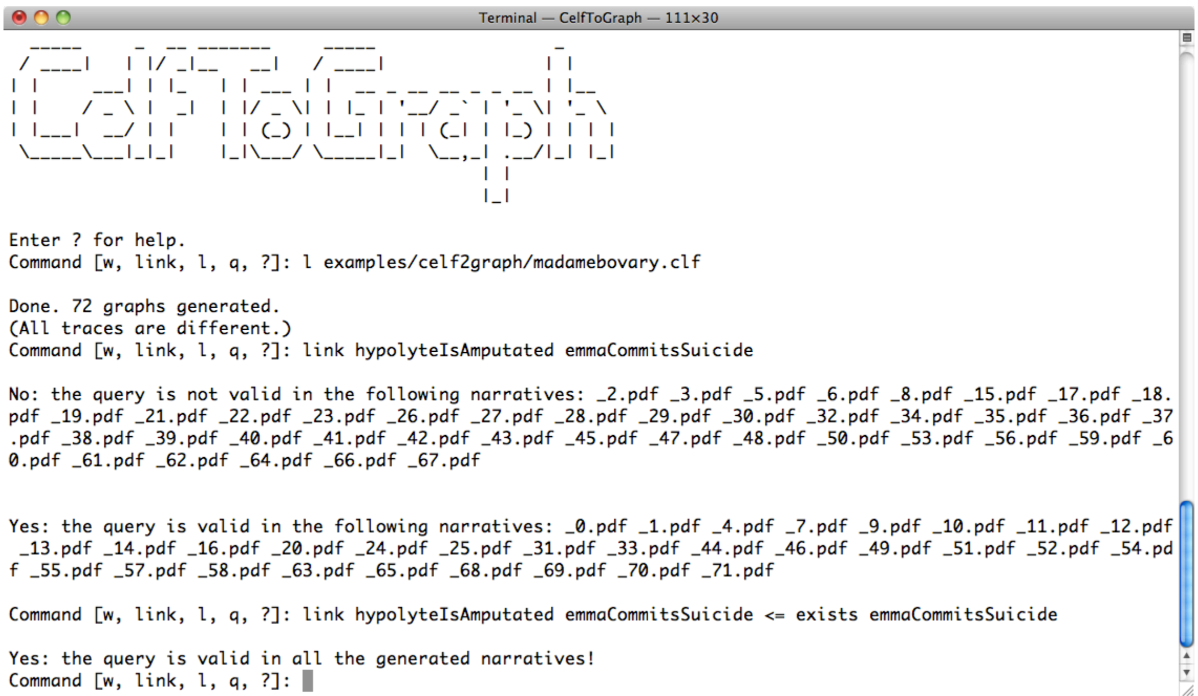


Figure 3: Loading a Celf file and querying the set of generated plots. The first query discriminates stories depending on a causal path between Emma's suicide by Arsenic and Hypolyte's amputation. The second story restricts the search of the same causal link to stories where Emma takes Arsenic, evidencing Hypolyte's amputation is then one of the cause of Emma's death.

other common conception of interactive fiction, *parser* IF, from the hypertext conception.

4 Encoding game mechanics in Celf

Previously, we saw how rules written in linear logic can represent *narrative actions* taken by characters in a story to drive the plot forwards. We can represent game mechanics in a similar way, where the rules in the signature offer an *interface* for an interactor to make choices.

We start by progressing from a *branching* story, as described above, to an *interactive* story.

4.1 Interaction Model

Linear logic gives us a way of expressing choice with the & operator. A forward-chaining linear logic program also expresses choice through the nondeterminism of rule selection. So far, we have been thinking of the "chooser" as the inference engine. One way to introduce human choice would be to simply prompt for a choice at each of these points.

Consider the simple "choose-your-own-path"-style story below.

```
Page 1
------
You reach a familiar house. You could inspect the wine cellar or enter
through the front to the den. To go to the cellar, turn to page 2. To go to
the den, turn to page 3.

Page 2
------
You are in the cellar. There's a small door here, which you can open by
turning to page 4. There's also a dusty set of stairs leading up to the
den, which you can do by turning to page 3.

Page 3
------
You are in the den. There's a trapdoor in the floor, leading down to the
cellar. To enter the cellar, turn to page 2.

Page 4
------
You come across a secret passage! To be continued...
```

This story consists of several labelled *passages*, or pages. the interactor makes a sequence of choices that the story author cannot predict, yet the story must progress in every case. A sequence of choices generates a single *static* story, as in traditional fiction.

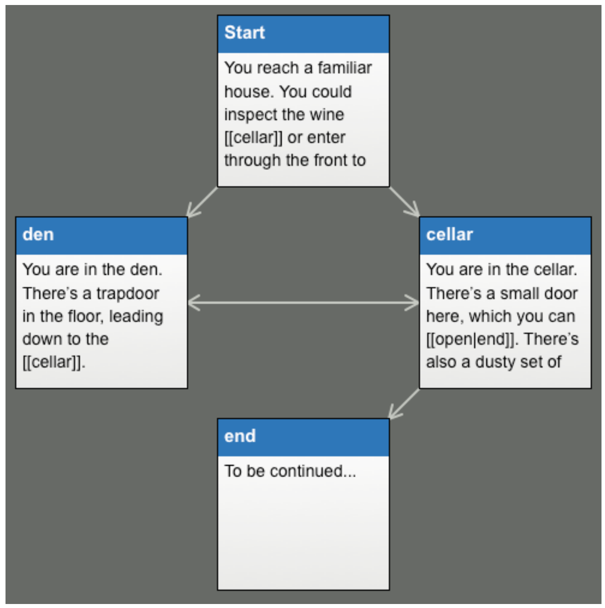
Here's the example encoded in Celf:

```
start_choices : start -o {den & cellar}.
den_choices : den -o {cellar}.
cellar_choices : cellar -o {den & (key -o opendoor)}.
open_door : opendoor -o {end}.

init : type = {start * key}.
```

As before, the key representational concept is *narrative resources*, which include story-relevant objects such as the key, but also serve to represent *situations*, such as being in a particular room of the house.

Tools such as Twine allow the author to compose these stories by directly manipulating the *passage graph*, so that we get a sort of state machine view of the story:



Note that the graph of possible choices need not be acyclic—an interactor can repeatedly go between the den and the cellar ad nauseum, according to the world's transition rules. Some presentations distinguish between *world rules* (dictating the reliable laws of "physics" in the fictional universe) and *narrative actions*, which may only be performed once—in Celf, this is a matter of putting rules in the signature (in which rules are treated persistently) versus including them as part of the initial state.

Next, we add a couple of pieces of mutable, global state—a key and a lamp that can be retrieved and used to access more of the game. In this setting, what is meant by "mutable state" is effectively a way of recording a past choice, whereas by default the choose-your-own-path model keeps no choice history. (One way that physical choose-your-own-path books implemented state was by asking the reader to keep track of values on a piece of paper.) The following transcript shows an example of interaction with such a story.

```
You are in a familiar house. You see doors to a wine cellar and to the den.
> enter cellar

You are in the cellar. There's a small door here. There's also a dusty set
of stairs leading up to the den.
> open door
It's locked.
```

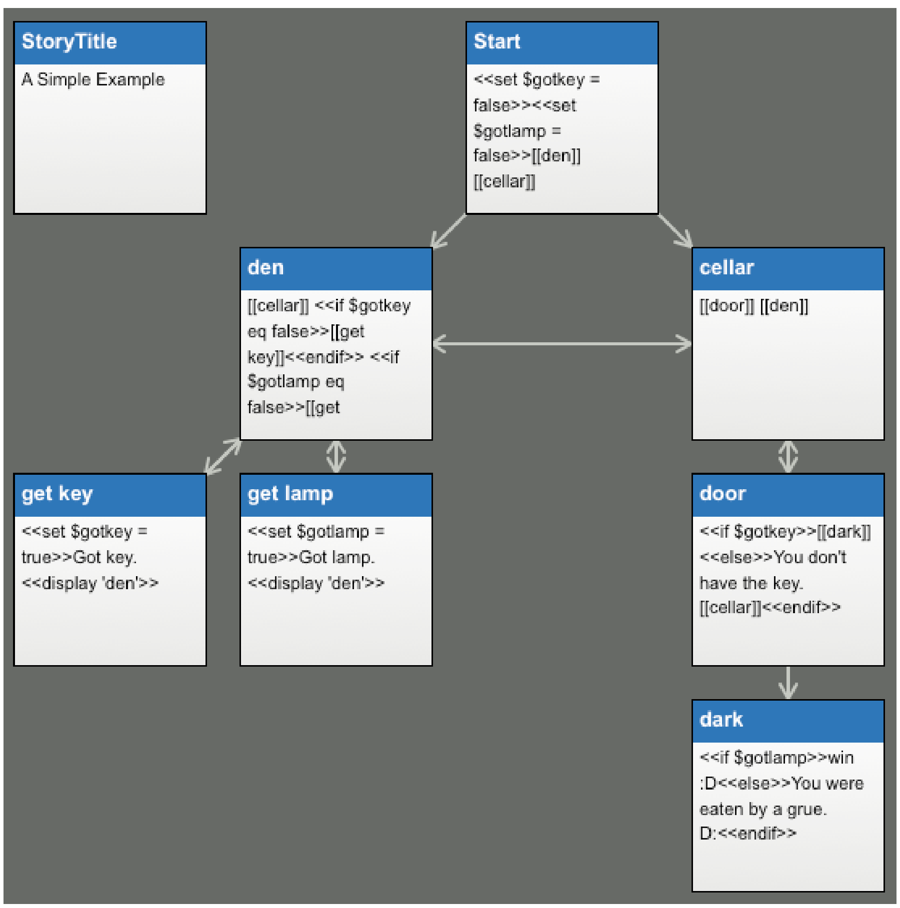
```
> go up

You are in the den. There's a trapdoor in the floor, leading down to the
cellar. A lamp and a key hang on hooks on the wall.
> get lamp
Taken.
> get key
Taken.

You are in the cellar. [...]
> open door

You find yourself in a secret passage. The way is lighted by your lamp.
> [...]
```

In Twine, to add this functionality requires the use of "macros," or calls to functions written in JavaScript appearing between certain delimiters in the passage text, as hinted at in the following screenshot:



Suddenly the visual structure offered by the passage graph is much less informative—it tells us nothing about the new control flow constraints we've introduced via the key and the lamp mechanisms.

We can write this story as a linear logic program by having some choices generate a linear resource inside the forward-chaining monad that cannot be consumed immediately:

```
%% A simple twine game with a few bits of state.
%% For full parseable Celf file, see twine-state.clf.

start_choices : start -o {den & cellar}.

den_choices : den -o
        {cellar
         & (nolamp -o {getlamp})
         & (nokey -o {getkey})}.
get_lamp : getlamp -o {gotlamp * den}.
get_key : getkey -o {gotkey * den}.

cellar_choices : cellar -o {den & opendoor}.
open_door_without_key : opendoor * nokey -o {cellar * nokey}.
open_door_with_key : opendoor * gotkey -o {dark}.

dark_without_lamp : dark * nolamp -o {lose}.
dark_with_lamp : dark * gotlamp -o {win}.

init : type = {nokey * nolamp * start}.
```

Here, another distinction between narrative actions and world rules becomes apparent: there are certain transitions that the player has no control over, such as opening the cellar door in darkness leading to being eaten by a grue.[8] Movement, picking up items, and opening things, are choices that, by Celf's default semantics, will nondeterministically fire, but we would like some way of instead offering that choice to a user.

Nonetheless, typing this up in Celf and running it lets us generate "random" play sequences and inspect them after the fact. Running a `#query` on `init -o {win}`, for example, asks for a proof that from the initial state we can get to a winning state, and Celf will nondeterministically explore the game space and return a proof if it successfully reaches `win`—just as it can explore the space of narratives within a narrative situation as in the previous section. An example of such a proof is:

```
Solution: \X1. {
    let {[X2, [X3, X4]]} = X1 in
    let {X5} = start_choices X4 in
    let {X6} = cellar_choices (X5 #2) in
    let {X7} = den_choices (X6 #1) in
    let {X8} = X7 #2 #1 X3 in
    let {[X9, X10]} = get_lamp X8 in
    let {X11} = den_choices X10 in
    let {X12} = cellar_choices (X11 #1) in
```

[8]A grue is a fictional monster invented for the Zork games; see http://zork.wikia.com/wiki/Grue

```
let {X13} = den_choices (X12 #1) in
let {X14} = X13 #2 #2 X2 in
let {[X15, X16]} = get_key X14 in
let {X17} = den_choices X16 in
let {X18} = cellar_choices (X17 #1) in
let {X19} = den_choices (X18 #1) in
let {X20} = cellar_choices (X19 #1) in
let {X21} = open_door_with_key [X20 #2, X15] in
let {X22} = dark_with_lamp [X21, X9] in X22}
```

The syntax `M #n` (e.g. `X5 #2`) indicates a selection of choice n from an $n+k$-ary choice $A_1 \& \ldots \& A_n \& \ldots \& A_{n+k}$. So since `X5` has the type `den & cellar`, the proof term `X5 #2` indicates taking the second branch, the cellar. This forms a suitable (type-correct) resource to subsequently apply `cellar_choices` to, as seen in the next line.

Incidentally, while in practice the $\&$ connective is a handy tool for explaining choice in terms of linear logic, we will typically just write a rule $A \multimap \{B \& C\}$ as two rules, $A \multimap \{B\}$ and $B \multimap \{C\}$. While not strictly operationally equivalent (they produce different proof terms), the two formulations are logically equivalent, and for the sake of a more uniform formalism in which all rules may be written with just the connectives $\otimes$, $\multimap$, and $\{-\}$, the remainder of the document will take the latter representation when convenient.

4.2 Example: text adventure

So far, all that we've done is notice that *simulating* an interaction with an interactive fiction matches the same pattern as generating narratives from a narrative situation. But the distinction between *situation* and *actor*—or between *player* and *game*—is still unclear: some rules seem to define game logic, whereas others define actions available to the player.

We now take the above example and delineate what we mean by an action, then prefix all narrative action rules by a fact that the "current action" corresponds to the rule under consideration:

```
% A simple branching story game with a few bits of state and an interaction
% model.

%  These are all the player controls, including motion.
action : type.
'start : action.
'opendoor : action.
'getlamp : action.
'getkey : action.
'starttoden : action.
'starttocellar : action.
'dentocellar : action.
'cellartodoor : action.
'cellartoden : action.

% current action
cur : nat -> type.
```

```
cur_act : action -> type.

% Transition rules. Every rule from the previous version has an additional
% cur_act premise, and outputs a "tick" whenever we want to return control
% to the player.

start_to_den : cur_act 'starttoden * start -o {den * tick}.
start_to_cellar : cur_act 'starttocellar * start -o {cellar * tick}.

den_to_cellar : cur_act 'dentocellar * den -o {cellar * tick}.
den_to_lamp : cur_act 'getlamp * den * nolamp -o {getlamp}.
den_to_key : cur_act 'getkey * den * nokey -o {getkey}.
get_lamp : getlamp -o {gotlamp * den * tick}.
get_key : getkey -o {gotkey * den * tick}.

cellar_to_den : cur_act 'cellartoden * cellar -o {den * tick}.
cellar_to_door : cur_act 'cellartodoor * cellar -o {opendoor}.

open_door_without_key : opendoor * nokey -o {cellar * nokey * tick}.
open_door_with_key : opendoor * gotkey -o {dark}.

dark_with_lamp : dark * gotlamp -o {win}.
dark_without_lamp : dark * nolamp -o {lose}.

init : type = {nokey * nolamp * start * cur z * tick}.

% Mechanism for simulating interaction
player_sim : tick -o {Pi a:action.cur_act a}.
```

In this version of the encoding, we can observe which rules correspond to input from the player: the ones which consume a `cur_act` token. Now the player is simulated by a process which generates these tokens, offering a clearer stratification.

The player simulation process makes use of another logical construct, $\Pi x : A.B$ (or `Pi x:A. B` in concrete syntax). In fact this construct appears implicitly in all of our other rules that use capital-letter, or *unification*, variables: they are implicitly universally quantified. In other words, a rule `foo(X) -o {bar(Y)}` is syntactic sugar for `Pi x. Pi y. foo x -o {bar y}`.

When the Π construct appears on the right-hand side of a rule, it plays a slightly different role: the inference engine must *come up with* a term to substitute for the bound variable when the rule is applied. This works conveniently for our example, wherein essentially we want the inference engine to play the role of the player and (nondeterministically) select an action. In the next example, we will look at an alternative means of specifying or simulating player input.

The `tick` token represents the transfer of control from the game back to the player. We have had to thread this atom through the entire control flow of the game, including carefully omitting it in the postfix of rules that trigger further game logic, but making sure every control flow branch of that game logic eventually emits a `tick`.

This programming pattern suggests a flaw in the language, and my project includes a proposal to solve it with the introduction of *phases*. A phase in this example is an indicator of *whose turn it is*—the interpreter's or the player's. Relatedly, we'd like to actually accept input from a human player's keyboard, not just simulate random input. These issues are addressed in the core of my proposal, the *phase* language described in Section 5.

4.3 Example: Sokoban

Sokoban[9] is a classic 2D tile puzzle game in which the player controls an entity that can move through empty spots on the grid and push blocks into empty spots. Usually, one or more empty spots is also marked as being a "target," and the object of the game is to cover all of the targets with blocks.

Sokoban is the initial pedagogical example for PuzzleScript,[10] a recently-made browser-based scripting language for turn-based, 2D tile games. PuzzleScript is very domain-specific for this purpose, and as such used a lot of specialized syntactic sugar for relationships between entities in 2D space. The encoding of Sokoban in PuzzleScript is the following single rule:

```
[ > Player | Crate ] -> [ > Player | > Crate ]
```

This rule can be read, "If the player tries to move right and there is a crate to the right of the player, then the player moves right and the crate moves right." In fact, the direction "right" symbolized by `>` and the left-to-right reading of English text is polymorphic over each of the cardinal directions.

This way of describing tranformations on 2D space in combination with player actions, then rendering the transformation rules as a game, can also be found in the StageCast Creator visual programming environment.[11]

My aim by describing Sokoban in Celf is to demonstrate how, with the tools already available to us in linear logic, and thus without sacrificing the simplicity and generality of our language, we can immitate this kind of straightforward representation with just a little more overhead than these tools (PuzzleScript and StageCast Creator) which are noted for their directness and ease of learning.

Since linear logic does not have any inherent notion of adjacency, our version of Sokoban cannot be quite as concise. We will code up adjacency as a three-place relation `in_dir` between a grid position, a direction (north, east, south, or west), and another position, where `in_dir(P,Dir,P')` means that `P'` is one hop away from `P` in direction `Dir`:

```
- : in_dir (posxy X Y) north (posxy X (s Y)).
- : in_dir (posxy X Y) east (posxy (s X) Y).
- : in_dir (posxy X (s Y)) south (posxy X Y).
- : in_dir (posxy (s X) Y) west (posxy X Y).
```

Now I can describe the preceding PuzzleScript rule in Celf's notation:

[9] http://en.wikipedia.org/wiki/Sokoban
[10] http://www.puzzlescript.net/documentation/rules101.html
[11] http://www.stagecast.com/

```
push :
loc pusher L * in_dir L Dir L' * loc block L'
             * in_dir L' Dir L'' * empty L''
  -o {empty L * loc pusher L' * loc block L''}.
```

We need one additional rule to allow unlimited movement (this comes for free in PuzzleScript):

```
move :
loc pusher L * in_dir L Dir L' * empty L'
  -o {empty L * loc pusher L'}.
```

As before, we can introduce an interactivity model by adding an extra atom at the beginning of the rules:

```
push :
cur_act (arrow Dir) *
loc pusher L * in_dir L Dir L' * loc block L'
             * in_dir L' Dir L'' * empty L''
  -o {empty L * loc pusher L' * loc block L''}.

move :
cur_act (arrow Dir) *
loc pusher L * in_dir L Dir L' * empty L'
  -o {empty L * loc pusher L'}.
```

The act of building these control points into our program forces us to consider the language of actions, i.e., the *controller* part of the user interface. In this case we choose to use arrow keys as our control interface. The language of actions available to the player is then given by those keys, or formally as

```
action : type.
arrow : dir -> action.
cur_act : action -> type.
```

The purpose of this example is to demonstrate another point in a wide space of *languages of action* that constitute interfaces to games. So far we have seen relatively constrained languages of action related to movement and finitary choice. My next example discusses *parser* interactive fiction as the basis for a more expressive and programmer-extensible language of action.

4.4 Parser Interactive Fiction

In this example we will describe distinct sets of nouns and verbs that may be combined arbitrarily accourding to our action language. Also, instead of a distinct type representing each passage location, we now have a type of passages and a predicate "location" to indicate which one we are in.

```
object : type.
key : object.
lamp : object.
door : object.

got : object -> type.
~got : object -> type. % complement.

% Passages
passage : type.
start : passage.
den : passage.
cellar : passage.
dark : passage.

location : passage -> type.

opendoor : type. % an internal game logic directive.
lose : type.
win : type.

% Externally available (controller) actions.
action : type.
'get : object -> action.
'open : object -> action.
'move : passage -> action.
```

Now “get” and “open” are actions that can apply to any object in the game, and “move” takes an argument which is the passage to move to.

The rules governing relevant game actions are as follows:

```
start_to_den : cur_act ('move den) * location start
                   -o {location den * tick}.
start_to_cellar : cur_act ('move cellar) * location start
                   -o {location cellar * tick}.

den_to_cellar : cur_act ('move cellar) * location den
                    -o {location cellar * tick}.
den_to_lamp : cur_act ('get lamp) * location den * ~got lamp
                  -o {got lamp * location den * tick}.
den_to_key : cur_act ('get key) * location den * ~got key
                 -o {got key * location den * tick}.

cellar_to_den : cur_act ('move den) * location cellar -o
                    {location den * tick}.
cellar_to_door : cur_act ('open door) * location cellar -o {opendoor}.

open_door_without_key : opendoor * ~got key
                            -o {location cellar * ~got key * tick}.
open_door_with_key : opendoor * got key
                            -o {location dark}.
```

```
dark_with_lamp : location dark * got lamp -o {win}.
dark_without_lamp : location dark * ~got lamp -o {lose}.
```

As before, we emit a `tick` whenever we want to transfer control back to the player. The only point where this doesn't happen immediately after processing a `cur_act` is the `'open door` command, which has two cases that we handle with separate rules.

Here is the rule for generating a new action upon receiving a `tick`:

```
next_act : tick * cur N * nth_act N A -o {cur_act A * cur (s N)}.
```

Finally we can specify a test case and an initial state:

```
% 'starttoden, 'getlamp, 'getkey, 'dentocellar, 'cellartodoor.
act0 : nth_act z ('move den).
act1 : nth_act (s z) ('get lamp).
act2 : nth_act (s (s z)) ('get key).
act3 : nth_act (s (s (s z))) ('move cellar).
act4 : nth_act (s (s (s (s z)))) ('open door).

init : type = {~got key * ~got lamp * location start * cur z * tick}.
```

And if we run a query, we get a trace output like the following:

```
Solution: \X1. {
    let {[X2, [X3, [X4, [X5, X6]]]]} = X1 in
    let {[X7, X8]} = next_act [X6, [X5, act0]] in
    let {[X9, X10]} = start_to_den [X7, X4] in
    let {[X11, X12]} = next_act [X10, [X8, act1]] in
    let {[X13, [X14, X15]]} = den_to_lamp [X11, [X9, X3]] in
    let {[X16, X17]} = next_act [X15, [X12, act2]] in
    let {[X18, [X19, X20]]} = den_to_key [X16, [X14, X2]] in
    let {[X21, X22]} = next_act [X20, [X17, act3]] in
    let {[X23, X24]} = den_to_cellar [X21, X19] in
    let {[X25, X26]} = next_act [X24, [X22, act4]] in
    let {X27} = cellar_to_door [X25, X23] in
    let {X28} = open_door_with_key [X27, X18] in
    let {X29} = dark_with_lamp [X28, X13] in
    let {X30} = report_win [X29, X26] in X30}
```

We could also, as before, generate random inputs to the game; the variation is just to illustrate that we may also code up specific test cases.

There are a few problems with the encoding of a parser IF game as presented here.

The first is that for state such as `got` that indicates the possession of a particular game object, we also had to give its complement `~got`, and make sure we maintained the invariant that exactly one of those was in the context at all times. We needed to use this hack because we have no built-in form of negation. And that fact is not just oversight or a missing feature; it runs counter to the design goals of Celf as a logically-motivated language: negation of this form in a logic program is notoriously difficult to account for in logic.

This complementary property of particular pairs of atoms is something I believe I should be able to resolve in proposed work through the use of *checked state invariants*. In a later section I will sketch what syntax and semantics of constructs like this form of negation might look like.

The other, arguably more major, problem with this example is that now that I've expanded the language of actions to include nonsense actions, like *take door*, I need to actually account for nonsense. In other words, the program will get stuck if I program the player to type such a command!

I can account for some possible failures as follows:

```
- : cur_act ('open lamp) -o {tick}.
- : cur_act ('open key) -o {tick}.
- : cur_act ('get door) -o {tick}.
```

But this does not account for any situation where the command parses properly but simply does not apply, such as if I try to move to the cellar while already in the cellar. Trying to account for each specific failure case by matching positively against it seems like a losing proposition, so we will need something better to count as success.

Hopefully at this point I have convinced you that CLF and Celf represent a promising logical paradigm for programming interactivity that nonetheless has several shortcomings we need to address. The rest of the proposal will concentrate on addressing those flaws.

4.5 Unifying story and game

Note that in the discussion of "2D" games above, while they require a higher-dimensional visual rendering space than text-based fiction, we saw that their dimensionality of *gameplay* was unrelated to this fact—a text interface can be used to render the nonlinear exploration of a 2-or-more-D world, and a 2D interface can constrain the space of meaningful actions by making progress depend on very specific movements (even if many movements are seemingly available).

An illustration of this point can be found in the 2013 game Gone Home,[12] in which a character-driven story is told through the player controlling one character's navigation of 3D space. The character does not star in the story as it takes place "in the present," but rather she discovers the story in fragments as she comes across story-relevant objects within a dense space of *irrelevant* objects.

While the controls of the game have a realistic, continuous-feeling physics model, the important narrative actions in Gone Home are discrete: picking something up, uncovering a new part of the map, entering a lock combination. The *structure* of this game and story could be entirely textual, even though one can certainly argue that the high detail and realism of the rendering, manipulation, and physics of objects lends an important qualitative element.

One of the next steps in my proposal is to see how far I can get, using existing and proposed tools, in modeling the narrative and interactive structure of such a game.

[12] http://thefullbrightcompany.com/gonehome/

What I have demonstrated via the examples in this and the previous section is a unified formalism for two types of media:

- Storytelling: establishment of setting, character motivation and choices, causal structure, plot devices;
- Game mechanics: specifying the rules governing entity interaction by specifying a *language of actions* available to the player.

By giving these examples and, as I move forward, attempting to systematize the methogology of representation, I hope to yield an approach to interactive media design that allows exploring *ludonarratives* as a unified concept, rather than patching story and game together as separate, incongruous entities.

5 Adding Phases to CLF

The first observation I have made while attempting to specify interactivity in Celf was a lack of programmer control over *when rules can fire*—consider any game where players "take turns," including single-player games where the interpreter/game engine takes a turn. In most cases, we don't want the game logic to be listening for player input constantly, but only when the internal logic has run to completion.

In general, the logic program's rules are unordered—all must be considered at every point of execution. This yields a nice way of programming when in fact we don't *want* to specify the order of execution, but it leaves us without several of the benefits of traditional programming, such as scoping and subroutines. *Phases* are effectively a way of grouping rules in the program and giving an ordered structure between the groups.

The basic idea we work from, then, is that an interactive logic program is a collection of resources and rules that, when no more rules may productively fire, may transfer control to another part of the program. A *phase* is what we call this delimited set of rules. The interphase program control is also done using linear logic programming, using special predicates `qui` $-$ and `phase` $-$ to transfer control between phases.

More formally, the program is a linear context Δ tracking current state and a set of forward-chaining rules denoted by a phase ϕ that are currently active, which run to quiescence, then pass control to another set of rules denoted by ϕ' (or to a built-in input source like `STDIN`). At that point, additional resources can be added to Δ, and the active signature may be able to continue firing. The quiescence atom is not something that we expect (or allow) rules to introduce in the program; rather, it is like a *sensing* predicate in the sense introduced by Meld. [2] [13]

The current extent of my exploration of this idea suggests it is general enough to address most of the shortcomings brought forth in previous sections.

[13] Meld is a language for programming Clayronics, which are ensembles of small robotic cubes with actuators for movement and generating colored lights. They use *sensing* predicates to determine when physically proximal cubes have sent messages, and *acting* predicates to send signals their hardware actuators (motors, lights).

5.1 Language Proposal

We start with a base language, a Prolog-like (Horn clause) fragment of CLF. For now we will also only consider linearity in the forward-chaining fragment (and persistence only in the backward-chaining fragment). In other words, this language can be thought of as Prolog plus multiset rewriting rules (linear forward chaining). This language coincides with the "replacement rule" programming paradigm (including ! for persistent predicates) described in Section 2.

Types and terms for representing data (persistent):

$$\begin{array}{rcl} K & ::= & \mathsf{type} \mid \tau \to K \\ \tau & ::= & a \mid a\, M \mid \tau \to \tau \\ R & ::= & x \mid c \mid R\, M \\ M & ::= & R \mid \lambda x{:}\tau.M \\ LFdecl & ::= & a : K \mid c : \tau \end{array}$$

Linear transition declaration types:

$$\begin{array}{rcl} p & ::= & a\, M \\ A & ::= & S \multimap \{S\} \mid \Pi x{:}\tau.A \\ S & ::= & 1 \mid p \mid S \otimes S \end{array}$$

We might also consider adding a construct dual to Π which dynamically introduces new terms of a given type:
$S ::= \cdots \mid \exists x{:}\tau.S$

This construct can be a convenient mechanism for generating new abstract names for things like locations. It should be orthogonal to most of the existing theory.

Linear contexts:

$$\Delta \quad ::= \quad \cdot \mid \Delta, x : S$$

We add atop it a way of delimiting sets of rules into phases:

Phase contents:

$$\Sigma \quad ::= \quad \cdot \mid \Sigma, r : A$$

Phase declarations:

$$phasedecl \quad ::= \quad \mathsf{phase}\ \phi = \{\Sigma\}$$

So far, what this lets us do is describe *named sets of rules*, such as

```
red : type.
blue : type.
nat : type. z : nat. s : nat -> nat.
rcount : nat -> type.
bcount : nat -> type.

phase change = {
  -init : init -o {1}.
  -change : red -o {blue}.
}

phase count = {
  -init : init -o {rcount z * bcount z}.
  -rcount : rcount N * red -o {rcount (s N)}.
  -bcount : bcount N * blue -o {bcount (s N)}.
}
```

As a first step for understanding the meaning of this delineation, we give a translation of the above example into pure CLF/Lollimon:

```
ph : type.
change : ph.
count : ph.

phase : ph -> type.
init : ph -> type.

% program-specific type families.
rcount : nat -> type.
bcount : nat -> type.

%% phase "change"
change-init : init change -o {phase change}.
change-change : phase change * red -o {phase change * blue}.

%% phase "count"
count-init : init count -o {phase count * count-rcount z * count-bcount z}.
count-rcount : phase count * rcount N * red -o {rcount (s N) * phase count}.
count-bcount : phase count * bcount N * blue -o {bcount (s N) * phase count}.
```

This translation treats the "`init`" atom specially as the one rule per phase which does not require (but rather creates) the *phase token*, `phase` P. The other rules have been modified to consume and reproduce this token.

As written, the two subprograms are isolated. However, we have not provided a top-level to determine which phase starts or when it is appropriate to change phases.

That said, the source program does not specify this information. There are a few different things we might mean, including "run the count phase followed by the change phase", "run both phases simultaneously", or "alternate the two phases in a loop, starting with the change phase."

In general, we want the ability to program *behavior at quiescence*. If we had access to some magical (runtime-generated) predicate `qui`, which would pop into existence at quiescence, we could write some global rule (outside the phases) like

```
qui * phase change -o phase count.
```

Effectively what we want to do is give the programmer access to such a predicate in a controlled way—we only expect it to make sense on the left of a transition, for instance, and we expect rules using it to maintain certain invariants pertaining to phase tokens.

What exactly are those invariants? A simple one might be "always consume a phase token and replace it with another one." That is, exactly one phase is always active.

That is approximately the invariant we will accept (for now, although it may make sense to extend the language to be able to run two phases simultaneously as well).

Thus, the grammar for linking rules is

$$L \quad ::= \quad qui \otimes \mathsf{phase}\ \phi \otimes S \multimap \{S' \otimes \mathsf{phase}\ \phi'\}$$

Because a qui token is always found with a phase token, in the future I will write $qui \otimes \mathsf{phase}\ \phi$ as `qui` ϕ.

Open question: because this is such a restricted fragment of even the diminished subset of CLF, is it worth using a different syntax to describe it, i.e. considering it a separate language? I'm currently leaning toward "no," simply because these rules *can* be given meaning using the existing interpretation of the connectives. But really what we are doing with this subset of linear logic is describing program behavior *at quiescence of a phase* (or more precisely, at quiescence of the whole program given a particular phase token is in the context), then doing standard linear logic programming at the phase level (pattern-matching and consuming state, and generating new state, including a new phase).

5.2 Compilation of linking rules

The quiescence primitive *does* have a natural interpretation in CLF, but it relies on the interaction of forward and backward chaining and the specific semantics thereof.

If we assume a formulation of our logic (a variant of *focusing* [1]) that requires all work to be done on the left-hand side of a monadic sequent before decomposing the goal, then we can interpret rules of the form

$$r : (S \multimap \{S'\}) \multimap A$$

as a way of proving A given by first runing S to quiescence, then solving S'.

This is because as soon as the higher-order premise is introduced into the context and focussed on, we enter into a state with a monadic goal, which means we postpone solving it until no more work can be done using monadic (forward-chaining) rules. If the only rules for which we care about quiescence are the monadic/forward-chaining ones, then this gives us exactly the quiescence meaning we are after.

This scheme suggests that we can string phases together in the following way:

- For each phase ϕ, designate a type run ϕ
- Define run ϕ with a backward-chaining rule with a subgoal of the form $S \multimap \{\texttt{run}\ \phi'\}$, where S is ϕ's initial state, and ϕ' is the phase to run at ϕ's quiescence.

To be more concrete, the previous example with the linking rule

```
qui change -o {phase count}.
```

can be "compiled" to Celf's fully general language as two rules:

```
r1 : run change
      o- (init change -o {run count}).
r2 : run count
      o- phase P
      o- (init count -o {phase P' * rcount R * bcount B}).
```

So far, this only demonstrates how to transfer control unilaterally from one phase to the next. We would also like to be able to "match" the state at quiescence and specify different behavior for different results; for example, in a REPL, we would like to be able to fail back to the prompt (the "read" phase) on unparseable input, rather than moving on to the evaluation phase.

This branching pattern can be written using the form of rules already described; we just need to write multiple rules that fire at quiescence of the same phase.

Thus we give the following general compilation form for linking rules:

For every P for which there are rules of the form

```
qui * phase P * S_i -o {phase P_i' * S_i'}
```

add rules

```
- : run P o- (init P -o {done P})
- : done P
      o- {S_1 * (S_1' -o run P_1')}.
...
- : done P
      o- {S_n * (S_n' -o run P_n')}.
```

With this simple construct we can start to create more complex multi-phase programs, including looping and branching patterns.

5.3 Interaction as a Phase

The original reason for developing phases was that I wanted to think of *interactive execution* as a program with (at least) two alternating phases: the game engine "takes a turn," and then the player does. The idea is that the program phase can run to quiescence, then wait for interactor input before proceeding.

As such, for my implementation I intend to have built-in phases for keyboard and mouse input. The programmer can create phase links that invoke these phases, and there will be a documented API for the atoms that they may introduce into the context, such that the programmer can also create phase links that case-analyze interactive input and continue with the next program phase.

The only theoretical component I still need to work out is when to consider an interaction phase quiescent. I intend to make this also programmer-specifyable; e.g. we can say that when "enter" is pressed or when the mouse button releases, the input phase is quiescent. (Note that this is inherently a synchronous, pull-based interactivity model; further work might extend the model in other directions, likely related to the prior question regarding simultaneous/concurrent phases.)

5.4 Example: Reading keyboard input

To get a sense for what I mean by processing interactions, here is an example keypress-processing program. We have a phase `read`, which checks for `key K` predicates in the state, and maintains the currently entered text with `entered T`.

```
phase read = {
-init : init -o {1}.
-char : key (ch C) * entered T -o
                {entered (snoc T C) * listen}.
-done : key enter -o {pressed_enter}.
}
```

If the key pressed was a character, we add it to our entered text buffer and continue listening for keys. If it was the enter key, we emit the atom `pressed_enter` and delegate to the phase link.

The phase links for this phase are:

```
- : qui * phase read * listen -o {listen * phase inputChar}.
- : qui * phase read * pressed_enter -o {phase print}.
```

Next we specify the `inputChar` phase. This phase is a proxy for human input; it emits a `key` atom and is considered quiescent after one keypress, implementing a low-level form of user input.

```
phase inputChar = {
-init : init -o {1}.
_a : listen -o {key (ch a)}.
_b : listen -o {key (ch b)}.
_c : listen -o {key (ch c)}.
_d : listen -o {key (ch d)}.
_e : listen -o {key (ch e)}.
_f : listen -o {key (ch f)}.
_enter : listen -o {key enter}.
}
```

Its one phase link transfers control back to `read`.

```
- : qui * phase inputChar -o {phase read}.
```

Finally, we add a dummy printing phase and terminal phase link.

```
phase print = {
-init : init -o {1}.
}

- : qui * phase print -o {1}.
```

The full code can be found in Appendix B, and its compilation according to the scheme described previously can be found in Appendix C.

5.5 Example: An interactive fiction game loop

The Inform 7 language for writing interactive fiction [14] divides player command processing into a few distinct phases: one *checks* that the rule is applicable, one *carries out* the rule by making changes to the game world, and the one *reports* the result of issuing the command. While this architecture may not be the most natural way to write interactive fiction in our setting, we can nonetheless account for it with phases, as the following example demonstrates.

```
%% A parser-style interactive fiction loop.

obj : type.
cmd : type.
take : obj -> cmd.
look : cmd.

action : cmd -> type.

phase read = {
  % ... recv keystrokes until enter ...
} % postcond: "entered T"

phase parse = {
  % ... tokenize string & translating it to a command ...
  - : init -o {outcome none}.
}
% postcond: outcome success + outcome none + (outcome failure * message M)

phase check1 = {
  - : init * outcome X -o {outcome none}.

  - : $action (take Obj) * inventory Obj
      -o {outcome failure * message "You already have it."}.
```

[14] http://inform7.com/

```
  - : $action look -o {outcome success}.
}
phase check2 = {
  - : $action (take Obj) * outcome none * visible Obj
      -o {outcome success}.
}


phase carryout = {
  - : action (take Obj) * in Obj C
      -o {inventory Obj * message "taken"}.
  - : action look * $in player R * $description R D
      -o {message D}.
}

phase report = {
  - : message D -o {stdout D}.
}

- : qui read -o {parse}.
- : qui parse * outcome none -o {message defaultParseError * report}.
- : qui parse * outcome failure -o {report}.
- : qui parse * outcome success -o {check1}.
- : qui check1 -o {check2}.
- : qui check2 * outcome success -o {carryout}.
- : qui check2 * outcome failure -o {report}.
- : qui check2 * outcome none -o {message default * report}.
- : qui carryout -o {report}.
```

5.6 Example: Voting Protocols

Finally, to demonstrate that the idea of phases has some applicability outside the domain of games and interactive applications, I visit an example from Henry DeYoung and Carsten Schürmann's work on formalizing voting protocols in CLF. [8]

The more complex example in their paper is a formalization of single transferrable vote (STV). The specification of STV is:

1. If a candidate has enough votes to meet the quota (#ballots/(#seats $+1$)$+1$), she is declared elected. Any surplus votes for this candidate are transferred.

2. If all ballots have been assigned to candidates and no candidate meets the quota, then the candidate with the fewest votes is eliminated and her votes are transferred. If several candidates tie for the fewest votes, one is eliminated at random.

3. When a vote is transferred, it is assigned to the hopeful candidate with the next highest preference listed on that ballot. That is, candidates that are already elected or defeated do not receive transferred votes.

4. If, at any point, there are at least as many open seats as hopeful candidates remaining, then all remaining hopefuls become elected.

There are a few program patterns that implementing this protocol needs that CLF does not support naturally: keeping track of the number of uncounted ballots so that we can do something different when there are none left, finding the minimum numerical index in a collection of predicates, and comparing the number of seats to the number of hopefuls so that we can elect everyone once there are fewer hopefuls than seats.

They implement these patterns using phase tokens and keeping numerical invariants like the index of the `count_ballots` phase for tracking the number of uncounted ballots, which must consistently match the number of `uncounted-ballot` atoms in the state at all times.

To illustrate these techniques, I will give the code for one of the program's phases. The meanings of a few relevant predicates are given below:

predicate	meaning
`count-ballots S H U`	phase token for ballot counting indicating `S` open seats, `H` hopeful candidates, and U uncounted ballots remaining
`defeat-all`	phase token for marking all remaining candidates as defeated.
`defeat-min S H M`	phase token for computing the minimum-vote candidate indicating `S` open seats, `H` hopeful candidates, and `M` potential minimums remaining
`uncounted-ballot C L`	uncounted ballot w/highest pref for `C` & lower prefs `L`
`counted-ballot C L`	counted ballot w/highest pref for `C` & lower prefs `L`
`hopeful C N`	`C` is a candidate with `N` votes so far

Here are the rules defining the `count-ballots` phase in the original Celf code:

```
count/1 : count-ballots S H (s U) *
          uncounted-ballot C L *
          hopeful C N *
          !quota Q *
          !nat-less (s N) Q
            -o {counted-ballot C L *
                hopeful C (s N) *
                count-ballots S H U}.

count/2 : count-ballots (s (s S)) (s H) (s U) *
          uncounted-ballot C L *
          hopeful C N *
          !quota Q *
          !nat-lesseq Q (s N) *
          winners W
```

```
            -o {counted-ballot C L *
                !elected C *
                winners (cons C W) *
                count-ballots (s S) H U}.

count/3 : count-ballots (s z) H U *
          uncounted-ballot C L *
          hopeful C N *
          !quota Q *
          !nat-lesseq Q (s N) *
          winners W
            -o {counted-ballot C L *
                !elected C *
                winners (cons C W) *
                !defeat-all}.

count/4_1 : count-ballots S H U *
            uncounted-ballot C (cons C' L) *
            !elected C
              -o {uncounted-ballot C' L *
                  count-ballots S H U}.

count/4_2 : count-ballots S H U *
            uncounted-ballot C (cons C' L) *
            !defeated C
              -o {uncounted-ballot C' L *
                  count-ballots S H U}.

count/5_1 : count-ballots S H (s U) *
            uncounted-ballot C nil *
            !elected C
              -o {count-ballots S H U}.

count/5_2 : count-ballots S H (s U) *
            uncounted-ballot C nil *
            !defeated C
              -o {count-ballots S H U}.

count/6 : count-ballots S H z
            -o {defeat-min S H z}.
```

The first two rules simply add ballots to a candidate's tally. The third rule observes that we are filling the last seat and invokes the `defeat-all` phase. Rules 4.1-5.2 handle ballots marked for candidates that are already elected or defeated. Finally, rule 6 covers the case wherein we have counted all ballots but not filled all seats, in which case we switch to a phase for the determining the minimum-vote candidate so that we can transfer her votes to lower-preference candidates.

The indices in the `count-ballots` predicate track how many instances of `uncounted-ballot` occur in the context, although this fact is not expressed formally, much less checked, so the programmer must take care to maintain it. Similarly, it tracks the number of `hopeful` instances. It tracks these values so that we can write rules that fire when counting is "done," since we cannot otherwise test for the *absence* of a predicate.

In a phase-structured language, we can write the code more directly. We no longer need a `count-ballots` token for the phase, and we do not need to track the number of uncounted ballots and hopefuls remaining. We still need to track the number of seats remaining, but that can be its own separate predicate. We use the semantics of quiescence for the phase and at the end, branch on whether any seats remain to determine the next phase.

```
phase count = {
1 : uncounted-ballot C L * hopeful C N * !quota Q * !nat-less (s N) Q
    -o {counted-ballot C L * hopeful C (s N)}.

2 : seats-remaining (s S) * uncounted-ballot C L * hopeful C N *
    !quota Q * !nat-lesseq Q (s N) * winners W
    -o {counted-ballot C L * !elected C * winners (cons C W) *
        seats-remaining S}.

3_1 : uncounted_ballot C (cons C' L) * !elected C
        -o {uncounted-ballot C' L}.

3_2 : uncounted-ballot C (cons C' L) * !defeated C
        -o {uncounted-ballot C' L}.

4_1 : uncounted-ballot C nil * !defeated C -o {1}.

2_2 : uncounted-ballot C nil * !elected C -o {1}.
}

% Phase transitions
- : qui count * seats-remaining (s S)
    -o {seats-remaining (s S) * phase defeat-min}.
- : qui count * seats-remaining z -o {phase defeat-all}.
%% as a bonus, we can do some error checking!
- : qui count * uncounted_ballot _ _ -o {error}.
```

It remains to prove that this program has the same semantics as the original CLF code with invariants tracking. Since it will probably be easier to justify in terms of a source-level semantics for phases, rather than the compiled program, and the source-level semantics is still in progress, I leave this proof to be completed for the thesis.

5.7 Negation

Previously, in the example with parser interactive fiction in section 4, I said that I would be able to address "negation," i.e. testing for the presence of a particular atom. Being able to program with quiescence lets us encode this idea.

Suppose `foo` is the atom we want to test for. We can encode a rule $\neg$`foo` $\multimap \{A\}$ as

```
phase testFoo = {
  - : init -o {present false}.
```

```
  - : foo * present false -o {present true}.
}

qui testFoo * present false -o {A}.
```

5.8 Source-level semantics

The interpretation of phase-structured programs into (higher-order, mixed-chaining) CLF programs depends on a particular interpretation of the monadic formula $\{B\}$ as a *goal* of proof search. It requires that we work on everything to the left of the $\vdash$ in the sequent, and, only when no more work can be done, work on the right. The problem with this interpretation is that it lacks a precise coincidence with provability in any known logic: it is *sound* in the sense that it will not admit sequents not admitted by a well-behaved logic, but it is *incomplete* in the sense that it must forbid certain proofs (that, say, break the quiescence condition) even though they are perfectly valid.

This issue combined with the fact that anecdotally speaking, higher-order mixed-chaining rules are hard to reason about, motivates me to give a direct (source-level) semantics to phase-structures logic programs.

I will introduce a judgment that is effectively a transition relation between states $\Delta \rightsquigarrow \Delta'$, decorated with the needed pieces of the signature to specify all of the possible steps. This judgment builds on the idea of treating sequent calculus rules working on the left

$$\frac{\Delta' \vdash C}{\Delta \vdash C}$$

as transitions $\Delta \rightsquigarrow \Delta'$, ignoring the goal C, a la the methodology discussed by Deng et. al. [7]

First, we need to track the current phase in addition to the current state. As we maintain the invariant (for now) that at most one phase is active, we can simply write the phase name as a distinguished atom in the state:

$$\langle \Delta, \phi \rangle \rightsquigarrow \langle \Delta, \phi' \rangle$$

Next, we need to account for which signature–set of LF declarations LF, phase declarations Φ, and phase linking rules Λ— we are operating under. Call the whole program bundle Ξ. Then we write

$$\langle \Delta, \phi \rangle \rightsquigarrow_{\Xi} \langle \Delta', \phi' \rangle$$

We wish to define this judgment such that something like the following metatheorem holds:

If $(\Delta, \phi) \rightsquigarrow_{\Xi} (\Delta', \phi')$
where Ξ compiles to CLF signature C,
then $\Delta, \mathrm{phase}\,\phi \rightsquigarrow_{C} (\Delta', \mathrm{phase}\,\phi')$.

The following definition of the judgment is speculative; I have not yet proven that the above metatheorem holds, but I am using it as a guidance metric for developing these rules.

The key rules are these two:

$$\frac{\Phi \vdash \phi \mapsto Rs \quad \Delta \rightsquigarrow_{(LF,Rs)} \Delta'}{\langle \Delta, \phi \rangle \rightsquigarrow_{(LF,\Phi,\Lambda)} \langle \Delta', \phi \rangle} \text{ within-phase}$$

$$\frac{\Phi \vdash \phi \mapsto \Sigma \quad \Sigma \vdash \Delta, \Delta' \; \texttt{quiescent} \quad \Lambda \vdash \texttt{qui} \; \phi \otimes S \multimap \{\texttt{phase} \; \phi' \otimes S'\} \quad \Delta \vdash_{LF} S}{\langle \Delta, \Delta', \phi \rangle \rightsquigarrow_{(LF,\Phi,\Lambda)} \langle \Delta', S', \phi' \rangle} \text{ phase-transition}$$

We can think about the metatheorem in relation to these two rules: the first one (within-phase) should be straightforwardly connected to the compilation of phases into rules that are "guarded" by phase tokens.

The second rule, which appeals to quiescence and makes a phase transition, will probably represent the bulk of the work.

The main missing piece lies in how I define $\Sigma \vdash \Delta$ quiescent. Because I want this notion to coincide with the peculiar semantics of Celf's mixed-chaining, effectively what I plan to do is identify the two notions of quiescence (but perhaps otherwise leave them as "implementation-defined," or simply write the quiescence-checking program in inference rules).

Another plausible answer is that I use the machinery sketched in the next section of *characteristic generators*. The short version is that this means I write a complete description of a program's quiescent states, which can be machine-checked; then, this rule simply checks that the state Δ can be generated by Σ's quiescent-state generator.

6 Characteristic Generators

In this section I describe a simple program analysis mechanism for phases called *characteristic generators*. The name has its origins in *generative invariants*, as described in Robert Simmons's thesis, [23] which use logic programming signatures to *generate* properties that hold invariantly for a program. Because some of the properties we want to check are not invariants, I use the term generative *characteristics*. (It is also a pun on the idea of a "characteristic function" of a set: the generators *characterize* a phase.)

In what I have described so far, we can write a lot of interesting logic programs, but as they get bigger we have to hold more things in our heads about the state until it becomes unmanageable. This analysis mechanism aims to provide an interface for the developer to describe facts about the program which can be checked and should change infrequently as the program grows.

6.1 Generative Invariants

Characteristic generators are a generalization of *generative invariants*, which are linear logic signatures that represent a complete set of possible states for another linear logic program:

anything generated by the source program may be generated by the invariant signature. Formally, a generative invariant of a CLF signature Σ, call it Σ_{Gen}, is said to describe a *world* (set of states) W iff for any state $\Delta \in W$, $\Delta \rightsquigarrow_{\Sigma} \Delta'$ implies $\Delta' \in W$.

We use the constructs already available in the logic programming formalism to describe a set of contexts that may be generated, and we can check that another set of rules (the actual program we're interested in) creates contexts in that generated set.

For example, here is a coin-changing program that changes a quarter into two dimes and a nickel and a dime into two nickels.

```
q -o {d * d * n}.
d -o {n * n}.
```

Call this signature Σ_{coins}. The following logic program is a generative signature representing contexts that contain only quarters, nickels, and dimes:

```
gen -o {q * gen}.
gen -o {d * gen}.
gen -o {n * gen}.
gen -o {1}.
```

(The `gen` token, or generation token, is assumed to be the initial seed for this logic program.) Call this signature Σ_{coins}^{gen}.

As a rough sketch, to show that Σ_{coins}^{gen} is an invariant of Σ_{coins}, we need inductively analyze the trace, i.e. the sequence of rules applied to arrive at a particular state Δ. We case-analyze the last rule applied in the trace, and for each rule in Σ_{coins} that could have generated Δ from Δ^{-}, show that if Δ^{-} is in the world W generated by Σ_{coins}^{gen} (which we get by inductive hypothesis), then so is Δ.

Let's look at a slightly more complex example, the Blocks World introduced in section 2. First we will think informally about the invariant shape of Blocks World states: we always have one arm, either in **free** or **holding** state. In addition, we have arbitrarily many blocks. Each block (other than the one possibly held by the arm) is either on the table or on another block. Additionally, if it is on the table or on another block, either there is another block on it or it is clear.

```
gen -o {genArm * genBlocks}.

genArm -o {arm_free}.
genArm -o {arm_holding X}.

genBlocks -o {on_table X * genTop X * genBlocks}.
genBlocks * genTop Y -o {on X Y * genTop X * genBlocks}.
genBlocks -o {1}.

genTop X -o {clear X}.
```

Again this signature would be seeded with an initial `gen` atom, and the various nonterminals (`genArm`, `genBlocks`, `genTop`) produce valid states for the simulation.

6.2 Active and Quiescent States

My proposed extension to this idea includes describing not just *invariants* of the program but also the set of states that may hold *at quiescence*, or during all the parts of execution that aren't quiescence, which I call *activity* (the program is "active" if there is a rule that can fire).

For example, the active states for the coin exchange program are those in which there is at least one quarter or dime in the context, described by the generative signature below seeded with `act`.

```
act -o {q * act'}.
act -o {d * act'}.
act' -o {act}.
act' -o {1}.
```

The quiescent states are those which contain only nickels, described by the generative signature below seeded with `qui`.

```
qui -o {qui * n}.
qui -o {1}.
```

For the blocks world example, the active states are those with either the arm holding a block, or the arm free and at least one clear block.

```
act -o {arm_holding X * actBlocks}.
act -o {arm_free * clear Y * actBlocks}.
actBlocks -o {on_table X * actBlocks}.
actBlocks -o {on X Y * actBlocks}.
actBlocks -o {clear X * actBlocks}.
actBlocks -o {1}.
```

6.3 Generators and Phases

My aim is for the ability to write, and have automatically checked, generative characteristics on a *per-phase* basis will allow us to express invariants of the game world and avoid common bugs that can be introduced in complex interactive environments. For example, we could ensure that game objects are never lost or duplicated (by expressing the property that one copy always exists, either in the player's inventory or accessible from the game world).

Additionally, by giving generators on a per-phase basis, we can require that if a phase ϕ precedes another phase ϕ', ϕ's quiescent states satisfy any invariants for ϕ'. This lets us impose restrictions on when it makes sense to compose two phases.

6.4 First-Order Logic Front-end

In typical descriptions of Blocks World, invariants are states as follows:[15]

[15] Examples taken from `http://www.cs.cf.ac.uk/Dave/AI2/node116.html`

- If the arm is holding a block it is not empty
- If block A is on the table it is not on any other block
- If block A is on block B, block B is not clear.

These descriptions seem in some ways more intuitive than the generative descriptions: they identify *bad* states and rule them out, rather than crafting a *positive* description of permitted states in the negative space defined by the forbidden states. As part of my thesis work, I would like to investigate the plausibility of using first-order logic formulae over states as a "front-end" to generators: ideally to extract generators from propositions, but perhaps just an ability to formulate both and prove a coincidence between them (or implication from the generator to some desired properties).

7 Proposed Work

7.1 Theory of Phases

I will continue developing the theory and metatheory of phases for linear logic programming.

Next step: iterate on the formulation of metatheoretic properies of phases. I have sketched an independent semantics and a correlation theorem between that semantics and the CLF compilation, but I need to prove the theorem and perhaps change the language design to meet it.

7.2 Large Examples

As I work on the language definition, I also plan to develop bigger examples that capture key properties of my target domain. I will flesh out the interactive fiction example and provide a few example games that could be built atop it. I will give examples based on other game paradigms, such as 2D tile games and first-person navigation and storytelling. I will not address game mechanics that are fundamentally based on elaborate models of physics of continuous, time-sensitive dynamics – these fall outside the scope of high-level game structure that I am trying to describe. In other words, the scope of my examples could be described as "turn-based games." I may encode some nondigital games (e.g. board games or playground games) as well, just to suggest the generality of the description mechanism.

I am developing a growing catalogue of examples, which the reader can find on my GitHub site for this project. [16]

These examples include:

- Card dealing (`cards.clf`)
- Abstracted two-player combat (`combat.clf`)

[16] https://github.com/chrisamaphone/interactive-lp/tree/master/examples

- An abstracted platformer (`platformer.clf`)
- An abstraction of Pong (`pong.clf`)
- Dialog simulation (`dialog.clf`)
- An elevator (`elevator.clf`
- An encoding of an Egyptian myth (`isis.clf`)
- Combat with random enemy spawning and item drop (`rl-query.clf`)

Most examples in the directory are written in (phaseless) Celf, and I hope to rewrite them in a simpler way using phase-based CLF.

A growing trend in the type of programs I am interested in specifying is that they lean towards *emergent storytelling*, *interactive simulation*, and *procedural generativity* more than traditional, human-crafted stories and puzzles.

The genre colloquially known as "roguelikes" strikes a chord with these interests. A roguelike typically has a lot of randomly-generated content, often intended to be adversarial to the player. The governing rules are often complicated and unpredictable, but many roguelikes are turn-based, making them prime subjects for my system. The particular example in `rl-query.clf` demonstrates the seeds of these ideas.

Combining generativity and reactivity is especially interesting to me, and a larger example I have in mind to develop is one that takes the idea of "randomly-generated levels" a step further: the player can only see a part of the level, and as they move through the game world, rather than incrementally discovering a pre-made level, the level builds *dynamically* around and in response to the player. For example, in a platformer setting you could imagine placing platforms in response to the player's jumps, and this could be done in either a helpful or adversarial (or just completely random) way. In the interactive fiction setting, I imagine this idea taking shape more in terms of dialogue with non-player characters.

7.3 Programmatic Analysis

After designing and demonstrating the efficacy of the core language, I will develop the theory of characteristic generators as outlined in Section 6. After working on a number of examples of such generators demonstrating their desirability for checking properies of programs, I will work on a scheme for mechanically checking them within my prototype framework.

In addition, I plan to develop a methodology for observing, querying, and studying *causal structure* in narrative worlds. This plan is inspired by three lines of development: the *CelfToGraph* tool described in section 3, which displays "causal edges" between narrative actions in a trace; the PlotEx tool developed by interactive fiction author Andrew Plotkin, [17] and the theory of *subordination* for LF.

[17] http://eblong.com/zarf/plotex/

Since I have not previously described PlotEx, the quick summary is that it is a tool written in Python for describing actions in interactive fiction (like movement and object acquisition), of which an author can then ask queries such as, "If I introduce a new solution to puzzle X, will the player be able to circumvent the constraints of puzzle Y later in the game?" and "If I suppress the availability of a certain object or action, can the player ever reach the final dungeon?"

CelfToGraph is designed to answer similar questions, although I have not explored its efficacy towards widely-branching, game-oriented structures.

The artifact I want to produce as an end goal of this investigation would probably take the form of a tool inspired by CelfToGraph adapted for my language supporting the visual observation of, and logical queries on, dependency data between actions and resources.

One direction I have been working in that may shed some light on causal and dependency structures is a theory of *subordination*. In LF, subordination is a relationship between constant type families a and b that holds if constructors for a cannot possibly be used to construct b. Inferring these relationships enables a *strengthening theorem*:

If $\Gamma, x{:}a\ M \vdash N : b\ P$ *and* a *is not subordinate to* b*, then* $\Gamma \vdash N : b\ P$.

Since subordination has been useful for the metatheory and implementation of LF, researchers in the CLF community have been interested in whether a similar theorem can be stated in the CLF setting. A simple form of subordination can be defined as stipulating that $\Sigma \vdash a \prec b$ iff there is a rule of the form $r : a\ M \to \cdots \to b\ N$, closed under transitivity. So perhaps this extends just as easily to the linear case $r : a\ M \multimap \ldots \multimap b\ N$.

The difficulty is that for *linear* proofs $\Delta, x{:}A \vdash M : B$, at least in the fragment of linear logic we have been using, we know we must use every variable in the context, so we cannot strengthen it away.

If instead we look at left-sided sequents, i.e. the forward chaining judgment $\Delta \leadsto^{\epsilon} \Delta'$, we can get more traction: knowing that $a \not\prec b$ means that $\Delta, a \leadsto \Delta', b$ must have a particular shape; in particular $\Delta \neq \Delta'$. Furthermore, we know that if the last rule we applied acted on a, then b must have already been in the original context Δ. And if we look at the *trace term* ϵ, which we have seen in the output to our queries as being made up of terms that look like

```
let [X1...Xn] = rule Y1 ... Yn
```

we notice that then *dependency structure* between the variables here is constrained by the subordination relation. In a sense, it gives us a static analysis of what kinds of traces we can expect to have.

The reason I think this idea might have something to do with the program analysis ideas mentioned previously is that *CelfToGraph* also gives us a notion of resource dependency. The graphs it shows us are relationships between *actions* rather than resources, but those relationships are defined by the consumption/production of resources determined by each action.

I believe that the theory of subordination bears further investigation as a potential theory to ground the casual relationships inferred by *CelfToGraph*, and in general to ground the study of causality and resource dependency in widely-branching narrative worlds.

7.4 Implementation of Prototype and Tools

The following list includes ways to bridge the gap between the capabilities afforded by Celf and the ideal framework for game mechanic experimentation and playable prototype-making.

I propose to implement these ideas atop a simple prototype of forward-chaining linear logic programming (already in development).

I have been co-developing with other members of the CLF community a prototype forward-chaining linear logic programming engine that we hope to make readable, well-documented, and extensible. I would like to build upon this prototype an implementation of phases. I will build the following, some of which may extend this prototype and others may extend the more fully-developed Celf implementation:

- Initial state specifications
- Proof irrelevance support for consume-then-produce patterns
- Visualization tools for traces
- Support for checking of characteristics specified with generators
- Reinvent the query/trace language, perhaps unifying them, to make something more amenable for programs that may not intend to terminate.

7.5 Compilation to/from Twee

Twee is the source language underlying Twine. For simple branching narratives and atomic pieces of state, I believe there is a relatively straightforward translation between a CLF representation and a Twine representation.

I have already sketched a translation from Twine passage graphs to Celf rules in section 4. Implementing this translation could widen the appeal of the ideas to use Twine's friendly front-end as a surrogate for linear logic notation.

Going the other direction would serve the purpose of doing most of the structural game development in linear logic, importing that structure over to Twine, then doing the actual descriptive writing in that setting. I am not sure whether that idea has as much value in terms of enabling a more useful workflow, but I believe that it would help me demonstrate what CLF is doing, i.e. serve as a waypoint for playable prototyping.

8 Means of Evaluation

To restate my thesis statement:

Phase-structured linear logic programming can form the basis of a framework for specifying, testing, and inventing ludonarrative mechanics.

Having now explained the terminology in this statement, I will explain how I hope to defend each of its key pieces:

Phase-structured linear logic programming can form the basis: I have given a linguistic definition of phase-strutured linear logic programming; I will verify that it forms a suitable basis for programming by proving the correspondence theorem outlined in section 5 and giving several example programs.

A framework: By "framework" I mean not just the language but also tooling around the language to make it easier to understand, debug, and refine programs. I will validate this part of the thesis statement by building a working prototype of the language with convenient syntactic sugar for common programming patterns, programmatic means of specifying initial states, means of inspecting and visualizing traces as they execute, and means of analyzing the causal structure of a program a la section 3.

Specifying: The language provides a means to write *executable specifications* of (e.g.) a game. In some cases, such a specification might suffice as an actual implementation, supposing rich enough tool support for "playing" the execution. However, it is *not* my aim to support the full production of industrial-quality games. Specifying a game should be considered an initial step of design.

Testing: The language should support both automatic "fuzz testing" of encodings by using the inference engine to act as a proxy for the player and nondeterministically select actions. It should also support manual testing via an interactive (playable) interface.

Inventing: I want to situate this tool within an iterative design process. The *specifying* and *testing* requirements only suggest that I should be able to encode existing games; as an additional criterion, I would like to use this tool to specify new games and also demonstrate that at least one person other than me can do so as well. I hope to have at my defense several examples of such work that my committee can judge as "interesting" on whatever subjective scales they have.

One way of making "invention" more concrete is that a large class of games I want to be able to specify could be described as *interactive simulations*: we program the parameters of a scenario, then let it run for an extended duration, generating content along the way. In this way the computer can aid the creative process by generating patterns that would never have been intentionally designed. This methodology borrows ideas from the field of *computational creativity*, except that rather than necessarily accepting the machine-generated work as the final masterpiece, we use it as fuel for human ideas. Toward this end, I have in mind several fairly large simulation/generation-style examples that I would like to get working in my framework as evidence of its inventive power.

Ludonarrative mechanics: Finally, I want to place an emphasis on *story games* in particular from within the quite broad space of what counts as a "game". By a story game I mean one in which the primary mechanic of the game is progress through a narrative, rather than manual dexterity or manipulation of physical space. Storytelling may be emergent, as in the interactive simulation setting described above. Even sufficiently abstract descriptions of puzzle-solving, combat, or platforming may qualify. But to the extent that the game provides the player with a narrative, the structure of that narrative should closely align with the structure of the game. By being precise about what I mean by *structure* and by providing graphical means of observing that structure, I will be able to demonstrate the success of this metric.

The standards I will set for myself, and thus ask my committee to keep in mind for this work, are based on what I know of the state of the art in free, accessible game construction tools. These include (previously mentioned and linked) frameworks such as Twine, Inform 7, Unity, GameMaker, StageCast Creator, PuzzleScript, and PlotEx.

While I don't expect my project to be as polished as these tools in terms of feature-completeness and interface design, nor in terms of the extensive mechanic-specific libraries that these tools have implemented (such as Unity and physics or Inform7 and its conversation engine), I hope that what I can create *linguistically* is at least as expressive and simple to use and reason about. Linguistically, most of these tools provide a thin veneer of accessibility capable of very limited expression, and then to do anything interesting, a developer must delve into JavaScript, C#, or other hacking in a standard industry language. The whole premise of this project is that, in terms of language design, *we can do better*. That is, we can have a clean, small, logically-motivated model of the core constructs involved in game design without resorting to "kitchen sink"-style languages. Additionally, very few of these tools provide any way of analyzing (either with eyeballs or with computers) the code that you write with them; I certainly expect to compare favorably to these tools along those lines.

I don't expect to have time to carry out empirical user studies, but I hope to make a qualitative comparison of my work to the linguistic interfaces of these tools.

9 Summary and Timeline

I have outlined the steps of creating a ludonarrative laboratory based on linear logic programming. I have established that an existing framework, Celf, offers a promising beginning for encoding ludonarratives, and I have sketched proposals for addressing each of its shortcomings. The four key difficulties and plans for overcoming them are:

Difficulty	**Solution**
Sometimes we want to impose partial orderings among rules.	Language proposal with phases.
Programming with state is hard to reason about!	Machine-checked invariants and other characterizations of states; analysis tools such as causality and dependency graphs.
Non-interactive, low-feedback programming workflow.	Visual state editor and trace rendering.
Lack of access to common game programming libraries for e.g. graphical rendering, text parsing, etc.	Implementation of a translation from the language to existing game frameworks (e.g. Twine) or as a scripting language within such frameworks (e.g. Unity).

With these pieces in place, I hope to have a convincing proof-of-concept unifying several existing frameworks' notion of game design into a cohesive, general whole, offering a basis for designers to invent and iterate upon novel ideas in interactive media.

I expect to defend in Summer 2015 according to the following timeline.

- **Spring 2014:** Finish working out theoretical concerns (language semantics, proofs, and sketch of phase contract checking)

- **Summer-Fall 2014:** Implementation and development of examples
- **Spring 2015:** Write dissertation
- **Summer or Fall 2015:** Defend dissertation

References

[1] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992. 5.2

[2] Michael P. Ashley-Rollman, Peter Lee, Seth Copen Goldstein, Padmanabhan Pillai, and Jason D. Campbell. A language for large ensembles of independently executing nodes. In *Proceedings of the 25th International Conference on Logic Programming*, ICLP '09, pages 265–280, Berlin, Heidelberg, 2009. Springer-Verlag. 5

[3] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *In Theorem Proving in Higher Order Logics: 18th International Conference, number 3603 in LNCS*, pages 50–65. Springer-Verlag, 2005. 1

[4] Anne-Gwenn Bosser, Marc Cavazza, and Ronan Champagnat. Linear Logic for non-linear storytelling. In *ECAI 2010*, volume 215 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2010. 1, 3

[5] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-2002-002, Department of Computer Science, Carnegie Mellon University, March 2002. Revised May 2003. 2.10

[6] Karl Crary. A syntactic account of singleton types via hereditary substitution. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, LFMTP '09, pages 21–29, New York, NY, USA, 2009. ACM. 1

[7] Yuxin Deng, Iliano Cervesato, and Robert J. Simmons. Relating reasoning methodologies in linear logic and process algebra. In Sandra Alves and Ian Mackie, editors, Proceedings 2nd International Workshop on *Linearity*, Tallinn, Estonia, 1 April 2012, volume 101 of *Electronic Proceedings in Theoretical Computer Science*, pages 50–60. Open Publishing Association, 2012. 5.8

[8] Henry DeYoung and Carsten Schürmann. Linear logic voting protocols. In *VoteID 2011*, pages 53–70. Springer LNCS 7187, 2012. 5.6

[9] Gustave Flaubert. *Plans et Scénarios de Madame Bovary*. Zuma, Cadeilhan, 1995. In French. Presentation, transcription and notes by Yvan Leclerc. 3.1

[10] Gustave Flaubert. *Madame Bovary*. Folio, 2001. In French. Also available in English, OUP Oxford, reissue edition, 2008. 3.1, 2

[11] Deepak Garg and Frank Pfenning. A proof-carrying file system. In D.Evans and G.Vigna, editors, *Proceedings of the 31st Symposium on Security and Privacy (Oakland 2010)*, Berkeley, California, May 2010. IEEE. Extended version available as Technical Report CMU-CS-09-123, June 2009. 1

[12] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987. 2.1

[13] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. 1, 2.5

[14] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In *2007 Symposium on Principles of Programming Languages*, 2007. 1

[15] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent Linear Logic programming. In *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, 2005. 2.4, 2.10

[16] Chris Martens, Anne-Gwenn Bosser, Joao F. Ferreira, and Marc Cavazza. Linear logic programming for narrative generation. In *Logic Programming and Nonmonotonic Reasoning 2013*, 2013. 1, 3

[17] M. Masseron. Generating plans in Linear Logic: II. A geometry of conjunctive actions. *Theoretical Computer Science*, 113(2):371–375, 1993. 1

[18] M. Masseron, Christophe Tollu, and Jacqueline Vauzeilles. Generating plans in Linear Logic: I. Actions as proofs. *Theoretical Computer Science*, 113(2):349–370, 1993. 1

[19] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206. Springer LNAI 1632, 1999. 1

[20] Julie Porteous, Marc Cavazza, and Fred Charles. Applying planning to interactive storytelling: Narrative control using state constraints. *ACM Trans. Intell. Syst. Technol.*, 1(2):10:1–10:21, December 2010. 1

[21] Anders Schack-Nielsen and Carsten Schürmann. Celf — a logical framework for deductive and concurrent systems (system description). In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'08)*, pages 320–326. Springer LNCS 5195, 2008. 1

[22] Anders Schack-Nielsen and Carsten Schürmann. Celf — a logical framework for deductive and concurrent systems (system description). In *Automated Reasoning*, pages 320–326. Springer, 2008. 2.10

[23] Robert J. Simmons. *Substructural Logical Specifications*. PhD thesis, Carnegie Mellon University, 2012. 6

[24] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The propositional fragment. In *TYPES*, pages 355–377, 2003. 1, 2.10

[25] R. Michael Young. 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. 1

A Full parser IF specification with test input

```
% A simple parser IF games based on twine-interact.clf
% combinatory verbs + nouns.

object : type.
key : object.
lamp : object.
door : object.

got : object -> type.
~got : object -> type. % complement.

% Passages
passage : type.
start : passage.
den : passage.
cellar : passage.
dark : passage.
location : passage -> type.

% game-internal state.
opendoor : type.
lose : type.
win : type.

% Externally available (controller) actions.
action : type.
'get : object -> action.
'open : object -> action.
'move : passage -> action.

% To specify an order on actions
nat : type.
z : nat.
s : nat -> nat.
nth_act : nat -> action -> type.
% A specific test case for player action.
% 'starttoden, 'getlamp, 'getkey, 'dentocellar, 'cellartodoor.
act0 : nth_act z ('move den).
act1 : nth_act (s z) ('get lamp).
```

```
act2 : nth_act (s (s z)) ('get key).
act3 : nth_act (s (s (s z))) ('move cellar).
act4 : nth_act (s (s (s (s z)))) ('open door).

% current action
cur : nat -> type.
cur_act : action -> type.

% Transition rules. Every rule from the previous version has an additional
% cur_act premise, and outputs a "tick" whenever we want to return control
% to the player.

tick : type.
next_act : tick * cur N * nth_act N A -o {cur_act A * cur (s N)}.

start_to_den : cur_act ('move den) * location start
                  -o {location den * tick}.
start_to_cellar : cur_act ('move cellar) * location start
                  -o {location cellar * tick}.

den_to_cellar : cur_act ('move cellar) * location den
                   -o {location cellar * tick}.
den_to_lamp : cur_act ('get lamp) * location den * ~got lamp
                  -o {got lamp * location den * tick}.
den_to_key : cur_act ('get key) * location den * ~got key
                 -o {got key * location den * tick}.

cellar_to_den : cur_act ('move den) * location cellar -o
                    {location den * tick}.
cellar_to_door : cur_act ('open door) * location cellar -o {opendoor}.

open_door_without_key : opendoor * ~got key
                            -o {location cellar * ~got key * tick}.
open_door_with_key : opendoor * got key
                            -o {location dark}.

dark_with_lamp : location dark * got lamp -o {win}.
dark_without_lamp : location dark * ~got lamp -o {lose}.

% failures?
- : cur_act ('open lamp) -o {tick}.
- : cur_act ('open key) -o {tick}.
- : cur_act ('get door) -o {tick}.
% what about when conditions don't apply?

% Reporting
ending : type.
w : ending. l : ending.
report : ending -> nat -> type. % the nat is the # of steps.
report_win : win * cur N -o {report w N}.
report_loss : lose * cur N -o {report l N}.
```

```
% Initial state
init : type = {~got key * ~got lamp * location start * cur z * tick}.

#trace * init.

#query * * * 1 init -o {report END NSTEPS}.
```

B Phase-structured program for reading keyboard input

```
% Compiling a looping program (readline)

char : type.
a : char.
b : char.
c : char.
d : char.
e : char.
f : char.

keycode : type.
ch : char -> keycode.
enter : keycode.
key : keycode -> type.

text : type.
nil : text.
snoc : text -> char -> text.

listen : type.
pressed_enter : type.
entered : text -> type.

% phase code

phase readline = {
-init : init -o {entered nil * listen}.
}

phase read = {
-init : init -o {1}.
-char : key (ch C) * entered T -o
                {entered (snoc T C) * listen}.
-done : key enter -o {pressed_enter}.
}

phase inputChar = {
-init : init -o {1}.
_a : listen -o {key (ch a)}.
_b : listen -o {key (ch b)}.
_c : listen -o {key (ch c)}.
_d : listen -o {key (ch d)}.
```

```
_e : listen -o {key (ch e)}.
_f : listen -o {key (ch f)}.
_enter : listen -o {key enter}.
}

phase print = {
-init : init -o {1}.
}

% linking code: readline; read; (inputChar; read)*; print

- : global_init -o {phase readline * init}.

- : qui * phase readline -o {phase read}.
- : qui * phase read * listen -o {listen * phase inputChar}.
- : qui * phase read * pressed_enter -o {phase print}.
- : qui * phase inputChar -o {phase read}.
- : qui * phase print -o {1}.

#query * * * 10 global_init -o {entered T}.
```

C Compiled keyboard reader

```
% Compiling a looping program (readline)

ph : type.
readline : ph.
read : ph.
inputChar : ph.
print : ph.

char : type.
a : char.
b : char.
c : char.
d : char.
e : char.
f : char.

keycode : type.
ch : char -> keycode.
enter : keycode.

key : keycode -> type.

text : type.
nil : text.
snoc : text -> char -> text.

listen : type.
pressed_enter : type.
```

```
entered : text -> type.

% linking atoms
run : ph -> type.
init : ph -> type.
phase : ph -> type.
done : ph -> type.

% phase code
readline_init : init readline -o {phase readline * entered nil * listen}.

read_init : init read -o {phase read}.
read_char : phase read * key (ch C) * entered T -o
                {phase read * entered (snoc T C) * listen}.
read_done : phase read * key enter -o
                {phase read * pressed_enter}.

input_init : init inputChar -o {phase inputChar}.
input_a : phase inputChar * listen -o {phase inputChar * key (ch a)}.
input_b : phase inputChar * listen -o {phase inputChar * key (ch b)}.
input_c : phase inputChar * listen -o {phase inputChar * key (ch c)}.
input_d : phase inputChar * listen -o {phase inputChar * key (ch d)}.
input_e : phase inputChar * listen -o {phase inputChar * key (ch e)}.
input_f : phase inputChar * listen -o {phase inputChar * key (ch f)}.
input_enter : phase inputChar * listen -o {phase inputChar * key enter}.

print_init : init print -o {phase print}.

% linking code: readline; read; (inputChar; read)*; print
run0 : run readline
           o- (init readline -o {run read}).

run1 : run read
           o- phase P
           o- (init read -o {done read}).

done1 : done read
            o- listen  %% test & preserve
            o- (listen -o {run inputChar}).

done2 : done read
            o- pressed_enter  %% test & consume
            o- run print.


run4 : run inputChar
           o- phase P
           o- (init inputChar -o {run read}).

run5 : run print
           o- phase P
           o- (init print -o {entered T * phase print}).
```

```
#trace * {entered (snoc nil d) * key enter * init read}.

#query * * * 10 run readline.
```