Chapter 3

Linear Logic Programming for Narrative Generation

> "Would you tell me, please, which way I ought to go from here?"
> "That depends a good deal on where you want to get to."
> "I don't much care where—"
> "Then it doesn't matter which way you go."
> —- Lewis Carroll, *Alice in Wonderland*

3.1 Introduction

The problem of *narrative generation* is to computationally derive event sequences so that they can be understood as the skeletal structure of a story (the *fabula*). Finding better and more varied approaches to the task concerns studies on computational creativity (in what sense can a computer construct works that are understood as creative?), virtual worlds (how should the world react in response to the player to construct convincing narrative?), and computer-supported education (how can a virtual environment help a human learner build understanding through narrative)? In this chapter, we consider linear logic as a representional system for the task.

Specifically, we propose the use of a *logic programming language* based on linear logic for specifying narrative worlds that generate stories. We extend the narrative representation ideas from Chapter 2 by showing how linear logic specifications may be operationalized (given semantics as a computer program) through proof construction. Proof construction can then be understood as a process that generates narratives when the specification in question represents a story world, or narrative possibility space.

Linear logic programming supports action description, use of resources, and changes imposed on the story world, and offers a basis for analysis of storylines and causal relationships. While similar affordances are found in planning languages, the systems of inference built into planning are usually *goal-directed* in that they are engineered to find *solutions* for getting from one world configuration to another, and that proof procedure itself is not programmer-controllable nor grounded in a theory that informs sound

modification and extension. Planners' goal-driven nature makes them less suitable by default for modeling story scenarios which are exploratory and generative rather than centralized on authorial intent (although variants of planners have been created for this purpose, which we describe in Section 3.6).

Meanwhile, authors, game designers, and scholars have long been seeking and proposing suitable notions of "emergence" in interactive narratives—many talk about wanting to codify richly general behavior, but some talk about the inevitability of unmanagable consequences of doing so, fearing that authorial intent will become impossible to convey. Our inspiration for framing and solving this problem comes from Emily Short's online article on emergence in narrative interaction design.[1]

We carry out our study using the pre-existing linear logic programming language Celf [SNS08], based on the theoretical framework CLF [WCPW03], which offers support for both goal-driven authorial intent and exploratory emergence in narrative generation. Two dual proof construction strategies, called backward and forward chaining, map onto these dual forms of storytelling. During the forward-chaining phases, the program can be thought of as evolving a system forward, as in classic systems of emergence such as cellular automata. In backward chaining, a particular *goal* is specified, which constrains the set of story world rules that are examined. This strategy may be used to impose constraints on the available forward evolutions, as well as to stipulate story endings. Combining forward and backward chaining has been found important for the modeling of duality in agent deliberation [THT12], which is reflected in our modeling of a multi-agent social story world. The proof-theoretic basis of forward and backward chaining [CPP08, HPW00] gives these programs their meaning and relates them to standard logical provability.

Our main result in this chapter is a sizeable example of a story world specified in Celf and its interpretation as a story-generating program. We show evidence that, despite the exploratory emergence created by general inference rules, the causal structure of proofs generated by Celf can assist the author in reasoning about the structure and consequences of emergent stories. This approach to narrative generation lets us design narratives with richly-interacting processes while maintaining the ability to describe goal-based reasoning when needed, as well as enabling the author to reason post-hoc about the space of stories generated.

3.2 Linear Logic Programming

3.2.1 Forward and Backward Chaining

Two dual proof search strategies, *forward* and *backward* chaining, give rise to dual program semantics. Consider a sequent $\Gamma; \Delta \vdash A$ and a fixed context (or *signature*) Σ that can be considered an extension of Γ, containing unchanging rules $A \multimap B$ and unchanging atomic facts. In **backward chaining** proof search, the prover scrutinizes the goal A of the sequent and searches for rules in Σ that lead to a proof of A, then recursively searches

[1] http://emshort.wordpress.com/2008/02/13/emergent-puzzle-solutions/

for proofs of those rules' subgoals. In **forward chaining** proof search, the prover instead scrutinizes the *assumptions* Δ and looks for rules in Σ that might *use* those assumptions to produce new assumptions. Only after this procedure has stabilized (referred as *quiescence*) does the prover then look to the goal A to determine whether it is derivable from the inferred set of assumptions.

In the context of narrative generation, doing backward-chaining search at the top level of a story specification means reasoning first from the desired narrative outcomes, such as *the brick house is the only house left standing* at the end of The Three Little Pigs. A purely backward-chaining approach to solving that story goal would first look at all of the rules whose consequents include $\mathsf{brick\_house}$, at which point it would find $r_3 : \mathsf{pig} \otimes \mathsf{bricks} \multimap \mathsf{brick\_house}$ and $r_6 : \mathsf{wolf} \otimes \mathsf{brick\_house} \multimap \mathsf{brick\_house}$ as candidates, then recursively try to solve for the antecedents of these rules using all of the assumptions in the starting configuration Δ_0. (If any resources are left un-consumed, it must fail.)

On the other hand, forward-chaining search corresponds more to an *exploratory* approach to narrative generation: starting from a world defined by Δ_0, what are the possible narrative actions that might arise? In our example,

$$\Delta_0 = \{\mathsf{pig}, \mathsf{pig}, \mathsf{pig}, \mathsf{straw}, \mathsf{bricks}, \mathsf{sticks}, \mathsf{wolf}\}$$

In forward-chaining proof search, the first rules we would consider would be the three house-building rules, since they are the only ones that apply to our current world state.

The approach we investigate in this chapter will focus primarily on forward chaining, although backward chaining may sometimes be needed to solve for the subgoals of narrative actions. Below we specify how a primarily forward chaining-based search strategy gives rise to a *transition system* interpretation of linear logic specifications.

3.2.2 Logic Programming: Proof Search as Execution

The logical machinery laid out so far tells us what a proof *is*, but does not dictate a particular process for finding one of a given goal sequent. A *logic programming language* is an operationalization of a proof system in such a way that search for a proof may be seen as running a program written in the language. The program itself is a collection of logical propositions, used as assumptions in the goal sequent.

Recall from Section 2.4.4 the derivation of the sequent

$$\Gamma; \mathsf{pig}, \mathsf{pig}, \mathsf{pig}, \mathsf{straw}, \mathsf{bricks}, \mathsf{sticks}, \mathsf{wolf} \vdash \mathsf{brick\_house}$$

where Γ contains these rules:

$$
\begin{aligned}
r_1 &: \mathsf{pig} \otimes \mathsf{straw} \multimap \mathsf{straw\_house} \\
r_2 &: \mathsf{pig} \otimes \mathsf{sticks} \multimap \mathsf{stick\_house} \\
r_3 &: \mathsf{pig} \otimes \mathsf{bricks} \multimap \mathsf{brick\_house} \\
r_4 &: \mathsf{wolf} \otimes \mathsf{straw\_house} \multimap \mathsf{wolf} \\
r_5 &: \mathsf{wolf} \otimes \mathsf{stick\_house} \multimap \mathsf{wolf} \\
r_6 &: \mathsf{wolf} \otimes \mathsf{brick\_house} \multimap \mathsf{brick\_house}
\end{aligned}
$$

The derivation consists mainly of uses of the $\multimap L$ rule on rules $r_i : A \multimap B$ in Γ, which transforms proof states of the form $\Delta, \Delta' \vdash \mathsf{brick\_house}$ into $\Delta, B \vdash \mathsf{brick\_house}$ when $\Delta' \vdash A$. Therefore, instead of the proof tree notation, it is more expedient to represent the derivation as a sequence of *transitions* on just the left-hand side of the sequent that look like

$$\Delta, \Delta' \to^{r_i} \Delta, B$$

where the notation $\Delta \to \Delta'$ can be read as "Δ takes a step to Δ'."

To write the derivation out fully takes much less space than the full proof tree:

$$
\begin{aligned}
& \mathsf{pig}, \mathsf{pig}, \mathsf{pig}, \mathsf{straw}, \mathsf{bricks}, \mathsf{sticks}, \mathsf{wolf} \\
\to^{r_3}\ & \mathsf{brick\_house}, \mathsf{pig}, \mathsf{pig}, \mathsf{sticks}, \mathsf{straw}, \mathsf{wolf} \\
\to^{r_2}\ & \mathsf{stick\_house}, \mathsf{brick\_house}, \mathsf{pig}, \mathsf{straw}, \mathsf{wolf} \\
\to^{r_1}\ & \mathsf{straw\_house}, \mathsf{stick\_house}, \mathsf{brick\_house}, \mathsf{wolf} \\
\to^{r_4}\ & \mathsf{wolf}, \mathsf{stick\_house}, \mathsf{brick\_house} \\
\to^{r_5}\ & \mathsf{wolf}, \mathsf{brick\_house} \\
\to^{r_6}\ & \mathsf{brick\_house}
\end{aligned}
$$

In this fashion, we give an operational semantics to forward-chaining proof search over a story world.

More generally, we can give the whole system of inference rules defining linear logic an operational semantics for a programming language by conceiving of it in terms of a *transition system*, or a set of rules for evolving one program state into another. With forward chaining search, we may do exactly that. Forward chaining proof search for a sequent $\Gamma; \Delta \vdash A$ effectively ignores the goal A, meaning that we can write forward chaining proofs and inference rules with the notation $\Gamma; \Delta \to \Gamma'; \Delta'$, where $\to$ is read as "transitions to" or "takes a step to," whenever it is the case that

$$\frac{\Gamma'; \Delta' \vdash \gamma}{\Gamma; \Delta \vdash \gamma}$$

is a permissible inference for an arbitrary conclusion γ.

3.2.3 Celf

The following is an example of a narrative action specified in Celf:

```
jealousy : eros A B * eros Witness A
          -o {eros A B * eros B A * anger Witness A * anger Witness B}.
```

We first explain the syntax piece-by-piece, then relate its semantics to the sketch of proof search given above.

The token `jealousy` is the name of the rule. (Here "rule" is synonymous with narrative action or scene.) The `:` separates the rule name from the rule. `eros A B` is an atomic proposition referring to two *logic variables*, or parameters to the rule, `A` and `B`. They are indicated as variables by the syntactic convention that they start with capital letters. The third parameter to the rule is `Witness`, referred to in the other part of the antecedent. The `*` symbol is ASCII for $\otimes$, combining the two atomic propositions, and `-o` is ASCII for $\multimap$, indicating a potential rewriting of part of the state matching the left-hand side of the rule to that represented in the right-hand side of the rule. The other atomic propositions should be self explanatory. The braces syntax `{...}` around the consequent designates the rule as *forward chaining*, in a way that we make precise in Appendix A. Finally, the `.` ends the rule.

More semantically, this rule can be read as a transition schema: for any `A`, `B`, and `Witness`, if part of the state matches `eros A B * eros Witness A`, then we may replace that state with the consequent. And finally, adding a story interpretation, the rule says that whenever a witness watches someone they are attracted to exhibit attraction to someone else, the latter attraction becomes reciprocated, and the witness becomes angry at both parties. Such a rule can be imagined as on-theme for a story world designed to give rise to romantic drama.

The other half of a story world needed for an executable artifact is an initial configuration. The story world itself can be considered the program, while the initial configuration is its input. In Celf, we can provide this input to a run of the program via a *trace* directive. The following syntax indicates an execution of the program with a "love triangle" starting configuration:

```
#trace *
  { eros helena hermia * eros hermia helena
  * eros helena lysander * eros lysander hermia }.
```

This directive starts proof search in forward chaining mode with Δ_0 containing just the above tensored-together proposition, and with no specified goal, applying rules in the story world until quiescence (which may never be reached). The `*` just following the `#trace` directive is a bound on forward chaining steps, which can be a number or `*` meaning no bound.

Let us look in detail at one step of forward execution for this program. There is only one rule available, so the engine will attempt to match its antecedent to any combination of resources in the context. To do so, it must answer the question of what *instantiations* are available for the variables `A`, `B`, and `Witness` such that the antecedent holds.

For our given example, some available substitutions are:

```
A := hermia, B := helena, Witness := lysander
A := helena, B := lysander, Witness := hermia
A := lysander, B := hermia, Witness := helena
```

If a suitable instantiation is found, the proof search engine will *substitute* the appropriate terms for the variable in the rule before carrying out its effect. For instance, suppose the substitution selected is the first one. The rule with that substitution applied is:

```
jealousy[hermia, helena, lysander] :
  eros hermia helena * eros lysander hermia
  -o eros hermia helena * eros helena hermia
    * anger lysander hermia * anger lysander helena.
```

This means that the state $\Delta_0 =$

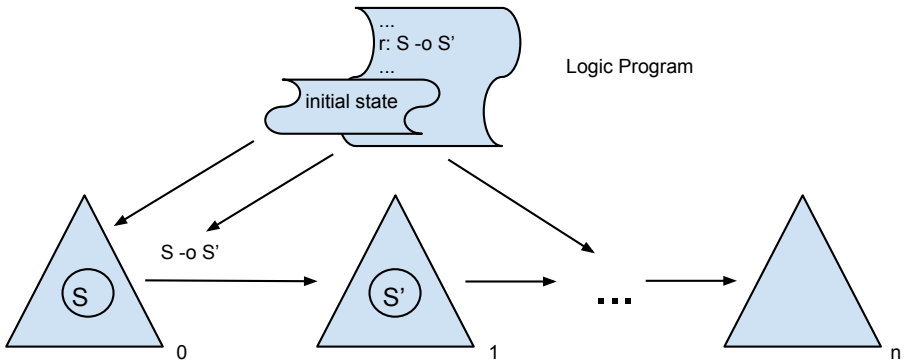
```
{ eros helena hermia, eros hermia helena
  eros helena lysander, eros lysander hermia }
```

may transition to the state

```
{ eros helena hermia, eros helena lysander,
  eros hermia helena, eros helena hermia,
  anger lysander hermia, anger lysander helena }
```

Note that the states we produce consist entirely of predicates over known terms, i.e. they do not contain variables. This conditions is referred to as *groundness*: a term, proposition, or context is *ground* if it contains no logic variables. To be a well-formed story world, we require that initial states are ground and that rules *maintain* groundness, i.e. the right hand side of a rule may only mention logic variables that appear on the left hand side of the rule.

If all story world rules obey that constraint, and if they take a similar form to the rule presented—that is, when all logic variables are instantiated, they have the form $S \multimap S'$, where where S and S' stand for arbitrary tensored-together atoms—and Δ_0 is the starting configuration for the story world, then program execution as an end-to-end process may be visualized as follows:



Here Δ_n is a *quiescent* context, i.e. one to which no rules in the program apply.

Note that there are potentially several substitutions and rules that may apply at any step. Celf's operational interpretation of this fact is *nondeterministic committed choice*,

meaning it selects at random from all available transitions, makes the transition, and does not backtrack. We will revisit this notion of nondeterminism in Section 3.4.

We summarize the linear logic connectives used to create story worlds as well as Celf's notation for them below:

abstract syntax	concrete syntax	meaning
$A \multimap \{B\}$	`A -o {B}`	forward-chaining rule
$A \otimes B$	`A * B`	conjunction of resources
$!A$	`!A`	persistent resource

Finally, a rule whose concrete syntax includes capital-letter logic variables such as `p X Y -o q X` is interpreted abstractly as a first-order logical proposition with the logical variables quantified at the beginning, i.e. $\forall x, y.\ \texttt{p}\ x\ y \multimap \texttt{q}\ x$.

3.3 Example: A Romantic Tragedy Story World

The key to programming with linear logic is formulating one's problem in terms of *resources*, i.e. the components of a story that may interact and change. Then, the author describes how (through what actions) those components change. These two steps may mutually iterate on one another, but at a glance, the creation of a story world as a logic program consists of the following steps:

1. Identify the components of story state, such as physical location and character relationships. Declare a predicate (type) for each of these; for example, declare that `anger C C'` is a well-formed state component when C and C' are characters. We map predicates in our example to their intended meanings in the table below.
2. Identify the *narrative actions*, i.e. ways that characters can interact with each other or their environs to cause changes in the state.
3. Define an initial configuration, or sets of initial configurations for testing. Formulate a trace or query on the program and initial configuration to run it.

A complete story world specification is simply a collection of these predicate and rule declarations. The author may then join it with its complementary half, a specification of setting elements, characters, and initial states, to generate stories or analyze the system we have described. The remainder of this section carries out an example, a case study of a Shakespeare-inspired romantic tragedy world, following the aforementioned workflow.[2]

The state we model includes sentiments between characters (several forms of love and hate), physical locations and basic movement, possession of key objects (such as weapons), relationship states (marriage and being single), desires for objects, and solitary emotions (depression).

[2]The complete, runnable code for this example can be found at `https://github.com/chrisamaphone/interactive-lp/blob/master/examples/tragedy.clf`.

Predicate	Meaning
`at C L`	C is (alive) in location L
`has C O`	C possesses an object O
`neutral C C'`	C feels neutrally toward C'
`philia C C'`	C feels affection toward C'
`anger C C'`	C feels anger toward C'
`eros C C'`	C feels attraction toward C'
`unmarried C`	C is unmarried
`married C C'`	C is married to C'
`depressed C`	C is depressed
`suicidal C`	C is suicidal
`!dead C`	C is dead
`!murdered C C'`	C murdered C'
`!actor C`	C is a character in the story

These predicates do not specify a particular cast of characters or setting, but they could be said to pertain to a particular *genre* of story, in this case romance and tragedy. The delineation of the story world into predicates is an act of careful human design, often iterated with the generation of stories, and the choices made here make all the difference in terms of which actions are possible to write and therefore what shapes the story can take. For instance, the choice to include two kinds of love, `eros` and `philia`, means that we can codify social rules about sexual relationships between characters. Similarly, the choice *not* to include a predicate for a character's gender renders it impossible to enforce heteronormative relationships. We have to make a choice about whether we want to model a realistic description of human behavior (which often contradicts social norms) or enforce social norms by constraining actions with our formal description.

A selection of rules for our chosen genre is given below, starting with the rules for basic social interaction:[3]

```
do/formOpinion/like
: at C L * at C' L *
  neutral C C'
  -o {at C L * at C' L * philia C C'}.

do/formOpinion/dislike
: at C L * at C' L *
  neutral C C'
  -o {at C L * at C' L * anger C C'}.

do/compliment/private
: at C L * at C' L * philia C C'
  -o {at C L * at C' L * philia C C' * philia C' C}.

do/compliment/witnessed
```

[3]The location predicates `at C L` in these rules signify not just location, but also that the character in question is alive and available in the story. The `at` atom is consumed when a character dies.

```
: at C L * at C' L * at Witness L * philia C C' *
  anger Witness C'
  -o {at C L * at C' L * at Witness L * philia C C' *
      anger Witness C' * philia C' C * anger Witness C}.

do/insult/private
: at C L * at C' L * anger C C'
  -o {at C L * at C' L * anger C C' * anger C' C *
      depressed C'}.

do/insult/witnessed
: at C L * at C' L * at Witness L *
  anger C C' * philia Witness C'
  -o {at C L * at C' L * at Witness L *
      anger C C' * philia Witness C' * anger C' C *
      depressed C' * anger Witness C}.

mixed_feelings
: at C L * anger C C' * philia C C' -o {at C L * neutral C C'}.
```

Note that there are two "versions" each of the actions for insulting and complimenting, one that happens "in private" and another that affects a witness in the same location. This encoding signals a weakness: it is unnatural to represent a *broadcast* of an action affecting every character who might be in range, since linear logic primarily codifies *local* state changes. On the other hand, this mechanism's nondeterminism could be argued to reflect the chance involved in whether an action goes noticed.

Next we codify the rules for romantic interaction, which include tranformations between eros and philia as well as flirting, marriage, and divorce:

```
do/fallInLove
: at C L * at C' L' *
  eros C C'
  -o {at C L * at C' L' * eros C C' * philia C C'}.

do/eroticize
: at C L * at C' L' *
  philia C C' * philia C C' * philia C C' * philia C C'
  -o {at C L * at C' L' * philia C C' * eros C C'}.

do/flirt/ok
: at C L * at C' L * eros C C' * unmarried C * unmarried C'
               -o {eros C C' * eros C' C *
                   unmarried C * unmarried C' *
                   at C L * at C' L}.
do/flirt/discreet
: at C L * at C' L * eros C C'
  -o {eros C C' * eros C' C * at C L * at C' L}.
```

```
do/flirt/conflict
: at C L * at C' L * at C'' L *
  eros C C' * eros C'' C
                -o {eros C C' * eros C' C * eros C'' C
                    * anger C'' C' * anger C'' C
                    * at C L * at C' L * at C'' L}.
do/marry
: at C L * at C' L *
  eros C C' * philia C C' *
  eros C' C * philia C' C *
  unmarried C * unmarried C'
  -o {married C C' * married C' C * at C L * at C' L *
      eros C C' * eros C' C * philia C C' * philia C' C }.

do/divorce
: at C L * at C' L' *
  married C C' * married C' C * anger C C' * anger C C'
  -o {anger C C' * anger C' C * unmarried C * unmarried C'
      * at C L * at C' L'}.

do/widow
: married C C' * at C L * dead C'
  -o {unmarried C * at C L}.
```

These rules include the generation of sentiments from a neutral stance, transformations between the two kinds of love `philia` and `eros`, and flirting, which strengthens both kinds of love, but causes anger if witnessed by another paramour. We also include rules that modify the marriages of characters.

Note in particular that rules can have as premises *multiple copies* of a particular resource to represent a greater quantity of a certain sentiment, such as `philia C C'` in the `do/eroticize` rule. In practical terms, repeating a resource more times creates additional dependencies for the rule, and makes its application less likely to appear as an event early in the story.

Next we supply rules governing death and violence:

```
do/murder
: anger C C' * anger C C' * anger C C' * anger C C' *
  at C L * at C' L  * has C weapon
  -o {at C L * !dead C' * !murdered C C' * has C weapon}.

do/becomeSuicidal
: at C L *
  depressed C * depressed C * depressed C * depressed C
  -o {at C L * suicidal C * wants C weapon}.

do/comfort
: at C L * at C' L *
  suicidal C' * philia C C' * philia C' C
```

```
  -o {at C L * at C' L *
      philia C C' * philia C' C * philia C' C}.

do/suicide
: at C L * suicidal C * has C weapon -o {!dead C}.

do/grieve
: at C L * philia C C' * dead C'
  -o {at C L * depressed C * depressed C}.

do/thinkVengefully
: at C L * at Killer L' *
  philia C Dead * murdered Killer Dead
  -o {at C L * at Killer L' * philia C Dead *
      anger C Killer * anger C Killer}.
```

These rules introduce several potential feedback loops between murder and vengeance, suicide, grieving, and depression.

Finally, we have a few actions that can affect possession:

```
do/give
: at C L * at C' L * has C O * wants C' O * philia C C'

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

do/loot
: at C L * dead C' * has C' O * wants C O
  -o {at C L * has C O}.
```

Given this set of rules, we note that the story world is multi-agent in nature—the interactor with such a story doesn't obviously "play" one particular character, and the rules aren't defined as "behaviors" attached to a given agent. They portray the interiority of all characters at once, allowing them to be referenced and changed in combination.

3.3.1 Initial State

After describing the general rules of our Shakespearean tragedy story world, which are parameterized over characters and locations, we can fill in specific elements, such as the characters and setting of Romeo and Juliet, to have a complete and runnable specification.

First we can describe the *persistent* (unchanging, i.e. not linear) facts about the story, in this case just the world map::

```
mon/town : accessible mon_house town.
town/mon : accessible town mon_house.
cap/town : accessible cap_house town.
town/cap : accessible town cap_house.
```

Next, we need to designate the *initial* state of all the linear predicates. It could look something like this:

```
story_start :
init -o { at romeo town * at montague mon_house * at capulet cap_house
  * at mercutio town * at nurse cap_house * at juliet town
  * at tybalt town * at apothecary town

  * has tybalt weapon * has romeo weapon * has apothecary weapon

  * unmarried romeo * unmarried juliet
  * unmarried nurse * unmarried mercutio * unmarried tybalt
  * unmarried apothecary

  * anger montague capulet * anger capulet montague
  * anger tybalt romeo * anger capulet romeo * anger montague tybalt

  * philia mercutio romeo * philia romeo mercutio
  * philia montague romeo * philia capulet juliet
  * philia juliet nurse * philia nurse juliet

  * neutral nurse romeo
  * neutral mercutio juliet * neutral juliet mercutio
  * neutral apothecary nurse * neutral nurse apothecary}.
```

The first two groups of atoms describe the story world locations where the characters begin and their possessions. The next group represents which characters are unmarried at the start of the story. The next two groups represent existing relationships (sentiments) among characters, and finally the last group represents which characters haven't met each other yet (and so feel neutrally towards each other).

3.3.2 Program Query

We have seen one way to execute the code, namely traces, but if we specify some particular goal conditions (story endings), then Celf will produce a full proof for us, which is necessary for the kind of analysis we will do next.

Here is one way to specify a few possible endings that will drive the simulation to termination:

```
ending_1 % a marriage and a death
: nonfinal *
  at C1 _ * at C2 _ * at C3 _ *
  married C1 C2 * dead C3
  -o {final}.

ending_2 % love triangle
: nonfinal *
  at C1 _ * at C2 _ * at C3 _ *
```

```
    eros C1 C2 * eros C2 C3 * eros C3 C1
    -o {final}.

  ending_3 % vengeance
  : nonfinal *
    at C1 _ * at C2 _ * at C3 _ *
    murdered C1 C2 * philia C3 C2 * murdered C3 C1
    -o {final}.
```

Additionally, we add `nonfinal` to our starting configuration. Now we can initiate a *query* such as

```
?- init -o {final}.
```

This query asks whether there is a *proof* from the state described by `init` to the atom `final`.[4] Proof search starts in backward chaining mode, breaking down the linear implication via the $\multimap R$ rule: it adds `init` to the current state and considers how to prove the goal `{final}`. The modality indicated with curly-braces causes proof search to switch to a *forward-chaining* or *generative* mode, running inference forward from the `init` atom. Only once the entire story has terminated will it look for `final`, at which point search will succeed if it finds it.

But the point of executing the query isn't really to find out whether the initial state leads to a valid conclusion. What we are interested in is the *trace* generated by execution—the witness to the validity of the proposition, i.e. the proof!

3.4 (Forward-Chaining) Proofs as (Causally-Structured) Stories

We have already seen in Chapter 2 how a proof may represent a story: the right-hand side of the goal sequent may be seen as the story outcome, and different rules that are applied in the derivation can be read (bottom-up) as a sequence of narrative actions. However, *forward chaining* proofs have the additional property that the rules they select occur in narrative-temporal order, and they also create a relaxed ordering on those events that reflects causality.

To understand how forward-chaining proofs model causality, let us return briefly to the Three Little Pigs example, which is small enough to describe and comprehend in full. In Section 3.2.1, we gave a sequence of transitions corresponding to forward-chaining rule applications on the story world. This time, let's *name* the elements of the context so that we can better track how they evolve:

[4]Technically, the state at quiescence needs to contain *only* the `final` atom in order for the proof to succeed—the consequent of the query needs to precisely describe the shape of the expected outcome. However, in our actual implementation, we have used another connective in Celf (`@`) similar to `!` that marks a resource as *affine*, meaning we are not forced to consume it (but we still cannot duplicate it). All story resources other than "final" and "nonfinal" are marked as affine. See Schack-Nielsen [SN09] for discussion of affine propositions in CLF.

$$x_1 : \mathsf{pig}, x_2 : \mathsf{pig}, x_3 : \mathsf{pig}, x_4 : \mathsf{straw}, x_5 : \mathsf{bricks}, x_6 : \mathsf{sticks}, x_7 : \mathsf{wolf}$$

We will also make up new names whenever some new resource is added to the context.

$$\begin{array}{ll}
 & x_1 : \mathsf{pig}, x_2 : \mathsf{pig}, x_3 : \mathsf{pig}, x_4 : \mathsf{straw}, x_5 : \mathsf{bricks}, x_6 : \mathsf{sticks}, x_7 : \mathsf{wolf} \\
\rightarrow^{r_3} & x_8 : \mathsf{brick_house}, x_2 : \mathsf{pig}, x_3 : \mathsf{pig}, x_4 : \mathsf{straw}, x_6 : \mathsf{sticks}, x_7 : \mathsf{wolf} \\
\rightarrow^{r_2} & x_9 : \mathsf{stick_house}, x_8 : \mathsf{brick_house}, x_3 : \mathsf{pig}, x_4 : \mathsf{straw}, x_7 : \mathsf{wolf} \\
\rightarrow^{r_1} & x_{10} : \mathsf{straw_house}, x_9 : \mathsf{stick_house}, x_8 : \mathsf{brick_house}, x_7 : \mathsf{wolf} \\
\rightarrow^{r_4} & x_{11} : \mathsf{wolf}, x_9 : \mathsf{stick_house}, x_8 : \mathsf{brick_house} \\
\rightarrow^{r_5} & x_{12} : \mathsf{wolf}, x_8 : \mathsf{brick_house} \\
\rightarrow^{r_6} & x_{13} : \mathsf{brick_house}
\end{array}$$

This holistic view of the state change does not give us a very precise picture of which resources in the context are being removed and which are added by the rule applied. Instead of labeling each transition arrow with just a rule, we can also mark the rule with the names of the resources it uses, and the new names it generates for the resources it adds. In other words, every transition can be written as a *binding*

$$\textbf{let } \langle x'_1, \ldots, x'_n \rangle = r_i \; x_1 \; \ldots \; x_m \textbf{ in } \ldots$$

where the x'_is are fresh variables introduced by applying the transition, and the x_is are the names of resources used by the rule.

In general, the right-hand side of the binding may take some form other than a rule applied to a bunch of variables: for one thing, the arguments to the rule could be constants from the signature, or other terms representing backward-chaining proofs, that are needed as inputs to the rule. The full language of proof terms is described in prior work [WCPW03], and is out of scope for the particular applications of this thesis, but we will acknowledge the possible generality by referring to arbitrary right-hand sides of these let-bindings as *atomic proof terms* R.

If we extrapolate the binding notation across the whole transition sequence above (this time ignoring the holistic representation of the state), we get

$$\begin{array}{rcl}
\textbf{let } \langle x_8 \rangle & = & r_3 \; x_1 \; x_5 \textbf{ in} \\
\textbf{let } \langle x_9 \rangle & = & r_2 \; x_2 \; x_6 \textbf{ in} \\
\textbf{let } \langle x_{10} \rangle & = & r_1 \; x_3 \; x_4 \textbf{ in} \\
\textbf{let } \langle x_{11} \rangle & = & r_4 \; x_7 \; x_{10} \textbf{ in} \\
\textbf{let } \langle x_{12} \rangle & = & r_5 \; x_{11} \; x_9 \textbf{ in} \\
\textbf{let } \langle x_{13} \rangle & = & r_6 \; x_{12} \; x_8 \textbf{ in} \\
 & & x_{13}
\end{array}$$

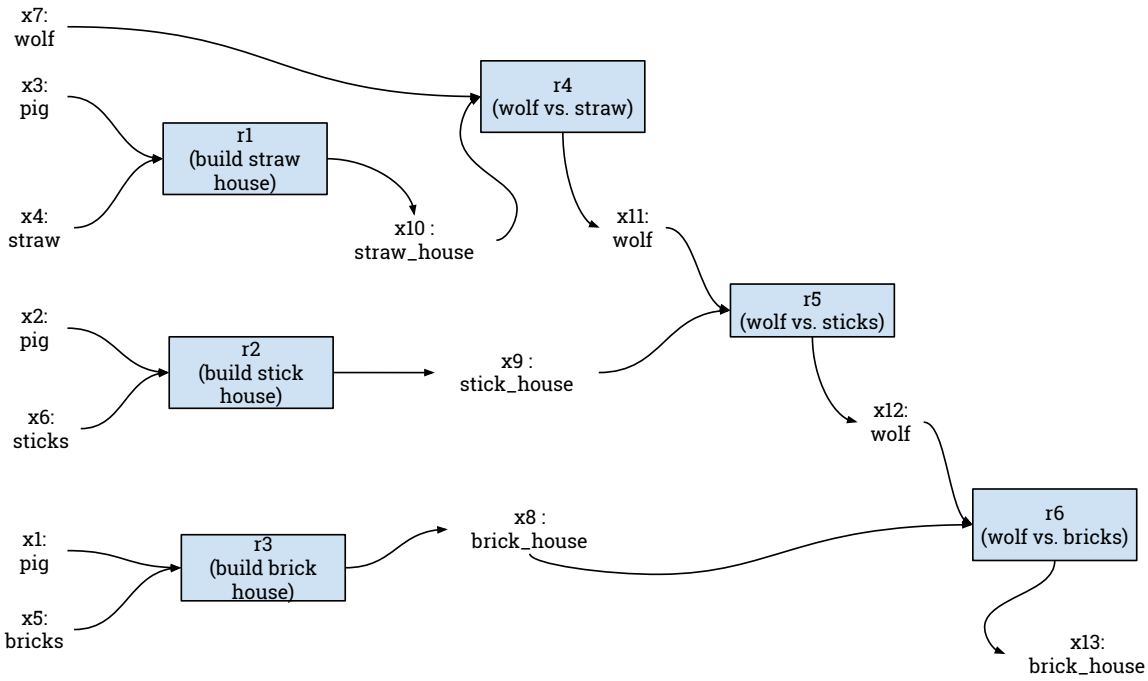
What this representation gives us is a way of describing dependency, and more importantly, *independence* between narrative actions. To see how, let us define an ϵ as a sequence of let bindings representing forward chaining proofs. A trace ϵ can either be empty ($\langle\rangle$), a single binding **let** $p = R$, or one trace after another $\epsilon_1; \epsilon_2$, such that ; is associative and $\langle\rangle$ is its unit, i.e.

$$\begin{array}{rcl} \langle\rangle; \epsilon = & \epsilon & = \epsilon; \langle\rangle \\ (\epsilon_1; \epsilon_2); \epsilon_3 & = & \epsilon_1; (\epsilon_2; \epsilon_3) \end{array}$$

Epsilons that are independent of one another can be characterized by *interchangability*: they can occur in either order, i.e. $\epsilon_1; \epsilon_2 = \epsilon_2; \epsilon_1$. Such an equation is permissible in terms of provability exactly when no variable introduced by one ϵ is used in the other and vice versa. To make this notion of independence precise, we define the pre-set $\bullet(\epsilon)$ and post-set $(\epsilon)\bullet$ of variables for a given proof term. We write $\mathsf{fv}(R)$ for the free variables (input resources) of an atomic proof term R and $\mathsf{bv}(p)$ for the bound variables (output resources) of a pattern p.

$$\begin{array}{rcl} \bullet(\textbf{let}\ p = R) & = & \mathsf{fv}(R) \\ (\textbf{let}\ p = R)\bullet & = & \mathsf{bv}(p) \\ \bullet(\epsilon_1; \epsilon_2) & = & \bullet(\epsilon_1) \cup (\bullet(\epsilon_2) - (\epsilon_1)\bullet) \\ (\epsilon_1; \epsilon_2)\bullet & = & ((\epsilon_1)\bullet - \bullet(\epsilon_2)) \cup (\epsilon_2)\bullet \\ \bullet(\langle\rangle) & = & \{\} \\ (\langle\rangle)\bullet & = & \{\} \end{array}$$

Mapping these notions to a visual representation, if we draw rule names as nodes and draw a directed edge between nodes n_1 and n_2 for each variable $x \in \bullet(n_2) \cap (n_1)\bullet$, we extract exactly the simultaneous (or *concurrent*) structure of the Three Little Pigs narrative shown in Chapter 2:



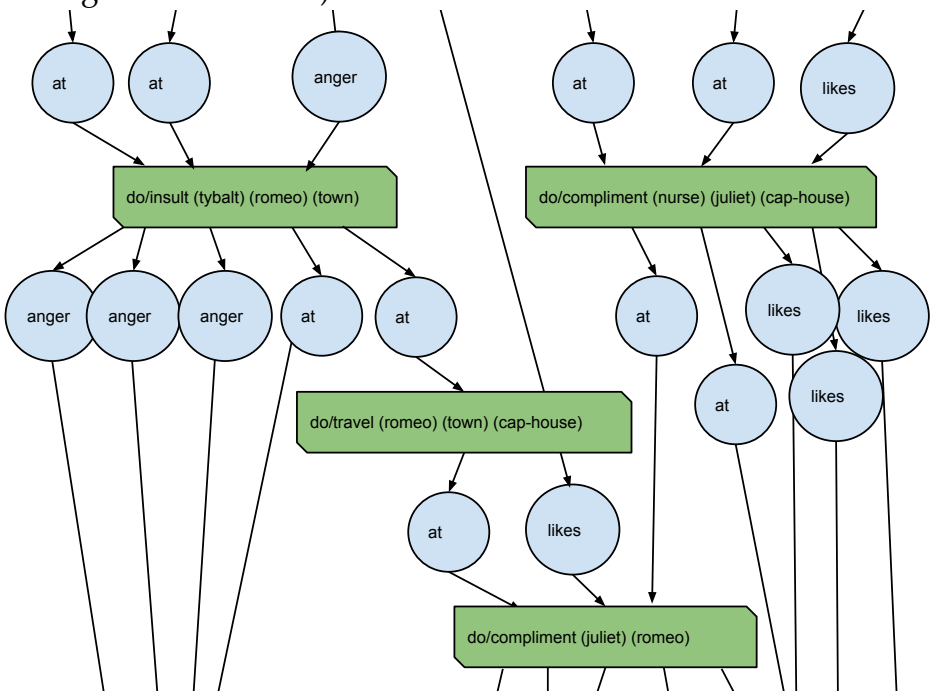
Here we can see that the narrative actions representing the pigs building their houses are independent, and thus the order in which they are told in the story is arbitrary. However, the resources representing the wolf (x_7, $x_1 1$, and $x_1 2$) are threaded through every interaction between house and wolf, so those actions are strictly ordered. In terms of causality, we can say for instance that the wolf having successfully blown down the straw house, and one of the pigs having built the stick house, *cause* the narrative action of the wolf blowing down the stick house: those two transitions r_4 and r_2 generate resources x_{11} and x_9 that are consumed by the transition r_5.

Now, returning the the Shakespearean tragedy story world, we can understand Celf's output for the query `init -o {final}`. One possible proof output generated by search contains the following trace fragment:

```
...
let {[X73, [X74, [X75, [X76, X77]]]]}
  = do/insult/private [a-tybalt, [a-romeo, [X68, [X66, X72]]]] in
let {[X85, [X86, X87]]}
  = do/becomeSuicidal [a-romeo, [X79, [X41, [X59, [X52, X77]]]]] in
let {[X88, [X89, [X90, [X91, X92]]]]}
  = do/comfort [a-mercutio, [a-romeo,
      [X78, [X85, [X86, [X81, X83]]]]]] in
let {[X101, [!X102, [!X103, X104]]]}
  = do/murder
```

```
    [a-romeo, [a-tybalt,
      [X58, [X40, [X76, [X51, [X94, [X96, X27]]]]]]] in
let {[X105, [X106, [X107, X108]]]}
  = do/compliment/private
    [a-nurse, [a-juliet, [X46, [X47, X30]]]] in
let {[X109, [X110, [X111, X112]]]}
  = do/compliment/private
    [a-juliet, [a-nurse, [X106, [X105, X108]]]] in
let {[X113, X114]}
  = do/loot [a-romeo, [a-tybalt, [X101, [X102, [X26, X87]]]]] in
...
```

This trace shows an interleaving of story scenes: there is one thread (or sequence of dependent actions) wherein Tybalt nearly drives Romeo to suicide, but he is comforted by Mercutio, then murders Tybalt; there is another wherein a loving conversation between the Nurse and Juliet occurs. Because the resources involved in each of these subtraces do not overlap, though, they can be seen as simultaneous, and depicted as follows (omitting variable names):

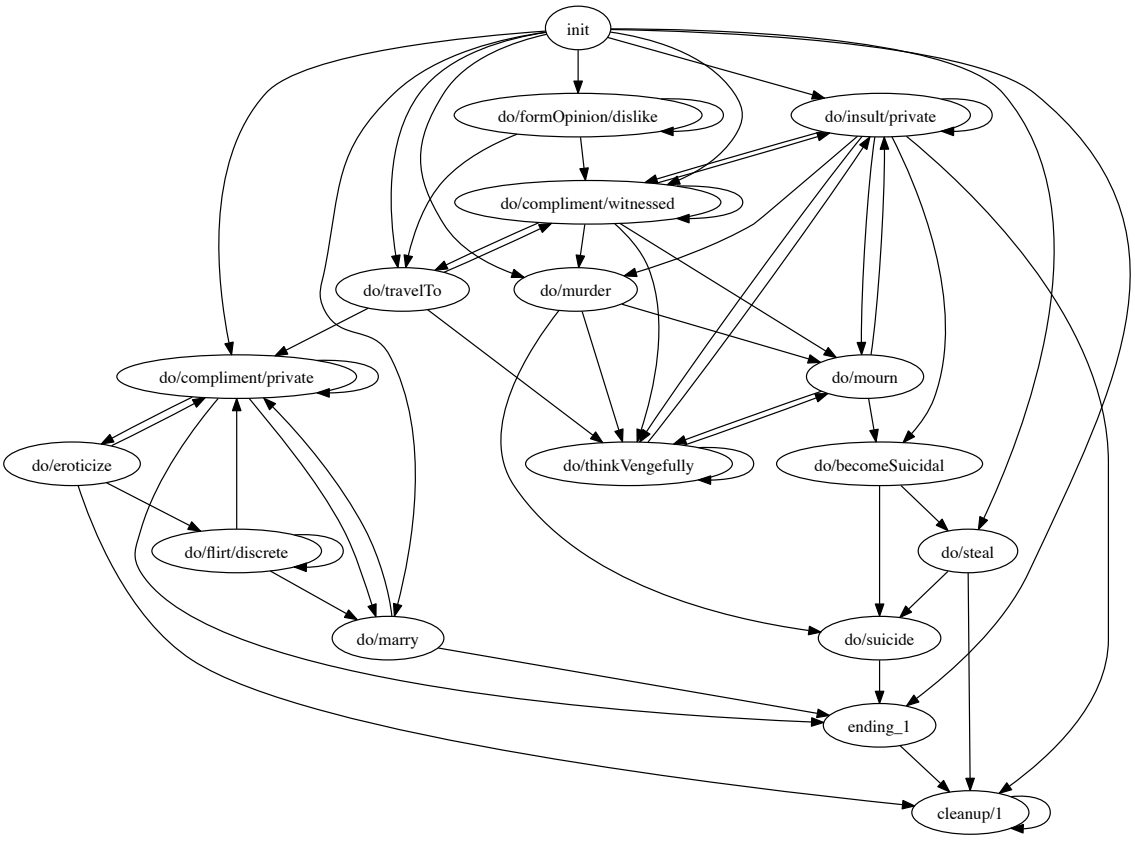


3.4.1 Automated Causal Graph Production

For the sake of making this dependency structure available to the author, one of our collaborators (Joao Ferreria) built a tool called *CelfToGraph*, part of the TeLLer suite,[5] that automatically translates generated stories into causal diagrams in the form of directed graphs. Nodes of the graph are actions in the story, and edges can be viewed as causal relationships between story events involving those actions.

[5] Available at `https://github.com/jff/TeLLer`

This tool takes a proof term and the story specification as inputs, and it generates the extrapolation of the let-binding illustration given in the previous section to the whole trace, unifying each instance of repeated story actions. We wind up with an approximate causal network of action nodes representing the story's nonlinear progression of events, e.g.:



The tool also has an interface for *queries* on sets of traces. E.g. the query `exists ending_1` would tell us the set of stories with `ending_1` (a marriage and a death). This tool allows us to test our specifications for how closely they match our authorial intent. For instance, if we want to find out if any stories contain unfulfilled vengeance, we might issue the query

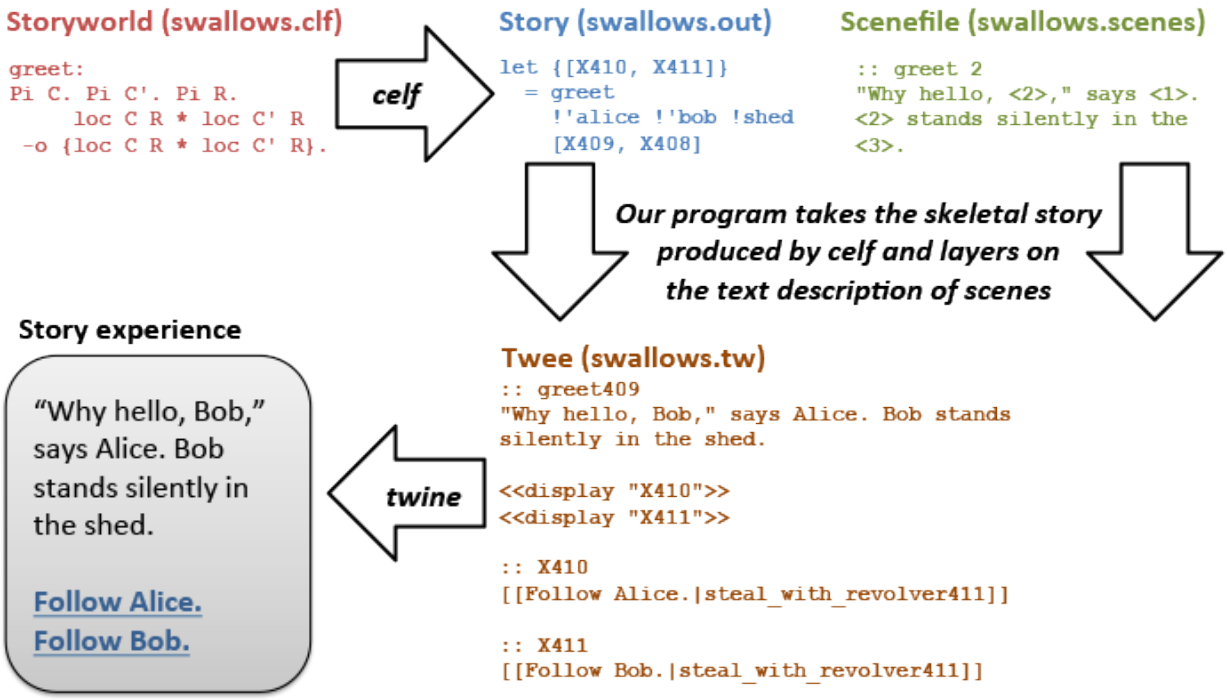
```
exists do/thinkVengefully && ~link do/thinkVengefully do/murder
```

(read as: there exists a "think vengefully" action, but there is no link between "think vengefully" and "murder") which might tell us that no stories satisfy the predicate. If we intend for vengeance to occur more or less frequently, we may tune the parameters of the rules (e.g. how many `anger` atoms it generates) and run the query again.

3.5 Playable Traces

Another application of CLF proof terms is to interpret their dependency structure as branching choice in a playable narrative: in other words, to re-interpret *simultaneous* structure (multiple characters acting in simultaneous scenes) as *alternative* structure (the choice of which character to follow). This transformation has the effect of choices in the narrative representing the opportunity to "follow" a resource in the logic program to the next place where it is used. If that resource corresponds to a character location, then the resulting artifact is a simultaneous story through which the player may choose an arbitrary path by following characters from scene to scene.

We implemented this idea as a compiler from Celf to Twine, with the tool architecture depicted below (*Swallows* is the name of a different example story world):



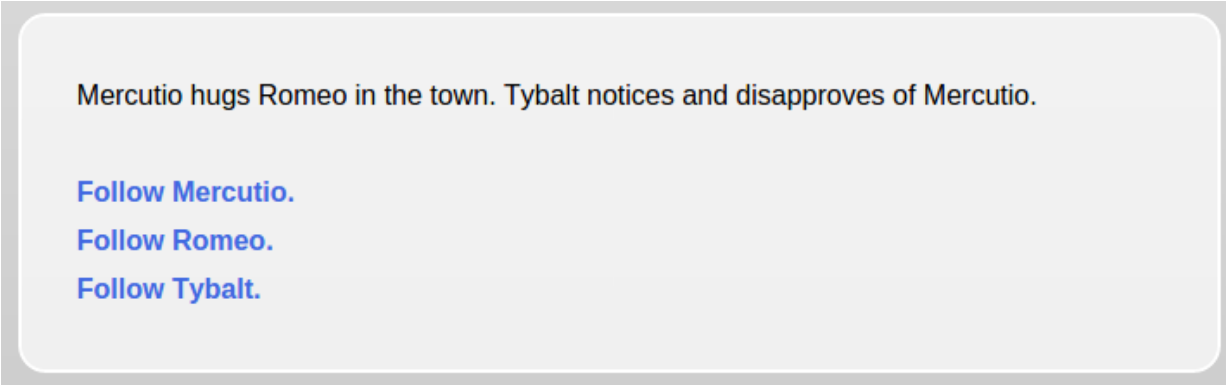
The story world author provides a *scene file* in addition to the Celf program that will generate the simultaneous narrative; the scene file provides the text to go along with the abstract story specifications. For example, here is an excerpt of the scene file for our Shakespearean tragedy world:

```
:: do/flirt/conflict 3
<1> flirts with <2> in the <4>. <3> notices and becomes jealous.

:: do/formOpinion/dislike 2
<1> looks at <2> from across the <3> and irrationally feels disgusted.
||
<1> frowns and looks at <2> in the <3>, thinking "I don't think I like <2>
very much."
```

```
:: do/compliment/witnessed 3
<1> smiles warmly at <2> in the <4>. "You look great today!" <3> frowns at
them both.
||
<1> hugs <2> in the <4>. <3> notices and disapproves of <1>.
||
<1> drapes an arm around <2>'s shoulders in the <4>, saying "You're so great!"
<3> scoffs mocklingly, making a mental note not to trust <1>.

:: do/murder 1
<1> murders <2> in the <3>.

:: do/becomeSuicidal 1
<1> collapses on the ground in the <2>. They feel despairing and hopeless.
An urge to die grows within them.

:: ending_1 1
<1> and <2> lived happily ever after, while <3>'s body rotted.

:: ending_2 1
<1>'s passion for <2> was exceeded only by <2>'s lust for <3>.
Meanwhile, <3> just wished <1> would pay them some attention.

:: ending_3 1
<3> avenged their beloved <2> by murdering <1>.
```

The syntax of these rules is as follows: first, the rule in the corresponding CLF file is named, with a number after it indicating how many of the atoms in its consequent represent *followable resources* (e.g. character locations). The numbers in `<brackets>` are placeholders for the logic variables in the rule, and will be filled in with things like character and room names. The separator `||` indicates alternatives; one of each set of alternatives will be selected for the generated Twine passage. Special `:: initial` and `:: final` scenes can also be given text.

Such a scene file together with a particular narrative generated by Celf can then be played as a Twine game at `http://play.typesafety.net/world/shakespeare/`. A sample passage for the rule `do/compliment/witnessed` is shown below:



The *Quiescent Theater* project hosts a collection of story worlds like this one at `http://play.typesafety.net`. It re-runs the Celf programs every hour to generate new story variations, which can then be played on the website.

In Chapter 5, we give a case study of this playable story generation technique for a theater piece, *Tamara*, whose original script has a mechanic of simultaneity—action taking place in several different rooms in the performance space—that makes it amenable to this technique.

3.6 Related Work

Our contribution with this work has been to investigate linear logic programming as a modeling language for story worlds such that proof search generates stories. This approach lets us model story scenarios which are primarily exploratory and generative, rather than purely goal-driven, using forward-chaining proof search. Because we are bridging ideas in computational logic with ideas in narrative generation, we identify related work in each of those categories.

The duality between intent (backward chaining) and exploration (forward chaining) has previously been used for combining deliberative and reactive behavior for agent modelling [HW04]. The Lygon language preceded those ideas and incorporates both forward- and backward-chaining linear logic programs [HPW96]. Forum [Mil96] is a linear-logic based specification language for describing concurrency, which was later investigated for use as a logic programming language [HP96]. All of these works influenced the development of LolliMon [LPPW05], the direct predecessor to CLF and Celf. Celf, due to its basis in proof search, supports both forward and backward chaining through the *focusing* theory of proof search [CPP08], which has been central to its theoretical underpinnings and practical implementation.

In narrative generation, Tale-spin [Mee76] is a canonical work, which generated stories though comprehensive simulation of worlds, characters, and properties, specified using a Common Lisp-based system. Its approach to story generation has many similarities to the forward-chaining linear logic programming approach in that narrative actions are selected on the basis of their applicability rather than any kind of unifying narrative intent. However, no connection to logical provability or goal-driven search is present.

Later, Riedl and Young develop the approach of using AI planners to generate narratives with a better balance of character-driven exploration and author-driven story goals [RY04, RY10], alongside the Oz Project [Mat99]'s *reactive planning* approach to combining hand-authored scenes in response to real-time player actions [MS03]. Porteous and Cavazza [PC09] address similar problems using constraints on plan trajectories.

Currently, the approach to using a linear logic programming language as a narrative generation engine has many of the problems that this later work on planning addresses: that is, it still takes very careful authoring to ensure that narrative actions are meaningful and interesting within the larger context of story. However, this is the first work to provide such "emergent narratives" with a formal basis in logical inference, and by do-

ing so we speculate that extending these planning techniques for controlling story input to our setting will be an easy and fruitful vector for future work. Meanwhile, the logical foundation of this setting provides a meaningful framework for analyzing causality, validating scenarios [DCA13], and extending the formalism with other notions such as knowledge and belief [GBB+06].

3.7 Conclusion

We built atop the groundwork laid in Chapter 2 on representing narrative structure in linear logic by using the Celf logic programming language to encode models for a "Shakespearean tragedy" story world, and we showed how proof search algorithmically generates stories from such a model [MBFC13, MFB14]. We presented a theory of dependency between partial proof traces that maps onto the idea of narrative simultaneity, and we showed two different ways of exploring that partial order structure: the *CelfToGraph* visualization tool and the Quiescent Theater Twine game generator.

In the next chapter, we introduce a new linear logic programming language augmented with facilities for interaction and separation of programs into *stages* for programmer convenience, transitioning our application domain from narrative generation to *interactive* narrative and game prototyping.