Programming Interactive Worlds with Linear Logic

Chris Martens

CMU-CS-15-134
September 2015

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Frank Pfenning, Co-chair
Karl Crary, Co-chair
André Platzer
Roger Dannenberg
Anne-Gwenn Bosser

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation under grant numbers CCF-0546550, CNS-0716469, DGE-0750271; Qatar National Research Fund under grant number NPRP-096671100; and Commission of the European Communities under grant number FP-6506909. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Programming Languages, Computational Logic, Game Design, Interactive Simulation, Interactive Storytelling, Interactive Fiction

Abstract

Interactive storytelling weaves together deep computational ideas with humanity's rich history of story and play, providing an important context for tools and languages to be built. At the same time, formal specification languages offer a palette of representation and inference techniques typically reserved for the analysis of programming languages and complex deductive systems. This thesis connects problems in the interactive storytelling domain to solutions in formal specification.

Specifically, we examine narrative from a structural point of view and observe that alternative narrative paths play a complementary role to simultaneous interacting timelines. *Linear logic* provides the representational tools necessary to investigate this structure, and by extending the correspondence to proofs and proof construction, we find a suite of computational possibilities. We present three efforts toward realizing those possibilities: (1) the use of linear logic programming to *generate* narratives; (2) a new programming language for authoring *interactive* narratives, games, and simulations; and (3) techniques for stating and proving design-level program properties.

We find that linear logic programming, enriched with a minimal extension to its logical semantics, enables a wide range of programming idioms and domain encodings. As evidence, we give five case studies, including social simulation, combat-based adventure games, and board games. To support reasoning about design correctness, we present techniques for stating and proving program invariants, as well as a decidability proof for automatically checking those invariants for a large fragment of the language.

These findings show that linear logic is a fruitful representation language to serve as the basis for modeling and executing interactive worlds, and they invite future investigations on using proof-theoretic methodologies for creative systems.

Acknowledgments

I am tremendously privileged to have been able to do a Ph.D. thesis on the beautiful subjects of logic and game design, and a tremendous number of people contributed toward my ability to do this strange and wonderful thing.

My advisors Frank and Karl have been instrumental every step of the way: from supporting my case for admission to CMU's graduate program, to training me in the foundations of my field, to keeping their minds open as I began to explore these highly unconventional applications of said foundations. Bob Harper's lectures and writings have also been a crucial part of my programming languages education.

Gwenn Bosser's work on linear logic for narrative structure gave me the solid stepping-stone I needed to bring this thesis to fruition, and her collaboration and encouragement has been critical for sustaining my optimism around the project. I also thank David Renshaw for pointing out her papers to me in the first place.

My other collaborators in papers, programs, and creative media, have given me so much energy and inspiration: João Ferreira, William Lovas, Rob Simmons, Vincent Zeng, Elizabeth Davis, Will Byrd, Claire Alvis, Philip Gianfortoni, Ben Blum, Lea Albaugh, Jamie Perconti, Tom Murphy VII, Salil Joshi, Andrew Cave, Kaustuv Chaudhuri, Dale Miller, Daniel Spoonhower: I thank you all for your patience with me as an often-fickle collaborative partner, and for the joy you've shared with me in what we created.

I thank the many attentive readers of this work (and its predecessors), including, of course, my thesis committee members—André and Roger in particular have contributed insightful advice on accessibility to audiences outside of logic. Other readers I thank include William Cohen, Nico Feltman, Adam Smith, Joe Osborn, Jurie Horneman, Paul Mazaitis, and several anonymous reviewers, whose feedback has invariably strengthened my understanding of prior work, communication of my own work, and inspiration for future work.

The CMU POP/ConcertRG student group has provided valuable discussion of papers, talks, and their own research, as well as attentive feedback on my own talks and papers. In particular, my classmates Arbob Ahmed and Henry DeYoung have motivated me with their fascinating research and deep analytical insights, and Ian Voysey provided incredibly thorough and helpful feedback on practice talks. The crew that graduated early in my time in grad school—Neel Krishnaswami, Noam Zeilberger, Jason Reed, William

Lovas, Rob Simmons, and Dan Licata—set an admirable tone for humble and patient research discussions that I tried to emulate after they graduated (it's harder than it looks). Current students like Nico Feltman, Joe Tassarotti, and Anna Gommerstadt have spiritually succeeded them with the proactive collaboration and careful thoughtfulness they bring to the group—I can't wait to see what they do next.

Ron Garcia, Ed Morehouse, and Stephanie Balzer, during their postdocs at CMU, have all provided valuable mentorship and research discussions. Gillian Smith and Jessica Hammer, although my relationships with both of them are relatively new, have been immensely helpful in me considering my career trajectory and imagining the sort of professor I might be someday. I feel incredibly lucky to have benefited from the mentorship of these wonderful people.

Will Byrd approached me at OPLSS in 2010 and, out of the blue, asked if I "knew anything about interactive fiction" because I came from CMU like (renowned IF author) Andrew Plotkin. We then shared hunches about logical formalisms for IF and put together a paper for OBT about it—long before I pursued the possibility of a thesis about the idea. Will has since been a wonderful collaborator and mentor, emailing me unprompted and frequently with opportunities, suggestions, and encouragement.

Independently-created games by Emily Short, Porpentine, Stephen Lavelle, Terry Cavanaugh, Squinky, Lydia Neon, Tom 7, Sam Barlow, Andrew Plotkin, Andi McClure, Michael Brough, Aaron Reed, and Merritt Kopas, have provided a great deal of creative inspiration for this thesis work. My students in 98-228 Interactive Fiction helped me, too, with their appreciation and insight for the works I assigned them to play, and the wonderful games they turned in as final assignments.

The UC Santa Cruz EIS lab, and the INT and FDG conference attendees, have helped me understand what games research can be, and that it is full of wonderful people. Conversations, papers, and talks by Kate Compton, Gillian Smith, Squinky, Clara Fernandez-Vara, Joe Osborn, James Ryan, Aaron Reed, Jacob Garbe, April Grow, Michael Mateas, Noah Wardrip-Fruin, Ian Horswill, Gail Carmichael, R. Michael Young, and Ben Kybartas, have been crucial to the joy I find in games and creativity research.

Back at home in Pittsburgh, there are a few who've stuck with me across the years: Lea Albaugh, who met me 11 years ago and lived with me for 6, patiently taught me things about Inform 7, sewing, soldering, and intersectional feminism. Lea's thumb on the pulse of local art and theater projects is also what led me to see the play *Tamara*, the study of which has been my favorite application of this thesis work. William Lovas, who persuaded me during one pivotal bike ride in 2012 to pursue my "linear logic for games" idea, despite my doubts that my advisor would take it seriously, has been there for me in countless ways, over countless drinks and meals, in countless cities across the world.

There have been several people in my life who pushed me to meet a higher bar of excellence than I was meeting before their intervention, but the first person to push me beyond what I thought I was capable of was Jamie Morgenstern, all but dragging me across the finish line of my first 5k race in 2010 to finish in under 26 minutes. She then proceeded to achieve elite status in her own running career and finish her Ph.D. in under 5 years, providing an inspirational (if unattainable) watermark for achievement.

For someone like me who's often timid about face-to-face interaction, internet communication channels are absolutely priceless, and the friendships I've made and/or maintained through LiveJournal and Twitter deserve just as much recognition as the few I've grown close to in physical space. Kat Hagan, Lindsey Kuper, Cass Sparks, Jason Reed, Danielle Kefford, Greg Hanneman, Katie Mazaitis, Tim Chevalier, Joshua Dunfield, Mym Johnson, Akiva Leffert, Annie Ogborn: your words have helped me in so, so many ways. And then there's Philip "8" Gianfortoni, whose weekly video calls are the only form of regular long-distance, voice-based contact I've ever maintained, and I'm so grateful for his part in sustaining them.

There's still an unfortunate stigma against addressing mental health in academia, and I'm going to try to do my part to break it here: I would unquestionably not have finished my Ph.D. without therapy for anxiety and depression. Thanks to Dr. W for her incredible skills.

I also would not have been able to afford my undergraduate education at CMU (an undeniably critical stepping stone to grad school there) without almost full grant support from the US Department of Education, for which I am immensely grateful. Furthermore, I doubt I would have been admitted in the first place without the groundbreaking work of Allan Fisher and Jane Margolis, 20 years ago, to identify and excise the irrelevant gender-correlated criteria previously associated with computer science talent in CMU's undergraduate admissions.

My paternal grandmother died in 2013, after she gave me 27 years of unwavering love; a brilliant, hilarious spirit as a role model; and stories and artifacts pertaining to my father, who died a few months after my birth. I only wish she could be here to see me graduate.

My mother has been the one constant in my life, and she has made her love known the whole way. Growing up, my "village" consisted of grandparents, aunts and uncles, teachers, and my mother's romantic partners, all of whose love and guidance deserve my profound thanks. But my mother shaped who I am today more than anyone else; it's hard to thank her without a touch of narcissism. Mom: thanks for showing me how to be fearless, opinionated, independent, creative, hard-working, vulnerable, selfless and selfish (as the situation warrants).

Finally, while it is clearly *possible* to be a successful academic without a partner who shoulders a large burden of domestic and emotional labor, it is abundantly clear to me that Rob's efforts to be an equitable and generous partner to me while juggling his own teaching career have made a huge difference in my productivity, creativity, and happiness. And furthermore, I am in the rare position to have a partner who understands and enjoys my thesis work, even to the point of putting up with my rambling about narrative structure after every television show, movie, and play we see together.

Thanks, everyone, for your patience, insight, love, and support.

Contents

List of Figures

Chapter 1

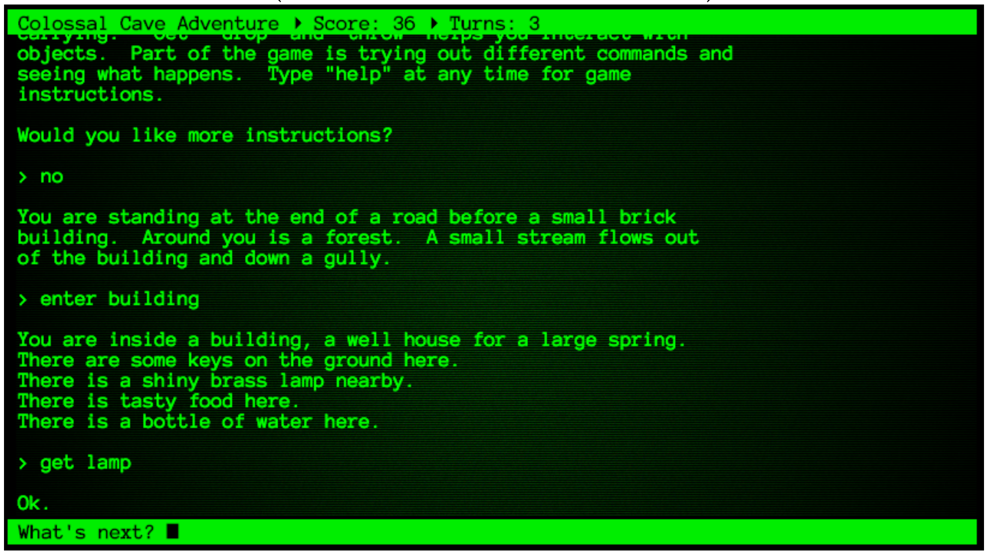
Introduction

The two activities of *playing games* and *telling stories* have shaped the way humans learn and bond with one another since at least the dawn of history. From a humanistic perspective, they are as central to our existence as language [Hui14]. They have also never been entirely distinct, with each drawing on the human ability to invent fictitious worlds, defined by rules and aesthetics originating from people's minds without persistence of truth outside their "magic circles."

Only in a comparatively tiny, recent sliver of humanity's timeline have *digital computers* entered into a place of import for the enhancement of these activities, but they have brought with them an explosion of new thought on how to design and construct deliberate playful, narrative experiences—new magic circles, as it were. One of the central ideas in this rennaissance is that computers enable much richer *system-driven interaction* than we were previously capable of creating. All games, digital or not, form a system, or set of rules that interact with one another nontrivially. For example, the game of Chess has rules for moving pieces of different shapes in patterns that are constrained by rules that depend on the positions of the other pieces. These rules are easy enough for a human to carry out by hand. But thanks to the speed of digital processing, we can now describe highly complex systems whose effects require calculation that would be too tedious to compute by hand while maintaining the spirit of play, such as the trajectories of particles bouncing around in simulated space.

One of the earliest forms of digital game was the *text adventure*, a kind of story told through interaction with a text-based command-line interface to navigate a fictional world. Commands like `go north` and `get lamp` could be typed and translated into changes in the game's state, which would be reflected in changes to the prose describing the player's surroundings, preceding the next prompt for input. (See Figure 1.1 for a sample interaction with one of the earliest known text adventures [CW77].) Because computers came with a means for displaying text already, all of the author's concentration could be left to designing the *narrative world* underneath: things such as the map, the descriptions of places and items, the modeling of reactive elements, such as machines and non-player characters. Items strewn throughout the game world could be collected and combined to solve puzzles, such as fixing broken machines or bribing stubborn characters. Text adventure enthusiasts often speak of the compelling sense of

Figure 1.1: Sample interaction with the game *Adventure* by Will Crowther. (Also known as Colossal Cave.)



```
Colossal Cave Adventure ▸ Score: 36 ▸ Turns: 3
objects.  Part of the game is trying out different commands and
seeing what happens.  Type "help" at any time for game
instructions.

Would you like more instructions?

> no

You are standing at the end of a road before a small brick
building.  Around you is a forest.  A small stream flows out
of the building and down a gully.

> enter building

You are inside a building, a well house for a large spring.
There are some keys on the ground here.
There is a shiny brass lamp nearby.
There is tasty food here.
There is a bottle of water here.

> get lamp

Ok.
What's next? █
```

agency they experience when playing such games: thinking of the right command to type involves a sense of imagination ("What would I do if I were really there?"), which is an entry point into a conversation with the game world in which the player tests the boundaries and rules established by the author. In other words, the player may participate in a *world* created by the game author in order to collaboratively author a *story* with the computer. So the ability to simulate worlds with computers has enriched how we tell stories, too, and has opened up new spaces at the intersection of play and narrative.

After the text adventure's heyday, the mass commercialization of digital games engendered a proliferation of "genre" terms, such as "first-person shooter" and "platformer," that entered the public consciousness as a taxonomy of play. In such a taxonomy, the *story game* is thought of as something entirely separate from the *strategy game*, with the former thought mainly of as alternatives to light reading (interaction mainly "turns the page" in a digital book) whereas the latter is thought of as "more of a game," with real decisions and player agency. These categories then inform how designers conceive of new games, and, at one further layer of indirection, how creators of game design *tools* think to build affordances and libraries. So now it is difficult to discuss, much less create, digital works that focus on things like mechanics, systems, and player agency for narrative play—for example, text games that may require ingenuity and strategy to tell their stories.

A counter-framework for designing and analyzing digital games from first principles was proposed by Hunicke et al. [HLZ04], suggesting a game be thought of in terms of *mechanics*, *dynamics* and *aesthetics* (MDA). Mechanics are the basic rules of the game (Chess example: queens can move diagonally), dynamics are the emergent patterns and strategies that arise from mechanics (Chess example: book openings), and aesthetics are the player's holistic experience of the game (Chess example: the sensory experience created by the pieces and the board; the cognitive experience of thinking several moves ahead to plan a strategy). Such a framework allows one to find similarities in games across artificial "genres" in terms of their mechanics or aesthetics, yet contrast them along another of the three axes.

From a game *creation* perspective, the MDA framework also enables us to imagine building a model of the mechanics separately from the game's rendering as a whole, aesthetic experience. This approach lends itself well to *rapid prototyping*, the idea that designers do not want to build a complete end-to-end product before iterating on their initial design, and so they should build rough but servicable mock-ups (prototypes) until the design seems likely to work. Especially when creating games based on complex systems with lots of interacting components, it can be helpful to test those systems piece-by-piece before they are too large to reason about.

From a computer science perspective, we are in search of *composable abstractions* for understanding narratives and games. The process of designing those abstractions, together with defining how they may be brought to life through execution and interaction, is what constitutes the design of a programming language. The executable model should give the designer some sense for the dynamics (and perhaps aesthetics) of the eventual finished game, and it may also provide some automation to speed development: automatically testing the game by carrying out some designated player strategies, for example. It could also perform analysis of the program and tell the author whether or not it has met some desired game properties. Such analysis plays a similar role to *regression testing* in current-day software practice, but it has the ability to check exhaustively for adherence to the specification.

Creating new abstractions for understanding different classes of computation has been the purview of the field of programming languages for several decades. Before digital computers had been invented, we had notions of computation that were independent of the particular physical artifacts for carrying it out, such as Church's λ-calculus [Chu32]. After a long detour through the "von Neumann bottleneck" [Bac78], we can now write programs in these machine-independent language models and run them on real machines. Meanwhile, game designers have been inventing their own programming languages, such as Inform 7 [Nel01], Twine [KA08], and Versu [ES], that suit the specific needs of people crafting interactive stories with incredible immediacy, in terms of affordances that match their imaginative process. But too often, these languages lean on the tradition of machine dependence (imperative state update) to establish their meaning, resulting in leaky or unstable abstractions. We champion the goal of these tools— accessible means for non-computer scientists to participate in the creation of games—but suggest that an approach based on machine-independent, linguistic techniques would serve the community. Furthermore, in sight of the broader goal of *understanding* games at a basic and timeless level, we posit that neither the class of games in question nor the programming tools in question should depend on the trends nor limitations of current technology.

As such: this thesis project endeavors to provide a conceptual and computational framework for understanding, building, and analyzing the interactive worlds that underlie games and stories.

1.1 Potential Narratives

Text adventures eventually came to be understood under the larger umbrella term *interactive fiction*, or interactive works that tell stories with words. Like computation itself, the history of interactive fiction begins before (and continues to exist outside of) the digital computer.

Nick Montfort, an interactive fiction author and historian, traces the history of interactive fiction to the *I Ching*, an ancient Chinese divination tool that used random chance to generate prophetic text, and to the Llull machine, a similar Western invention [Mon05]. The connection is that these systems generate text with algorithms, based on some underlying model, much like an interaction with a digital computer to derive a story. In the 1960s, the French surrealist group of mathematicians, artists, and other intellectuals going under the name Ouvroir de Littérature Potentielle, or *Oulipo* [Mot86], gave the term *potential literature* to these systems that were not stories themselves, but produced stories through human interaction. Following in step, we adopt Montfort's term *potential narrative* to describe our approach to narrative play, because we find it a useful way to unify the playful aspects of interactive fiction with more readily-accepted notions of game.

Interleaved with the development of digital interactive fiction were the "Choose Your Own Adventure" and "game book" collections of print media devised as youth entertainment, which present passages of text ending in choices, each of which corresponds to a page in the book that the reader should turn to to continue the story. [HWF10] These stories with enumerated choices at each branching point bear a strong structural similarity to the digital form of *hypertext*, now widely known as the format of the World Wide Web: text is organized into passages, each of which can contain *links* to other passages. A resurgence of interest in hypertext narratives came with the introduction of Twine [KA08], an accessible tool for creation in this form. Even these more finitary notions of participatory storytelling can be thought of as potential narratives: each path through the story graph corresponds to one story.

Even as text-based digital games have fallen out of mainstream fashion in favor of the dazzling possibilities afforded by modern graphics cards, storytelling still drives the imagination behind widely-acclaimed commercial games, for everything from the grand-scale adventure of Skyrim [Stu11] to the more personal world of Gone Home [Gay13]. The idea of inhabiting a fictional persona to explore and change a rich, reactive environment has seemingly never fallen out of favor, and the new experiences enabled by digital games seem to have the potential for lasting impact on human culture as much as literary and cinematic fiction is now accepted to have. By tapping into both historical precedent and computational possibilities, understanding these timeless forms of art can bridge the humanities and sciences, fostering new means of expanding human knowledge.

The relevance of interactive fiction to the broader domain of games and interactive simulations that I investigate in this dissertation is less about the fact that they tell stories with words and more about the fact that they *free the game designer* from certain constraints that are present when thinking in terms of graphical motion rendering. Emily

Short and Richard Evans, the authors of the interactive storytelling authoring system Versu, write about a similar concern in terms of the *composability* of text versus other generative systems, and while we are more concerned with internal representation than rendering, their explanation illustrates the relative cognitive concerns involved. [ES]

The main abstraction that our work borrows from interactive fiction and storytelling is that of describing a *possible event space* in which an actor may intervene. As a domain of computer science, interactive fiction contains interesting problems in the systems that underlie it, and the questions of how to make those systems more expressive.

1.2 Generative Systems

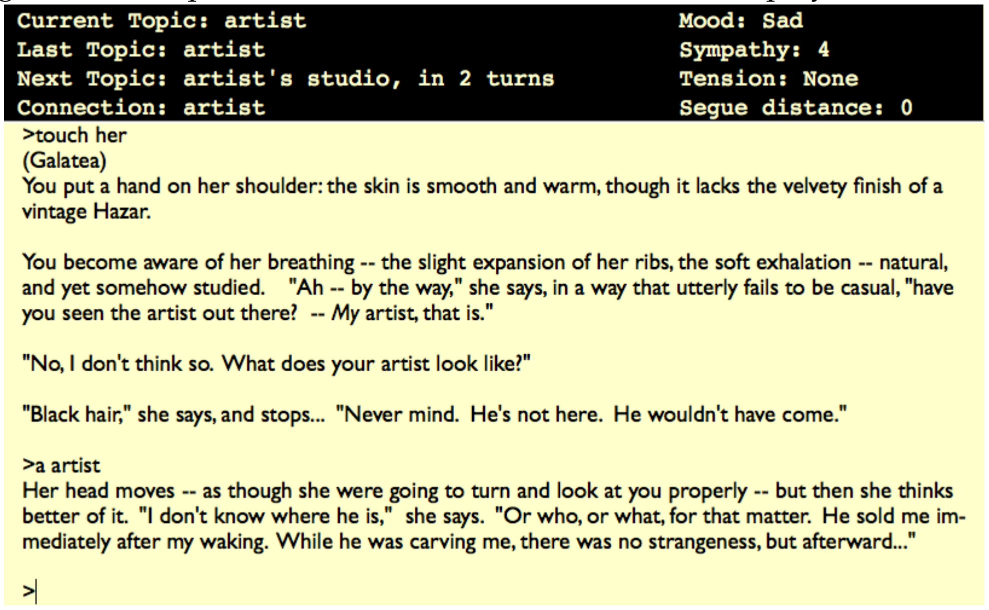
One of the central problems in expressive interactive narrative design also concerns many other artistic and scientific disciplines. The problem is one of human-created versus computer-created content. For any piece of media that changes in response to interactor input, the changes need to be generated on-demand. This means that any sequence of interactions needs to create a representable state. In the case of branching stories, any sequence of *choices* should result in a meaningful narrative situation. One possibility, and the one chosen for Choose Your Own Adventure books and hypertext, is for an author to create the amount of content necessary for every possible run in the world simulation. But when a world is composed of many components, such as a player inventory, the problem becomes intractable: every new variable doubles the number of possible states. Instead, computational methods can be used to create new scenes on-the-fly without having been seeded with them by the author. For instance, a data structure to track the player inventory across different rooms can be seen as a tool for generating a large number of story states. We refer to the techniques involved in systematically composing a small number of hand-generated pieces to derive a large state space as *generative methods*, though it is sometimes referred to as *procedural content generation* [COM13]. We review a few examples of games using generative methods that we take as inspiration for the kind of system we are interested in finding abstractions to describe.

One domain for such methods is the simulation of non-player characters (NPCs). A game can track a detailed model of character *interiority* [Pot05] including mood, knowledge, sentiment, and goals. *Galatea*, an interactive fiction game by Emily Short [Sho00], uses a complex dialogue model to create a "living statue" character that reacts differently to player commands depending on her mood, sympathy, tension, and conversation history and topics (see Figure 1.2). The system generates a wide space of possible stories due to variations with respect to these factors. Short implemented Galatea using a quip-and-response dialogue model reminiscent of a state machine. [1]

Another problem domain is the simulation of multiple characters or agents. Will Wright is an acclaimed game designer known for systems that generate worlds. He is the primary designer behind the Sims series, including SimCity [Wri96] and The

[1]See `https://emshort.wordpress.com/how-to-play/writing-if/my-articles/conversation/` for Short's own explanation.

Figure 1.2: Sample interaction with *Galatea* with state display tool enabled.



```
Current Topic: artist                          Mood: Sad
Last Topic: artist                             Sympathy: 4
Next Topic: artist's studio, in 2 turns        Tension: None
Connection: artist                             Segue distance: 0
```

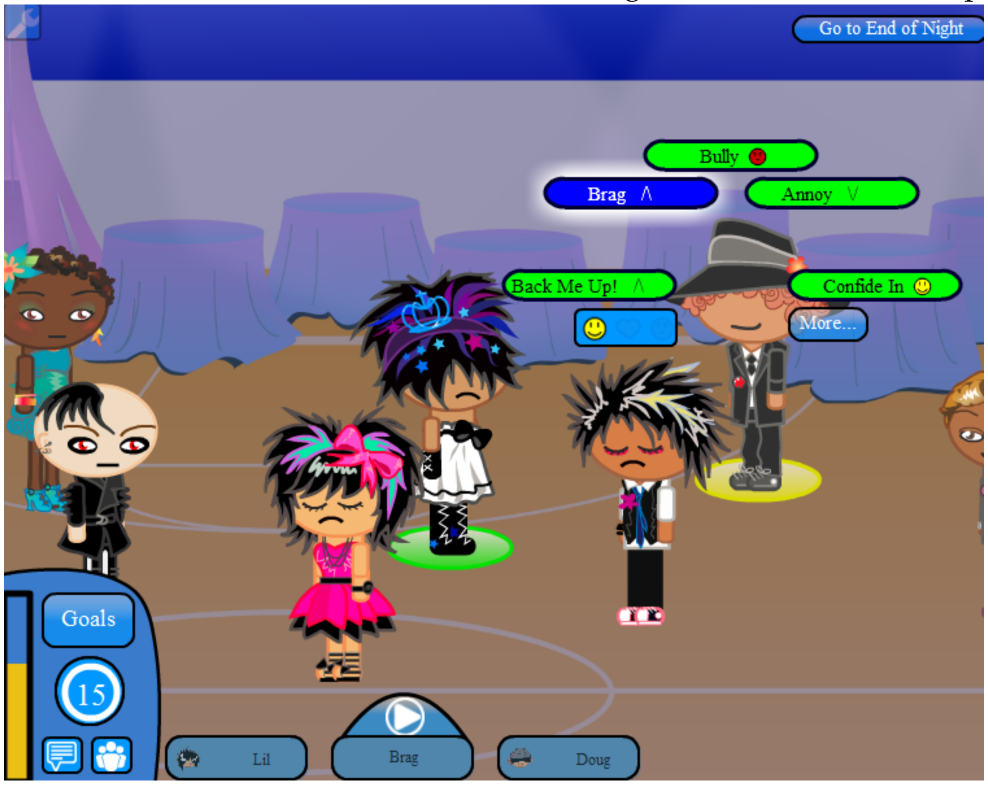
```
>touch her
(Galatea)
You put a hand on her shoulder: the skin is smooth and warm, though it lacks the velvety finish of a
vintage Hazar.

You become aware of her breathing -- the slight expansion of her ribs, the soft exhalation -- natural,
and yet somehow studied.   "Ah -- by the way," she says, in a way that utterly fails to be casual, "have
you seen the artist out there?  -- My artist, that is."

"No, I don't think so. What does your artist look like?"

"Black hair," she says, and stops...  "Never mind.  He's not here.  He wouldn't have come."

>a artist
Her head moves -- as though she were going to turn and look at you properly -- but then she thinks
better of it. "I don't know where he is,"  she says.  "Or who, or what, for that matter.  He sold me im-
mediately after my waking. While he was carving me, there was no strangeness, but afterward..."

>|
```

Sims [Wri00]. None of the systems in these games are explicitly stated as *potential narratives*, but they nonetheless allow human collaboration to tell stories. The connection with The Sims is perhaps most clear: the game provides a "digital dollhouse" with which to perform stories at the player's whim, without imposing external narrative goals. Sim City works similarly, but with a focus on city configuration and aggregate population data rather than the personal relationships and goals of the Sims. Spore [Art08] was a virtuosic attempt to include simulations at multiple levels within an overarching meta-system. All of these systems compose experiences out of myriad changing state components, requiring very little hand-authored content. The idea that a story can nonetheless be derived from such a system is sometimes called *emergent narrative* [Ayl99]. The implementation challenges of such a system, as far as we are aware, mainly lie in managing a very large evolving state (i.e. the status elements of every individual Sim).

More recent investigations into the space of generative narrative experiences include the high school social simulation game, Prom Week, which uses sophisticated social models to determine how characters react to actions such as bullying or confiding based on their own sentiments toward and memories of the acting character (see Figure 1.3). Prom Week grew out of a tradition in interactive drama that includes the landmark production of Façade [MS03], i.e. "believable drama." Their aim was to create a system that would generate a story using *social physics*, or rules for character interaction, rather than presenting hand-authored scenes. The primary technology driving Prom Week is actually a language, Comme il Faut (CiF) [MTS$^+$10], in which the social physics engine is implemented. Inference based on character interiority is used to generate possible intents, which combine with player selection to determine actions.

We present these examples as a (partial) survey of state-of-the-art techniques in

Figure 1.3: Screenshot of *Prom Week* demonstrating character interaction options.



playable narrative based on generative methods. Having established this context, we can now describe the goals, methods, and contributions of the dissertation.

1.3 Thesis Goals

Our overarching goal, extending beyond the scope of this thesis but informing its approach, is to develop computational tools and conceptual frameworks that support creating novel, experimental games and narrative systems. We frame the problem as one of *programming language design* in the sense that this perspective entails creating new abstractions with platform-independent, mathematical semantics. We also aim to bring high-level design language in closer alignment with programming constructs, i.e. make it so that the means of expressing ideas as executable artifacts closely mirrors game designers' conceptual framing.

The narrower goal for this thesis is to fully develop a correspondence between a theory of formal logic proofs and ideas in narrative and game design, and to realize its consequences in the form of a modeling tool. We have noticed a strong match between recently-emerging programming methodologies—specifically *forward-chaining, substructural logic programming*—and the idioms needed to express game worlds and mechanics, especially interactive fiction, and by exploring this connection we aim to advance ideas in both fields: new programming languages for game design can expand designer creativity potential, and new application domains for programming languages can expand the range of computing phenomena accounted for by mathematical understanding.

Our target audience for this work is mainly research communities within game design and logic, although we aspire to create an appeal to (non-academic) systems-oriented narrative and game designers, and programmers with an amateur interest in game design. Ideally, this work will serve to bridge communities interested in programming language theory and game design and reveal many of the overlapping concerns: for instance, both programming language design and game design consitute the creation of *interfaces* with a human user.

As artifacts whose goals are often dependent on human cognition such as *expressiveness*, *productivity*, *flow*, and *emotional impact*, both games and programming languages are difficult to evaluate in quantitative measures, at least with purely computer science-based techniques. If we intend to deepen a holistic understanding of activities like play and programming, clearly some humanistic and psychological understanding will be needed. Nonetheless, there are still questions that may be answered about the computer's half of the bargain, and indeed that can be answered with precise mathematics. It is those questions that we hope to bring into greater clarity for games through methodologies commonly practiced in programming language theory.

In programming languages, the search for objective evaluation criteria has turned mainly to mathematical proof: we create a formal model of a new programming language design, then prove its soundness with respect to the appropriate formal notions. For instance, a common theorem proved for a new language is *type safety*, which says that certain rules circumscribing the set of admissible programs in the language ensure that no program in that set will crash nor reach a stuck point in its evaluation. One can imagine by analogy that the rules circumscribing a *game* might also ensure some formal properties: for instance, in a multiplayer board game, we might want to ensure that play is *productive* in the sense that as long as the game is not in an end state, some player can always make a move that changes the game. The notion corresponding to a *program* in this setting is a *playthrough* of the game.

But before these proofs can be formalized, and thus before we can truly subject games to mathematical evaluation, we need a language for codifying playful rule systems themselves. This thesis describes a candidate for such a language, which is based on a logical system called *linear logic*. We demonstrate how narrative structure can be described abstractly in linear logic, then go on to see how we can realize linear logic as a suitable programming language.

Part of this thesis is a new programming language designed specifically for the support of narrative and game design, and its own design has the following goals:

1. Has a mathematical foundation for the sake of clear, portable, and extensible semantics
2. Is general enough to describe a wide range of operational logics in games
3. Can describe important aspect of narrative structure, supporting causal reasoning and dependency analysis between narrative events

4. Has the potential for accessible front-end tools to be built atop it.

Our evaluation methods are twofold: first, by a selection of case studies using the language, and second, by mathematical proofs about the language.

For the first criterion, we develop case studies that we believe to be representative of the generative storytelling ideas surveyed in this introduction. The code for each of these studies, as well as our reports on the iterative development process, serve as the non-quantitative results of the thesis, available for subjective evaluation to anyone who might consider using the language. Our own subjective judgment on the success of these case studies is presented at the end of Chapter 5.

For the second, we illustrate the use of a mathematical reasoning framework for stating and proving theorems about games encoded in our system. We prove that, in general, questions of whether certain properties hold of games are decidable. Thus, automated checking of designer-stated properties of games may potentially be integrated into the game prototyping and development process.

1.4 Approach

Now that we have stated our goals for the thesis, we may be more specific about the computer science tools that we use to achieve them.

1.4.1 Preliminary Definitions

While play and stories as aspects of human culture elude fully-characterizing definitions, we nonetheless need to define the scope of this thesis. Below we provide some working definitions.

We define a *world* as a *state space* together with *mechanics*. A *state space* is a set of parameters that can take on values that evolve over time. For example, the *state space* of Chess includes the board, pieces, the locations and captured status of the pieces, and which player's turn is in effect. A *state* is an element of the state space, e.g. a particular configuration of pieces on a Chess board. Mechanics are rules for how states can change.

Worlds may also specify one or several permissible *initial states*, in which case they can be *run*, or evolved step by step according to its mechanics.

An *interactive world* is a world whose mechanics allow human participation or intervention while the game is running. We can supplement this definition with informal, subjective notions to say when an interactive world is a *game* or a *potential narrative*; that is, an interactive world is a game when the interactor feels that they are *playing* it, and an interactive world contains *narrative* (or is a *story world*) when its runs can be understood by a human as stories.

The one aspect of these definitions that has not been completely detailed is what is meant by *rules* for state change. By a rule, we mean something approximately of the form "*If X then Y*," often codified as logical implication $X \supset Y$ ("X implies Y"). But standard logical implication does not adequately capture the idea of transforming state, as we explain next, leading us to an alternative formalism.

1.4.2 Linear Logic

Game rules refer not to a monotonically increasing body of knowledge but to a heterogeneously evolving state. They tell us what is permissible, but also what changes when an action is carried out. The fact that a player's rook is in a given Chess board cell may hold at one point in the game, but after an action is taken, its position may change. So it is not a "fact" in the same way that $1 + 1 = 2$ is a fact; whether it holds depends on *time* or some abstraction of it.

Logics of action and time have been studied well beyond the specific domain of games, and include temporal logic [Lam94], situation calculus [MH69], and event calculus [KS89]. All of these calculi stratify logical propositions by explicit timesteps, and action specifications increment the timestep. However, in order to model a world with a complex state space containing many components, each of these logics needs a notion of *inertia* or *frame rule* that says "whenever an action changes the state of some components, the rest of the state stays the same." These rules need to be axiomatized, leading to unwieldy proofs. Additionally, most applications and extensions of these logics have been concerned with adapting them to model the real world (e.g. for robotics applications) such that they do not easily suit the world of narrative fictions [RY10].

Jean-Yves Girard's *linear logic* [Gir87] is a logic that captures essential notions of state change without explicit reference to time. The observation behind linear logic is that standard logics' monotonicity could be accounted for by the so-called *structural rules*, or rules defining how assumed or known facts could be used in a proof—specifically, assumptions may be freely duplicated and ignored. If one explicitly marks the duplication and ignoring of assumptions, rather than building them into the logic, then what emerges is a notion of logical consequence embodying *state transition*, or evolving systems at a component-wise level, rather than permanent knowledge. Furthermore, linear logical deductions are structured in such a way as to track *resource dependency* between different state transitions, making them subject to analysis in terms of causality.

Thus, linear logic's nature as a proof-theoretically-justified "logic of changes" makes it suitable for our purposes to desribe narrative actions and game rules in this thesis. We detail its definition and use for narrative spefications in Chapter 2.

1.4.3 Logic Programming

Logic programming is the main technique we will use to link the logical with the computational. Traditionally, logic programming is realized by languages such as Prolog,[2] wherein programs are collections of clauses (logical implications) together with a *query*, or directive for proof search. A proof search engine then executes the program by attempting to assemble a proof of the query from the program clauses. For example, a program defining addition as a logical predicate `plus(X,Y,Z)` that holds whenever `X+Y=Z` may be given the query `plus(X,Y,10)`, which would compute all pairs of numbers that sum to 10. In general, there may be many proofs or no proofs. Generally one

[2]See `http://www.swi-prolog.org/` for one implementation.

is interested primarily in *whether or not* there is a proof, and if so, which terms satisfy it. In this domain, the engine searches for a proof with *backward chaining*, or reasoning backwards from a goal to the assumptions available.

However, in our case, the *proof itself* is a relevant artifact of study—it embodies the causal dependencies between events in the interactive world, which we illustrate in Chapter 3. In the case of simulation and generation of narratives, as well as open world games, we are often not especially interested in a particular goal (i.e. outcome of the simulation) so much as the process and intermediate states, which are recorded by the proof itself. In particular, we construct proofs with *forward chaining*, or reasoning forward from a set of assumptions (representing an initial world configuration), to model evolution of the interactive world "forward in time."

Because of our shift in focus from goal-based inference to forward simulation, we distinguish *proof search*—a process that may backtrack, and typically performs an exhaustive search of the provability space—from *proof construction*, or the use of logically valid deductions to navigate a space while not necessarily arriving at a particular goal. Put another way, proof construction may sacrifice *completeness* of provability, but not soundness.

In addition to its primary function in our domain as a representation system for rewriting rules over partial states, logic programming offers support for a variety of powerful computational idioms, such as user-defined datatypes, relation-based definitions, and pattern matching on narrative states. Many of these idioms are also found in approaches developed by the Artificial Intelligence (AI) community, such as planning and generative grammars. Many researchers in games have applied similar techniques and referred to them as "AI techniques" [TZE$^+$15] rather than "programming language features"—by presenting these ideas from a programming language point of view, we hope to bring some unification to ideas across these fields.

1.4.4 Logical Frameworks

A final source of techniques that we use as a starting point for our work is *logical frameworks* as described by Harper et al. [HHP93]. The aim of logical frameworks is to provide a basis for formal codification of deductive systems in such a way that theorems can be stated and proved by a human, and checked by computer, within that system. The logical framework LF was realized as the proof assistant Twelf [PS98] for this activity. Twelf's underlying machinery is largely based on logic programming, although the "logic programs" written within it are typically analyzed, not executed. Logical frameworks research has experimented with extending the underlying logic of Twelf to linear logic, and the community has since been searching for suitable notions of specification such that similar meta-theoretic capabilities to Twelf are possible. We rely heavily on progress in this area, and contribute to that progress, for our concerns around proving properties of games in Chapter 6.

1.5 Contributions

This dissertation describes the use of linear logic to specify potential narratives and game mechanics. Our primary contribution is a full explication of a deep correspondence between *proof* and *play*, or alternatively *proof* and *narrative*, in a way that gives a formal justification for unifying play and narrative both conceptually and for the sake of computational tools that can aid their development.

The formal model is implemented as two different logic programming languages, one existing prior to this thesis work (Celf) and one implemented for the purposes of deeper investigation into this work (Ceptre). Essentially, these investigations serve the purpose of pushing the envelope of the formalism as far as it can go to find the limits of its suitability for, and correspondence with, games and potential narrative specifications.

Our specific claim follows in the form of a thesis statement.

> **Thesis statement:** *Using linear logic to model interactive worlds enables rapid prototyping of experimental game designs and deeper understanding of narrative structure.*

As evidence for this claim, we provide a written explanation of the correspondence between linear logic proofs and stories, then illustrate the use of that correspondence for *narrative generation* (in Celf) and *game prototyping* (in Ceptre). These results extend the limits of present knowledge in how formal logic can be used to support generative methods and interactive world authoring, and they also provide supporting case studies for a particular methodology in programing language design. We detail how each part of the dissertation aligns with the thesis statement below.

Chapter 2 describes the correspondence between linear logic proofs and causally-structured story, building largely on work by Bosser et al. [BCC10]. This chapter mainly serves to establish the conceptual framework in which we understand narrative and simulation events, separate from the notions of generation and interaction.

Chapter 3 shows how to *operationalize* the interpretation of narratives in linear logic as programs, via a standard and general extrapolation of logical definition into proof construction procedure, known as logic programming. In this chapter we illustrate how to use the programming language Celf to generate narratives through a narrative world specification given in terms of linear logic. This chapter supports the *deeper understanding of narrative structure* part of the thesis statement.

Chapter 4 describes the development of a new programming language, Ceptre, suitable for programming not just *generative* but also *interactive* worlds. We introduce a new programming construct called *stages* that serve to partition a world specification into parts, then coordinate the control flow between those parts, resulting in a practical tool for prototyping game mechanics and simulations.

Chapter 5 presents several case studies of Ceptre's use for developing interesting examples in the union of narrative and play design. We posit this chapter as the primary support for the *rapid prototyping* part of the thesis statement.

In Chapter 6, we describe progress toward reasoning tools for linear logic programs that allow authors to specify and check constraints on the evolving state. Such tools, once fully automated, would allow world authors to specify their intent and ensure that program rules preserve that intent. This chapter serves as secondary support for the *rapid prototyping* part of the thesis statement, since program errors can often cause slow progress toward building a working prototype.

Finally, in Chapter 7, we conclude by recapitulating the thesis statement, summarizing how the work above has fulfilled its promises, and suggesting avenues of future work.

Chapter 2

Linear Logic for Narrative Structure

2.1 Introduction

What makes up a story? In computer science research, this question has been invoked repeatedly by artificial intelligence and computational linguistics researchers. The motivation for answering it includes several potential applications: autonomous agents that can collaborate on building stories with people (*interactive storytelling*) [CCM02, MS99]; computational understanding of narrative structure from natural language text (*narrative extraction*) [CJ08]; and the automatic generation of stories, a kind of investigation on the boundaries of computational creativity (*narrative generation*) [Mee76].

Besides these computational applications, understanding narratives can lend insight into human learning and cognition: narratives are an important way that humans make sense of the world around us [Her03, Bru90]. Narrative researchers have achieved consensus that *causality* is an important aspect of stories for narrative models to capture [TS85, Ada89, MW00, Dah12], even if it only barely scratches the surface of interesting facets of narrative [RFW09]. To understand causality in stories, we first need to be able to break them apart into events between which causal relationship may be found.

As described in Chapter 1, we are interested not only in the structure of existing stories but in systems that allow or describe *collaborative construction* of story. Understanding a story as made of separable pieces is a good first step for this purpose, too, since narrative play involves at least one human mediating between story components.

In this chapter, we aim to give a foundational account of narrative systems. We are interested in representations of narrative that can be interpreted by a computer for the sake of analysis, generation, and interaction, while also using high-level language constructs that are not bound by particular machine models.

We present *intuitionistic linear logic* (ILL) as a formalism to serve as the foundation for modeling narratives and their structure. Linear logic, first devised by Girard [Gir87], was originally suggested as a suitable representation for narratives by Bosser et al. [BCC10], whose work we largely recapitulate here.

Figure 2.1: Fabula versus Sujet in Memento

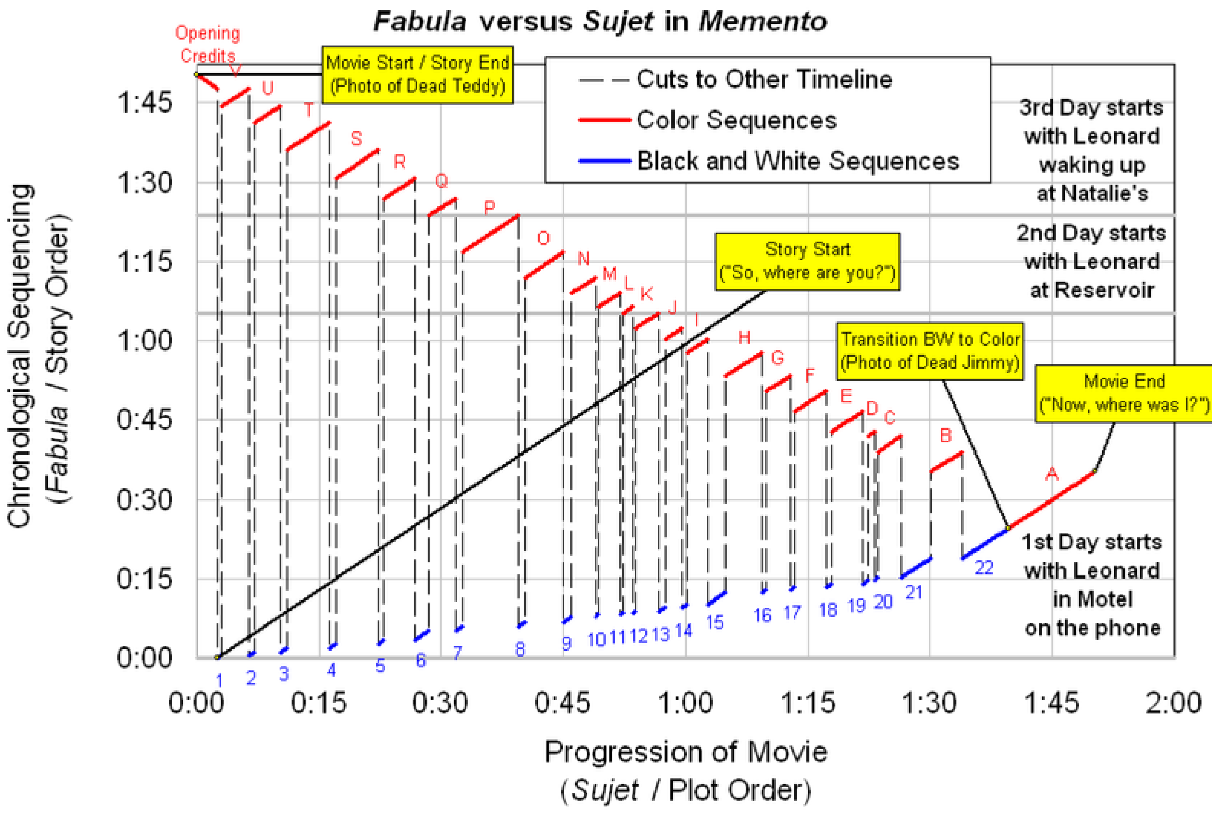


Diagram created by Steve Aprahamian, from `https://en.wikipedia.org/wiki/File:Memento_Timeline.png`, used on the conditions of the Creative Commons license described therein.

2.2 Modeling Narratives

We aim to give a composable formalism for narrative, so we must make clear what aspects of narrative we aim to make formal. In particular, we make the distinction between *fabula* and *sujet* made by Russian formalists in the 1920s [Bro74]: the fabula is the plot events, the "what happens" part of a story, whereas the *sujet* is the *telling* or orchestration of story elements into a (usually linear) digestible tale. In this work we aim only to formalize the *fabula*, while recognizing that there are many interesting research questions in the formalization of sujet and relating fabula to sujet (see, for example, the conflicting structure between the two in the film *Memento*) [Nol00] (see Figure 2.1).

The formal modeling of fabula has a long history in AI research, but until recently has depended on ad-hoc, genre-specific ontologies such as Propp's "morphology of a folktale" [Pro10]. Instead, we would like the particular genre or conflict structure to be *reflected* in the formalism, but not *constrained* by it. The core structure we would like to constrain with the formalism instead includes things like causality, action, and change. These properties are essential to enforcing *coherence* of a narrative [You99].

Folktales as a domain contain many interesting examples for the study of fabula, since their tellings vary with every storyteller, but their component events remain relatively consistent. Let us consider as an example the folktale *The Three Little Pigs*. This folktale consists of the following narrative events:

1. The mother pig sends her three baby pigs away from home, warning them about a big bad wolf.
2. The first pig builds a house out of straw.
3. The second pig builds a house out of sticks.
4. The third pig builds a house out of bricks.
5. The big bad wolf visits the straw house and blows the house down.
6. The big bad wolf visits the stick house and blows the house down.
7. The big bad wolf visits the brick house and attempts to blow it down, but cannot.

Even within formalization of fabula, there are many conflicting ideas about what parts of this story to include in narrative state and how to model relationships between these narrative elements. Some candidates for narrative components include:

- Characters in the story: the three little pigs, their mother, and the big bad wolf.
- Resources or props in the story that are needed to drive the story forward, such as the straw, sticks, and bricks that the pigs use to build their houses, and subsequently the houses themselves.
- Relationships between any of the above entities: feelings of family and allyship between the pigs; feelings of enmity between the pigs and the wolf; locations and possessions of characters.
- Scenes, or story events, which require availability of the above story elements and may change them. *The first pig builds a house out of straw* might be understood as changing the state of the pig, the straw, and the house.
- Establishment of initial configurations for characters, resources, and relationships. The first line of the story establishes the three little pigs, their mother, and the wolf as characters in the story.
- Endings, or narrative goals established by the author. "Happily ever after" is a common goal ending for fairy tales; in variants of The Three Little Pigs, the defeat of the wolf is an important constant.

Finally, we may also need to represent relationships between *scenes*. These relationships may not be represented explicitly by an author, but they are important for computational applications of story. For instance, in the context of interactive stories, we may need to include or derive links between story scenes to represent alternatives. In the context of story *analysis*, we may care about the ability to determine *causal* relationships between scenes, as mentioned in the introduction.

2.2.1 Linear Logic by Example

Modeling a story in logic means mapping the above concepts to the constructs of logic, which is to say *propositions* and the notion of *derivability* between propositions. Propositions can either be *atomic* or *compound*. An atomic proposition is something like "The straw house is standing," which we might abbreviate with the notation $\mathsf{straw\_house}$. Compound propositions are how we combine atomic propositions into more complex statements, analogous to English usage of *and*, *or*, and *implies* to combine statements. The logical analog of these conjoning words are the *connectives*.

The following linear logic proposition could model the story event of the wolf blowing down the straw house:

$$\mathsf{wolf} \otimes \mathsf{straw\_house} \multimap \mathsf{wolf}$$

The atomic propositions shown here are wolf and $\mathsf{straw\_house}$. The connectives are $\multimap$ ("lolli," linear logic implication) and $\otimes$ ("tensor," linear logic conjunction). $\otimes$ binds more tightly than $\multimap$, so the rule can be read as, "If the wolf is present and the straw house is standing, then transition to a state where (only) the wolf is present."

Note that this reading avoids the word *implies* or an *if-then* reading, although we asserted that $\multimap$ is linear logic's form of implication. In linear logic, derivability is defined in such a way that it represents *state change* more than inference about permanent truth. This mechanism lets us treat propositions as *resources*, or in this case, slices of narrative state, and implication (embodied by derivability) can be read as replacement of one piece of state with another.

Logical connectives need not always form compound propositions: for instance, the "connective" 1 represents the empty or null resource, and we can use it to model the removal, rather than replacement, of a piece of state. For instance, the following proposition models removing the wolf from the story:

$$\mathsf{wolf} \multimap 1$$

On the other hand, this proposition models introducing the wolf into the story from out of thin air:

$$1 \multimap \mathsf{wolf}$$

Another connective in linear logic, $\&$ ("with"), allows us to represent choices between alternatives. The following proposition models a transition from the pig state (representing a pig with no house) to either a straw, stick, or brick house:

$$\mathsf{pig} \multimap \mathsf{straw\_house} \,\&\, \mathsf{stick\_house} \,\&\, \mathsf{brick\_house}$$

Note that we could equally well represent this choice-yielding event as instead a choice between events:

$$(\mathsf{pig} \multimap \mathsf{straw\_house}) \,\&\, (\mathsf{pig} \multimap \mathsf{stick\_house}) \,\&\, (\mathsf{pig} \multimap \mathsf{brick\_house})$$

So far, all of these propositions are just syntax; we have not attempted to explain what they mean in terms of logical derivability, except at an intuitive level. But one of the more important omissions we have made is the status of these propositions that represent story events: how do we intend to compose them?

Linear logical implications do not really accurately model concrete story events: they instead describe a *possible* event that could occur given the fulfillment of the antecedent. We are instead describing a story *world*, a set of possibilities that we intend to give rise to a story or set of stories through logical deduction.

In this sense, we can imagine composing implications $E_1 \dots E_n$ representing story events as the compound proposition

$$E_1 \otimes \dots \otimes E_n$$

meaning that all events are simultaneously available for use. However, if we have two events $A \multimap B$ and $A \multimap C$, and A is some narrative state that we only expect to happen once, then one of the events will go unused, which is disallowed in linear logic.

Sometimes we *want* to enforce the usage of a given event because we are interested in *narrative drive*: the idea that an author needs certain scenes to occur to fulfill certain dramatic intent. In this case, the tensoring-together of narrative events should suffice, and any conflicting scenes would need to be carefully crafted as alternatives (e.g. $A \multimap B \,\&\, C$).

On the other hand, sometimes we are interested in modeling a more exploratory possibility space where some events may not take place at all, and others may take place multiple times. The unary connective ! gives us exactly this meaning: $!A$ is a proposition A that may be ignored or repeated. Thus, we can represent the more exploratory story world as follows:

$$!E_1 \otimes \dots \otimes !E_n$$

We hereafter refer to a proposition of the form $!(A \multimap B)$ as a *rule*. Since we may refer to rules multiple times in a story, we will often name them using the syntax $r : R$, where r is a name and R is a rule.

In the next section, we seek to clarify the meaning of these propositions and simplify our representations by explaining them in terms of a few carefully-considered, meta-logical principles.

2.3 Linear Logic

Any representation of fabula needs to include models of *action* and *change*, phenomena for which many formalisms have been investigated over several decades [MH69, KS89, Lam94]. The cited works effectively solve the variability problem by indexing every proposition with a time parameter and modeling actions as incrementing that parameter. As a consequence, they all need to explicitly manage *inertia*, or the fact that when a rule describes changes of one part of the world state from time t to $t+1$, the remaining parts of the world state are updated in $t+1$ to retain their state at time t.

Linear logic is an approach to the logical modeling of action and change that does not depend on explicit time indexing. This approach has the advantage that it is possible to conceive of distinct parts of the state changing independently of one another.

We note that, unlike Girard's original formulation of logic, our chosen formulation is *intuitionistic* in that we only consider sequents $\Delta \vdash A$ where A is a single proposition, rather than a disjunction of multiple propositions. The intuitionistic version of linear logic (ILL) was first presented by Girard and Lafont [GL87]. We select it mainly for its computational interpretations as described by Andreoli [And92], although we note that this choice may not be canonical, and classical linear logic bears investigating as an alternative in future work.

Intuitionistic linear logic was later reformulated [CCP03] according to the methodology of logic design espoused by Gentzen and Martin-Löf [Gen35, ML96] in which inference rules operate over *judgments*, or meta-logical syntax pertaining to propositions, and inference rules also obey certain meta-theoretic properties that make soundness of the logic easy to establish, and proof search within the logic easy to automate. This methodology, which includes giving a *sequent calculus* formulation of the logic, is described below.

First, we assert that to be a *logic* means to have a notion of *consequence*:

$$A_1, \ldots, A_n \vdash A$$

The above syntax can be read, "A is a consequence of $A_1, \ldots, A_n$," where A and all A_i are propositions (defined by the particular logic). Such a statement is called a *sequent*, and an arbitrary collection of assumptions $A_1, \ldots, A_n$ is often notated as Δ and called the *context*.

A logic may have *inference rules* of the form

$$\frac{J_1 \quad \ldots \quad J_n}{J}$$

where each J and J_i is a sequent, the J at the root of the rule is the *conclusion* sequent, and each J_i is a *premise* that must be satisfied in order for the inference to be valid.

A *proof* of sequent J is a composition of inference rules forming a tree whose leaves are rules with no premises and whose root is J.

Finally, to count as a logic, the notion of consequence must be *transitive*; that is, if it is possible to build proofs of $\Delta \vdash A$ and $\Delta', A \vdash C$, then we can also build a proof of $\Delta', \Delta \vdash C$.

This property, having a transitive notion of consequence, is also known as Admissibility of Cut [Gen35]. It is a meta-property of a system of inference rules and part of what gives it meaning. There are other such properties, called structural properties, that have often been previously considered part of what it means to be a logic:[1]

[1]In the literature, the terminology "consequence relation" has sometimes implied these properties. We depart from this tradition by using "consequence" without including weakening and contraction by

Weakening. If $\Delta \vdash C$ then $\Delta, A \vdash C$: in other words, adding unneeded assumptions does not affect derivability.

Contraction. If $\Delta, A, A \vdash C$ then $\Delta, A \vdash C$: in other words, having two copies of a proposition available is not any better than a single copy.

These properties encapsulate the idea that in a proof from assumptions, each assumption may be used arbitrarily many times to reach the goal. Linear logic, however, denies these properties, leading to a "use exactly once" treatment of assumptions in a proof. Put another way, if ordinary logic treats the context Δ as a *set* of assumptions, in linear logic Δ can be thought of as a *multiset* [Gir95] i.e. the multiplicity of assumptions matters.

Thus linear logic's notion of consequence $\Delta \vdash A$ embodies a sort of conservation property, and can be read "Resources Δ may be transformed into A." Connecting this notion back to narrative, we can think of Δ as an initial narrative situation and A as a narrative ending.

To include the ! connective in linear logic, we need to reintroduce a context that *is* subject to weakening and contraction. If we change our basic judgment (sequent form) to

$$\Gamma; \Delta \vdash A$$

where assumptions in Γ may be weakened and contracted, then we have the machinery we need to define !, effectively reintroducing ordinary, persistent notions of logical fact into the by-default resource-oriented logic. In this sense, intuitionistic linear logic can be thought of as a refinement on ordinary intuitionistic logic.

In this more foundational account of the logic, we can rewrite our story world representation from Section 2.2.1 as simply a specification for the context Γ: instead of imagining

$$!E_1 \otimes \ldots \otimes !E_n$$

as an assumed proposition in Δ, we can instead stipulate that

$$\Gamma = E_1, \ldots, E_n$$

We can now describe the mapping between story world and logical sequent that we will use as our primary model of representation going forward. The sequent $\Gamma; \Delta \vdash A$ represents a story world with rules (and persistent facts) in Γ, initial narrative configuration described by Δ, and narrative goal (or story ending) A.

Now that we have sketched the judgmental framework in which we intend to model story worlds, we can finally give the formal definition of linear logic as a sequent calculus.

default.

2.3.1 Intuitionistic Linear Logic: Sequent Calculus

In a sequent calculus, every connective is defined by two rules: instructions for how to prove it when it appears on the right-hand side of the turnstile ($\vdash$) in a sequent, and instructions for how to *use* it when it appears on the left-hand side of the turnstile. These instructions come in the form of *left* and *right* inference rules. We now present the inference rules of linear logic that define the connectives we need for narrative modeling.

The $\otimes$ connective models simultaneous conjunction between two resources:

$$\frac{\Gamma; \Delta_1 \vdash A \quad \Gamma; \Delta_2 \vdash B}{\Gamma; \Delta_1, \Delta_2 \vdash A \otimes B} \otimes R \qquad \frac{\Gamma; \Delta, A, B \vdash \gamma}{\Gamma; \Delta, A \otimes B \vdash \gamma} \otimes L$$

From a proof search perspective, we read these inference rules from the bottom up. So the right rule says that a proof of $A \otimes B$ can be formed from a context that can be partitioned into pieces Δ_1 and Δ_2, which can prove A and B respectively. The left rule says that an assumption of $A \otimes B$ can be interpreted as two separate assumptions, A and B.

Note that the persistent context Γ is simply repeated in every sequent in this rule (i.e. from a proof search perspective, it is passed unchanged to both subgoal sequents). The Γ context will be "carried through" in this manner for all inference rules that do not refer to it; that is, all rules except those defining !.

The rules for the tensorial unit, 1, are as follows:

$$\frac{}{\Gamma; \cdot \vdash 1} 1R \qquad \frac{\Gamma; \Delta \vdash \gamma}{\Gamma; \Delta, 1 \vdash \gamma} 1L$$

The null resource may be proven only when the linear context Δ is empty, and it may be used by simply removing it from the context.

The rules for implication ($\multimap$) follow a similar pattern to those for $\otimes$, except inverted with respect the the left and right rule:

$$\frac{\Gamma; \Delta, A \vdash B}{\Gamma; \Delta \vdash A \multimap B} \multimap R \qquad \frac{\Gamma; \Delta_1 \vdash A \quad \Gamma; \Delta_2, B \vdash \gamma}{\Gamma; \Delta_1, \Delta_2, A \multimap B \vdash \gamma} \multimap L$$

The right rule for $\multimap$ says that to prove $A \multimap B$, add A to our set of assumptions and work on proving B. The left rule requires again that we *partition* the context into the part that proves the antecedent A, and another part that, when given the consequent B, continues to prove the original goal.

The inference rules for $\&$ are given below – but note that we do not use this connective in the remaining chapters of the thesis, so understanding it is optional.

$$\frac{\Gamma; \Delta \vdash A \quad \Gamma; \Delta \vdash B}{\Gamma; \Delta \vdash A \& B} \& R \qquad \frac{\Gamma; \Delta, A \vdash \gamma}{\Gamma; \Delta, A \& B \vdash \gamma} \& L_1 \qquad \frac{\Gamma; \Delta, B \vdash \gamma}{\Gamma; \Delta, A \& B \vdash \gamma} \& L_2$$

The right rule says that to prove $A \& B$, we need to be able to prove A and B *from the same context* Δ. The left rule says that if we have a choice $A \& B$, we can make either selection to continue the proof.

The rules for ! are the ones that interact with the persistent context Γ:

$$\frac{\Gamma; \cdot \vdash A}{\Gamma; \cdot \vdash !A}\ !R \qquad \frac{\Gamma, A; \Delta \vdash \gamma}{\Gamma; \Delta, !A \vdash \gamma}\ !L$$

The $!R$ rule says that a resource A can be thought of as a fact $!A$ so long as it is proven *only* with factual propositions (the ones in Γ). Thus the rule enforces that the linear context Δ must be empty.

The left rule allows us to move a $!A$ from the linear context into the persistent context as just A.

Finally, there are two structural rules that interact with the contexts Δ and Γ. One of these is the copy rule:

$$\frac{\Gamma, A; \Delta, A \vdash \gamma}{\Gamma, A; \Delta \vdash \gamma}\ \mathsf{copy}$$

This rule allows us to copy any persistent assumption A into Δ (without also removing it from Γ, meaning we may do this as many times as needed).

Finally, we need a structural rule for proving (and equivalently using) atomic propositions:

$$\frac{}{\Gamma; p \vdash p}\ \mathsf{init}$$

This rule allows us to finish a branch of proof search by observing that the context and goal match up exactly (modulo persistent assumptions Γ).

We could imagine a more general init rule

$$\frac{}{\Gamma; A \vdash A}\ \mathsf{init}'$$

where A can be any proposition, not just atomic p, but it turns out that this rule is *admissible* in a well-designed logic, i.e. the rules already allow it for any given A. Furthermore, testing for its admissibility is a way to determine that the logic is well-designed; it is considered an internal completeness property.

2.3.2 Some Derivation Examples

We can see each of the connectives $\multimap$, $\otimes$, $\&$, and ! as internalizing some property of the judgmental framework: $\multimap$ internalizes consequence, $\otimes$ internalizes the comma for conjoining contexts, $\&$ internalizes a choice between two alternatives, and ! internalizes persistence.

In future chapters of this thesis, we will use all of these connectives frequently except for $\&$; in fact, refinements on this formalism will not include $\&$. We omit $\&$ because in settings where all implications are rules (live in the persistent context), occurrences of $\&$ to the right of a rule can be accounted for by adding multiple rules. In other words, a rule of the form $A \multimap B \& C$ can be replaced by two rules, $A \multimap B$ and $A \multimap C$, in a sound and complete way. [2]

We now prove this fact—that $\&$ is an unneeded connective in the context of persistent rules—which will also serve as an example of how to put the inference rules above together into proofs.

Claim:

$$!(A \multimap B \& C)$$

is interderivable with

$$!(A \multimap B) \otimes !(A \multimap C)$$

That is, we need to show that

$$!(A \multimap B \& C) \vdash !(A \multimap B) \otimes !(A \multimap C)$$

is derivable, as is its converse

$$!(A \multimap B) \otimes !(A \multimap C) \vdash !(A \multimap B \& C)$$

Recall that a *proof* is a composition of inference rules forming a tree whose leaves are rules with no subgoals and whose root is the sequent being proved. We form a proof by applying inference rules to the current goal sequent, resulting in a new set of goal sequents (the premises to the rule), until there are no more premises.

The first derivation works as follows:

Let $\Gamma = A \multimap B \& C$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma; A \vdash A} \quad \dfrac{\dfrac{}{\Gamma; B \vdash B}}{\Gamma; B\&C \vdash B}\,\&L_1}{\Gamma; A \multimap B\&C, A \vdash B}\,\multimap L}{\Gamma; A \vdash B}\,\text{copy}}{\Gamma; \cdot \vdash A \multimap B}\,\multimap R}{\Gamma; \cdot \vdash !(A \multimap B)}\,!R \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma; A \vdash A} \quad \dfrac{\dfrac{}{\Gamma; C \vdash C}}{\Gamma; B\&C \vdash C}\,\&L_2}{\Gamma; A \multimap B\&C, A \vdash C}\,\multimap L}{\Gamma; A \vdash C}\,\text{copy}}{\Gamma; \cdot \vdash A \multimap C}\,\multimap R}{\Gamma; \cdot \vdash !(A \multimap C)}\,!R}{\dfrac{A \multimap B \& C; \cdot \vdash !(A \multimap B) \otimes !(A \multimap C)}{\cdot; !(A \multimap B \& C) \vdash !(A \multimap B) \otimes !(A \multimap C)}\,!L}\,\otimes R$$

The leaves of this derivation are all of the form $\Gamma; A \vdash A$ which we know to be derivable for any proposition A by way of the identity admissibility theorem.

The second derivation works as follows:

Let $\Gamma = A \multimap B, A \multimap C$ in the derivation below.

[2]Rules of the form $A\&B \multimap C$, on the other hand, do not have a direct translation into the $\&$-less fragment. However, we have not encountered a need for such rules in any of the examples we investigate.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma; A \vdash A} \quad \dfrac{}{\Gamma; B \vdash B}}{\Gamma; A \multimap B, A \vdash B} \multimap L}{\Gamma; A \vdash B} \text{copy} \quad \dfrac{\dfrac{\dfrac{}{\Gamma; A \vdash A} \quad \dfrac{}{\Gamma; C \vdash C}}{\Gamma; A \multimap C, A \vdash C} \multimap L}{\Gamma; A \vdash C} \text{copy}}{\Gamma; A \vdash B \mathbin{\&} C} \mathbin{\&}R}{\Gamma; \cdot \vdash A \multimap B \mathbin{\&} C} \multimap R}{A \multimap B, A \multimap C; \cdot \vdash !(A \multimap B \mathbin{\&} C)} !R}{\cdot; !(A \multimap B), !(A \multimap C) \vdash !(A \multimap B \mathbin{\&} C)} !L^2}{\cdot; !(A \multimap B) \otimes !(A \multimap C) \vdash !(A \multimap B \mathbin{\&} C)} \otimes L}{}
$$

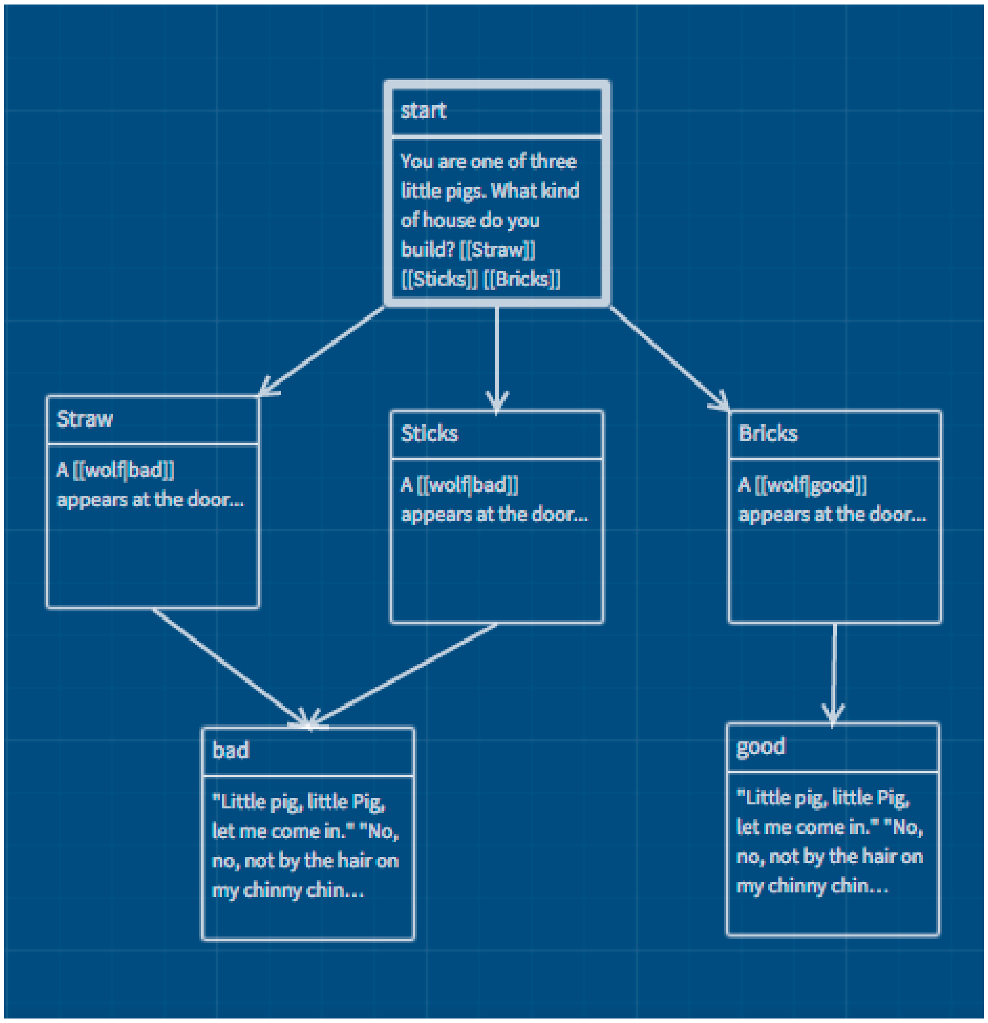
2.4 Alternative and Simultaneous Story Structure

Linear logic provides the tools we need to specify story worlds in such a way that we can define relationships of interest between story events, but there are still quite a few choices about which components to model in the story world that give rise to different event relationships. Let us now be more specific about what kinds of event relationships we would like to aim to model, and explain how we can use linear logic to model them.

2.4.1 Alternative Storylines

One kind of narrative structure of interest to interactive storytelling is *alternative storylines*. This structure is present in any Choose Your Own Adventure or Twine game: a reader is presented with multiple choices and, upon selecting one, is cut off from the other choices until replay (or until that scene is presented again, in the case of cyclic stories). These stories have a natural representation as a *graph*, with story scenes (or *passages* in Twine parlance) modeled by graph vertices, and possible consequences y of a scene vertex x modeled as directed edges from x to y. Sometimes this kind of narrative structure is called *branching*, but by calling attention to *alternatives* rather than branching, we aim to include story graphs that are not simply trees: paths that diverge (branch) may converge in a later scene, and divergence is not given any priority over convergence.

The graph structure just described is depicted in the Twine editor for the sake of the author to visualize their story structure. To relate this idea to our example, we can render the Three Little Pigs fabula as a Twine story in which the choices represent the alternatives between building a house out of straw, sticks, or bricks:


start
You are one of three little pigs. What kind of house do you build? [[Straw]] [[Sticks]] [[Bricks]]
Straw
A [[wolf|bad]] appears at the door...
Sticks
A [[wolf|bad]] appears at the door...
Bricks
A [[wolf|good]] appears at the door...
bad
"Little pig, little Pig, let me come in." "No, no, not by the hair on my chinny chin...
good
"Little pig, little Pig, let me come in." "No, no, not by the hair on my chinny chin...

This depiction of the story structure can be seen as containing three alternative storylines, each corresponding to one "playthrough" of the game. Two of those playthroughs end in the "bad" passage in which the wolf blows down the house, and the third ends in the "good" passage in which the wolf cannot blow down the house. The story graph is a more compact representation of these three stories that makes use of the shared structure between the first and second story, as well as their shared beginning. It can be understood as a *potential narrative* because each interaction with a player generates one of the three stories.

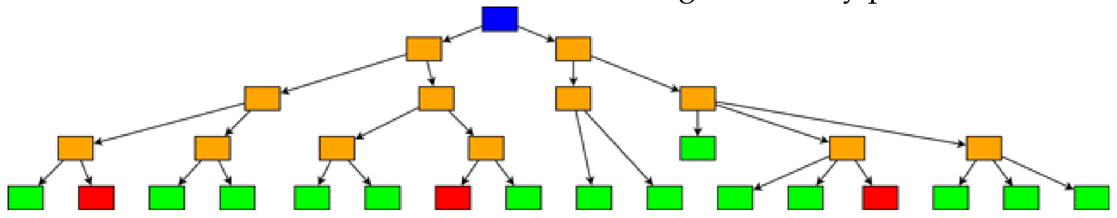
Alternative structure in stories already yields many interesting authoring choices. In an online article, Sam Kabo Ashwell reviews several patterns in the edges-as-alternatives graph structure for interactive storytelling, such as those in Figure 2.2. [3]

However, many rich interactive storytelling examples make use of complex state tracking, such as inventories in parser interactive fiction. These graphs do not depict how the story will vary depending on that state.

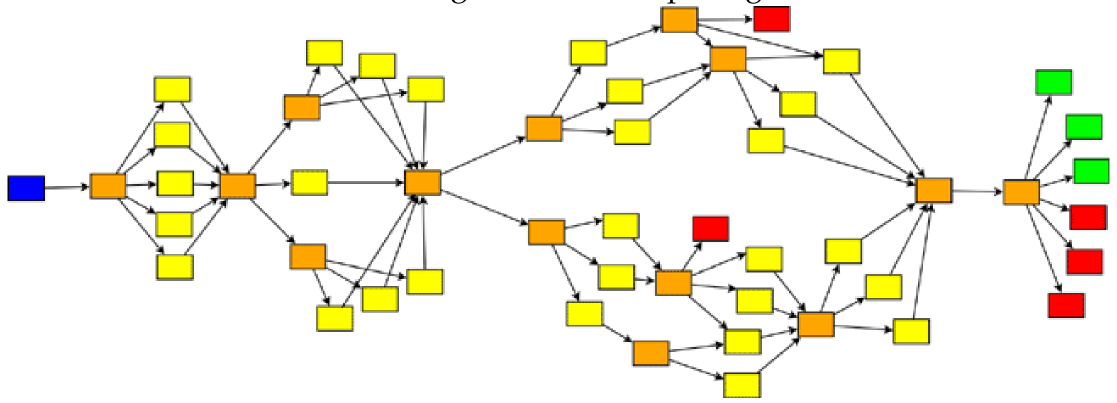
[3]The article is available at
`https://heterogenoustasks.wordpress.com/2015/01/26/standard-patterns-in-choice-based-games/`.

Figure 2.2: Four structural patterns in branching narrative.
Diagrams created by Sam Kabo Ashwell, used with permission.
Color key: red nodes are endings representing failure; green nodes are endings representing success; blue nodes are the start of the story; orange nodes are choice points; yellow nodes are waypoints (they only have a single out-edge).

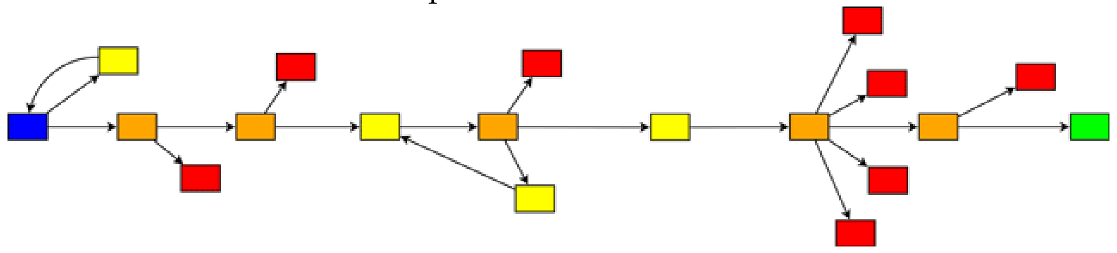
The *time cave*, in which branches never converge and every path is distinct:



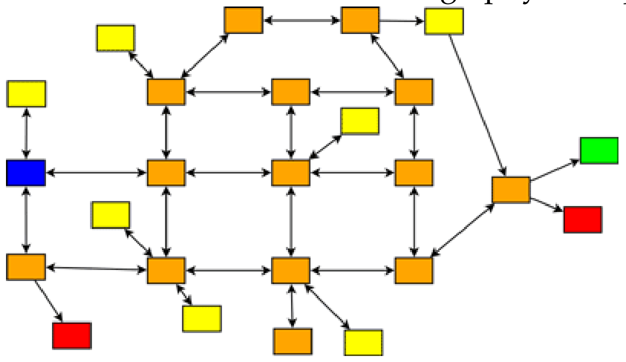
The *branch-and-bottleneck*, in which outward-branching alternatives occasionally re-converge on common passages:



The *gauntlet*, in which one particular path serves as the backbone for the story, and all other paths lead to dead ends:



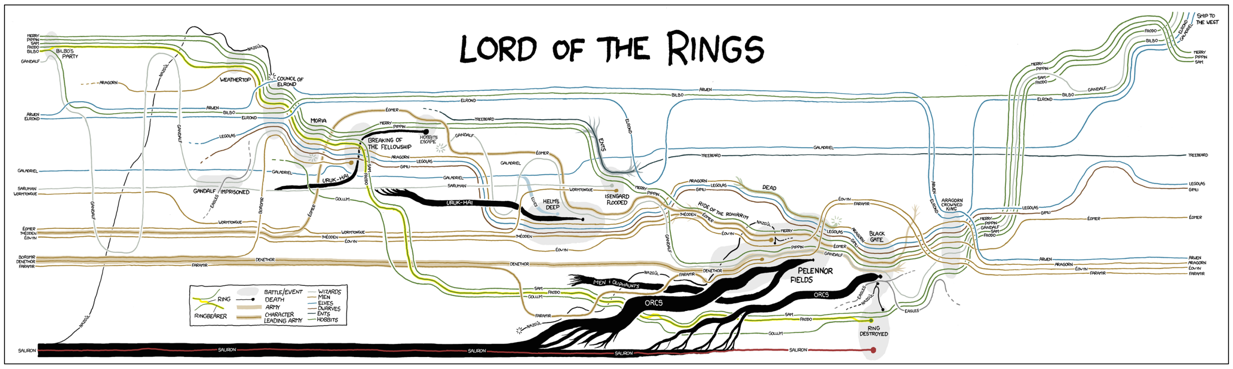
The *open map*, in which symmetric connections are made between nodes, often used to simulate reversible movement through physical space:



2.4.2 Simultaneous Storylines

Alternative structures are one way to conceptualize a story in terms of constituent pieces, but they are not the only way. Rather than imagining making choices from the perspective of one of the three little pigs, we might like to model all of the characters' actions and interactions together. We can map a narrative onto a graph in a different way: edges are characters in certain states of the narrative, and nodes are scenes where those characters interact.

In one strip of the online comic XKCD, Randall Munroe depicts several popular movie plots in this style, such as this plot graph for The Lord of the Rings: [4]

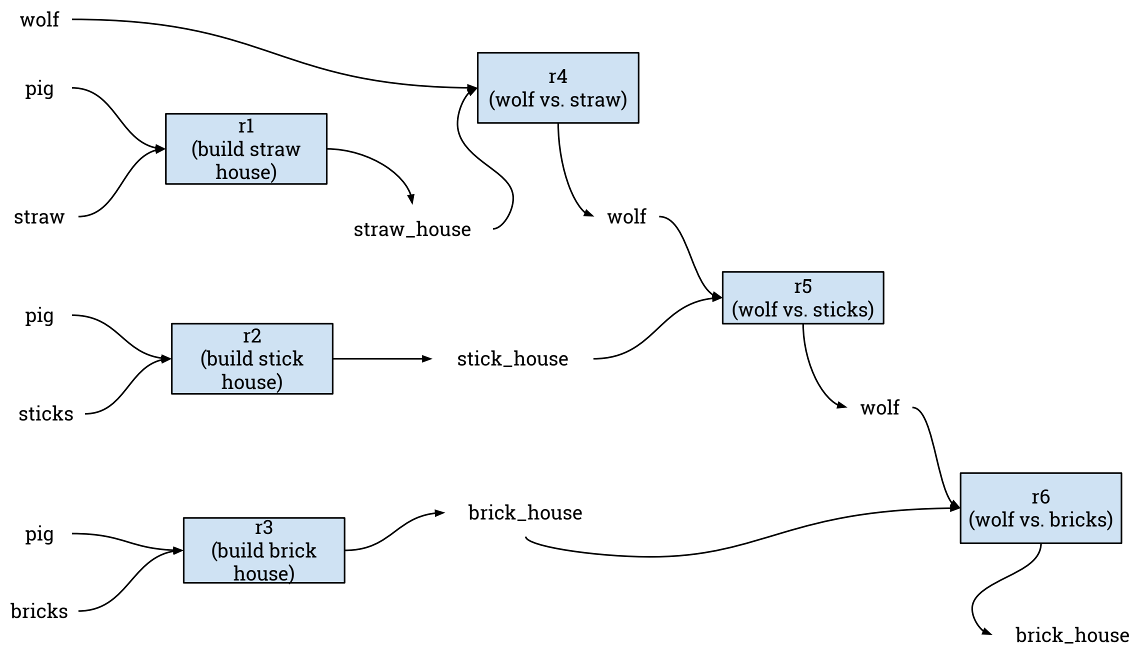


Along the vertical axis, characters are placed near one another to represent sharing a scene or far apart to represent independent activity. The horizontal axis is the forward progression of time. The *ring*, an imporant narrative resource, is drawn as an overlay atop the character holding it.

This version of narrative structure lends itself to the *analysis* of a plot more readily than alternative structure: we can see, for instance, that climactic moments correspond to the convergence of a large number of characters, and we can trace the thread of a particular character or group of characters to examine their overall influence on the plot. More relevantly to narrative play, we can envision the story in terms of autonomous, interacting characters who might have independent motives, interiority, and relationships.

To model the Three Little Pigs narrative in this way, we need to break down our story into smaller component pieces: one for each pig and the wolf, and perhaps also the building materials they use for their houses. If we are more precise than Munroe's drawing about what exactly a "scene" does, in terms of how it transforms these narrative resources, then we can draw the story's simultaneous structure as follows:

[4] Larger version can be found at `https://xkcd.com/657/`.



This diagram represents something more *static* than the Twine game version: it does not present clear affordances for interaction. However, it is also more *systematic*, in that it forces us to imagine a world model in which the events in the story could logically and physically take place. What we present in the next sections is a formal setting in which *both* of these valuable aspects of structural expression are available.

2.4.3 Alternatives and Simultaneity in Linear Logic

Linear logic's resource-oriented mechanisms enables connectives that neatly map onto alternative ($A\&B$) and simultaneity ($A \otimes B$). Due to the "use once" principle of assumptions in linear logic, these manifest as two forms of logical conjunction ("and"). The first one of these we will explain is $A \otimes B$, which means that resources A and B are available simultaneously.

The $\otimes$ and $\multimap$ connectives together give us the ability to write propositions such as $\mathsf{pig} \otimes \mathsf{bricks} \multimap \mathsf{brick\_house}$, modeling a resource exchange that characterizes the "pig builds a brick house" scene in The Three Little Pigs. Here, the $\otimes$ connective serves to conjoin the two atomic pig and bricks propositions into a more complex proposition. There is a *unit* of the $\otimes$ connective indicating the null resource, spelled 1.[5]

The other form of conjunction, $A\&B$, means that a choice between A and B is avail-

[5]For 1 to be a unit of $\otimes$ means that, for any A, $A \otimes 1$ is equivalent to (interprovable with) A and $1 \otimes A$.

able. This connective lets us model the first passage in our Twine game representation of The Three Little Pigs as $\mathsf{pig} \multimap \mathsf{brick\_house} \& \mathsf{straw\_house} \& \mathsf{stick\_house}$. [6]

To model stories in linear logic, we map atomic propositions onto pieces of narrative state. In the case of purely alternative story structures, a single piece of narrative state representing each scene/passage will suffice. In the case of narratives with simultaneous structure, propositions will model potentially more complex narrative elements and their relationships.

Here is a model of the additive story world for Three Little Pigs:

Let $\Delta =$

$$\begin{gathered}\{\mathsf{pig} \multimap (\mathsf{straw\_house} \,\&\, \mathsf{stick\_house} \,\&\, \mathsf{brick\_house}),\\ \mathsf{straw\_house} \multimap \mathsf{wolf\_wins},\\ \mathsf{stick\_house} \multimap \mathsf{wolf\_wins},\\ \mathsf{brick\_house} \multimap \mathsf{pig\_wins},\\ \mathsf{pig}\}\end{gathered}$$

Stories will be proofs of the sequent

$$\Delta \vdash \mathsf{wolf\_wins}$$

or

$$\Delta \vdash \mathsf{pig\_wins}$$

Here is a model of the simultaneous story world, in which we make use of linear logic's notion of simultaneity ($\otimes$) to introduce three copies of the *pig* narrative resource, and we also represent finer-grained narrative resources such as the building materials for the pigs' houses:

Let $\Delta =$

$$\begin{gathered}\{\mathsf{pig} \otimes \mathsf{straw} \multimap \mathsf{straw\_house},\\ \mathsf{pig} \otimes \mathsf{sticks} \multimap \mathsf{stick\_house},\\ \mathsf{pig} \otimes \mathsf{bricks} \multimap \mathsf{brick\_house},\\ \mathsf{wolf} \otimes \mathsf{straw\_house} \multimap \mathsf{wolf},\\ \mathsf{wolf} \otimes \mathsf{stick\_house} \multimap \mathsf{wolf},\\ \mathsf{wolf} \otimes \mathsf{brick\_house} \multimap \mathsf{brick\_house},\\ \mathsf{wolf}, \mathsf{pig}, \mathsf{pig}, \mathsf{pig}, \mathsf{straw}, \mathsf{sticks}, \mathsf{bricks}\}\end{gathered}$$

Stories will be proofs of the sequent

[6]There is a unit of $\&$ as well, spelled $\top$, which we do not include here because it is not needed for our examples.

$$\Delta \vdash \mathsf{brick\_house}$$

Note that in both of these models, scenes (represented as propositions of the form $A \multimap B$) appear on the same level as atomic narrative resources like pig. This means that the logic will enforce a *use exactly once* semantics for those scenes, embodying a kind of *narrative drive* (we are required to include those scenes to prove the sequent). If we want to make a scene A optional, all we need to do is create a choice between that scene and the null resource: $A \& 1$. Persistence ($!A$) can be used to model stories where scenes may repeat an arbitrary number of times.

2.4.4 Proofs as Stories

Finally, we can present concretely the mapping of stories onto proofs. For the Twine-based version of the story, where story events are related as alternatives, we can model the story world with $\Gamma =$

$$\begin{aligned}
r_1 &: \mathsf{pig} \multimap (\mathsf{straw\_house} \mathbin{\&} \mathsf{stick\_house} \mathbin{\&} \mathsf{brick\_house}) \\
r_2 &: \mathsf{straw\_house} \multimap \mathsf{wolf\_wins} \\
r_3 &: \mathsf{stick\_house} \multimap \mathsf{wolf\_wins} \\
r_4 &: \mathsf{brick\_house} \multimap \mathsf{pig\_wins}
\end{aligned}$$

The initial state Δ_0 will be the single resource pig, corresponding to the initial Twine passage.

Here is a derivation corresponding to the story of the pig that builds the straw house in the additive story world:

$$\dfrac{\dfrac{}{\mathsf{pig} \vdash \mathsf{pig}} \quad \dfrac{\dfrac{\dfrac{}{\mathsf{straw\_house} \vdash \mathsf{straw\_house}} \quad \dfrac{}{\mathsf{wolf\_wins} \vdash \mathsf{wolf\_wins}}}{\mathsf{straw\_house} \vdash \mathsf{wolf\_wins}} \multimap L(r_2)}{\mathsf{straw\_house} \mathbin{\&} \mathsf{stick\_house} \mathbin{\&} \mathsf{brick\_house} \vdash \mathsf{wolf\_wins}} \& L_1}{\Gamma; \mathsf{pig} \vdash \mathsf{wolf\_wins}} \multimap L(r_1)$$

The above derivation can also be thought of as a proof that the wolf can win (against a pig who makes a straw or stick house). Below is a proof that the pig can win , representing the story in which the pig builds the brick house:

$$\dfrac{\dfrac{}{\mathsf{pig} \vdash \mathsf{pig}} \quad \dfrac{\dfrac{\dfrac{}{\mathsf{brick\_house} \vdash \mathsf{brick\_house}} \quad \dfrac{}{\mathsf{pig\_wins} \vdash \mathsf{pig\_wins}}}{\mathsf{brick\_house} \vdash \mathsf{pig\_wins}} \multimap L(r_4)}{\mathsf{straw\_house} \mathbin{\&} \mathsf{stick\_house} \mathbin{\&} \mathsf{brick\_house} \vdash \mathsf{pig\_wins}} \& L_3}{\mathsf{pig} \vdash \mathsf{pig\_wins}} \multimap L(r_1)$$

Every different proof with $\Gamma; \Delta_0 \vdash \gamma$ at its root corresponds to a different potential narrative embodied by the alternative structure.

To describe *simultaneous* relationships between story events, we need to divide up the monolithic state of "which Twine passage are we in" into smaller pieces, such as each pig, the wolf, the building materials the pigs use to make their houses, and the completed houses.

The story ending is now not a a single fact (the wolf or the pig winning) but rather some composition (specifically, a $\otimes$ conjunction) of outcomes for different narrative resources.

In this version of the story, $\Gamma =$

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

The initial configuration for the story is the state

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

And we can create proofs of the sequent

$$\Delta_0 \vdash \mathsf{brick\_house}$$

to represent stories ending with just the brick house standing, i.e. the canonical Three Little Pigs folktale ending. Here is a proof that the brick house can be the only one left standing, representing the canonical Three Little Pigs story:

Let $\mathcal{D}_1 =$

$$\dfrac{\mathsf{pig} \vdash \mathsf{pig} \quad \mathsf{bricks} \vdash \mathsf{bricks}}{\mathsf{pig}, \mathsf{bricks} \vdash \mathsf{pig} \otimes \mathsf{bricks}} \otimes R$$

Let $\mathcal{D}_2 =$

$$\dfrac{\overline{\mathsf{pig} \vdash \mathsf{pig}} \quad \overline{\mathsf{straw} \vdash \mathsf{straw}}}{\mathsf{pig}, \mathsf{straw} \vdash \mathsf{pig} \otimes \mathsf{straw}} \otimes R$$

Let $\mathcal{D}_3 =$

$$\dfrac{\overline{\mathsf{pig} \vdash \mathsf{pig}} \quad \overline{\mathsf{sticks} \vdash \mathsf{sticks}}}{\mathsf{pig}, \mathsf{sticks} \vdash \mathsf{pig} \otimes \mathsf{sticks}} \otimes R$$

Let $\mathcal{D}_4 =$

$$\dfrac{\overline{\mathsf{wolf} \vdash \mathsf{wolf}} \quad \overline{\mathsf{straw\_house} \vdash \mathsf{straw\_house}}}{\mathsf{wolf}, \mathsf{straw\_house} \vdash \mathsf{wolf} \otimes \mathsf{straw\_house}} \otimes R$$

Let $\mathcal{D}_5 =$

$$\dfrac{\overline{\mathsf{wolf} \vdash \mathsf{wolf}} \quad \overline{\mathsf{stick\_house} \vdash \mathsf{stick\_house}}}{\mathsf{wolf}, \mathsf{stick\_house} \vdash \mathsf{wolf} \otimes \mathsf{stick\_house}} \otimes R$$

in

$$
\dfrac{\mathcal{D}_1 \quad \dfrac{\mathcal{D}_2 \quad \dfrac{\mathcal{D}_3 \quad \dfrac{\mathcal{D}_4 \quad \dfrac{\mathcal{D}_5 \quad \dfrac{\dfrac{\dfrac{}{\mathsf{wolf} \vdash \mathsf{wolf}} \quad \dfrac{}{\mathsf{brick\_house} \vdash \mathsf{brick\_house}}}{\mathsf{brick\_house}, \mathsf{wolf} \vdash \mathsf{wolf} \otimes \mathsf{brick\_house}} \quad \dfrac{}{\mathsf{brick\_house} \vdash \mathsf{brick\_house}}}{\mathsf{brick\_house}, \mathsf{wolf} \vdash \mathsf{brick\_house}}\multimap L(r_6)}{\mathsf{stick\_house}, \mathsf{brick\_house}, \mathsf{wolf} \vdash \mathsf{brick\_house}}\multimap L(r_5)}{\mathsf{stick\_house}, \mathsf{straw\_house}, \mathsf{brick\_house}, \mathsf{wolf} \vdash \mathsf{brick\_house}}\multimap L(r_4)}{\mathsf{straw\_house}, \mathsf{brick\_house}, \mathsf{pig}, \mathsf{sticks}, \mathsf{wolf} \vdash \mathsf{brick\_house}}\multimap L(r_2)}{\mathsf{brick\_house}, \mathsf{pig}, \mathsf{pig}, \mathsf{straw}, \mathsf{sticks}, \mathsf{wolf} \vdash \mathsf{brick\_house}}\multimap L(r_1)}{\Delta_0 \vdash \mathsf{brick\_house}}\multimap L(r_3)
$$

Note the lopsided structure of the above proof: we only apply $\multimap L$ rules up the entire tree except to derive rules' premises from the context. The proof could almost be read as a sequence of rule applications $r_3; r_1; r_2; r_4; r_5; r_6$ that navigate from the bottom sequent (in which the context represents the initial story configuration) to the top-right sequent (in which the context represents the final story configuration). On the other hand, by tracking exactly *which* resources in the context are consumed and produced by each rule application, we can extract a partial ordering between these rule applications, yielding the diagram depicted in Section 2.4.2. The rule sequence plus its input-output dependency links with other rules is exactly the information captured in the formal proof term syntax that we interpret as stories in Chapter 3.

Another note about the above proof is that it is not the only one that may follow from the story world rules and initial configuration $\Gamma; \Delta_0$. We can derive different outcomes where two or three of the houses are left standing, i.e.

$$\Delta_0 \vdash \mathsf{brick\_house} \otimes \mathsf{stick\_house}$$

and

$$\Delta_0 \vdash \mathsf{brick\_house} \otimes \mathsf{stick\_house} \otimes \mathsf{straw\_house}$$

That these sequents are provable suggests that even in our "simultaneous" story world, there is some alternative structure to consider—i.e. there is nondeterminism in the system of inference rules, and several might apply.

To see what stories these unconventional endings might give rise to, let us examine a proof of the sequent $\Delta \vdash \mathsf{brick\_house} \otimes \mathsf{straw\_house} \otimes \mathsf{stick\_house}$ (where we abbreviate the right-hand side of the sequent A).

Let $\mathcal{D}_6 =$

$$
\dfrac{\dfrac{}{\mathsf{brick\_house} \vdash \mathsf{brick\_house}} \quad \dfrac{\dfrac{}{\mathsf{straw\_house} \vdash \mathsf{straw\_house}} \quad \dfrac{}{\mathsf{stick\_house} \vdash \mathsf{stick\_house}}}{\mathsf{straw\_house}, \mathsf{stick\_house} \vdash \mathsf{straw\_house} \otimes \mathsf{stick\_house}}\otimes R}{\mathsf{brick\_house}, \mathsf{straw\_house}, \mathsf{straw\_house} \vdash A}\otimes R
$$

in

$$\dfrac{\mathcal{D}_1 \quad \dfrac{\mathcal{D}_2 \quad \dfrac{\mathcal{D}_3 \quad \dfrac{\dfrac{\dfrac{}{\mathsf{wolf} \vdash \mathsf{wolf}} \quad \dfrac{}{\mathsf{brick\_house} \vdash \mathsf{brick\_house}}}{\mathsf{brick\_house}, \mathsf{wolf} \vdash \mathsf{wolf} \otimes \mathsf{brick\_house}} \otimes R \quad \mathcal{D}_6}{\mathsf{straw\_house}, \mathsf{brick\_house}, \mathsf{stick\_house}, \mathsf{wolf} \vdash A} \multimap L(r_6)}{\mathsf{straw\_house}, \mathsf{brick\_house}, \mathsf{pig}, \mathsf{sticks}, \mathsf{wolf} \vdash A} \multimap L(r_2)}{\mathsf{brick\_house}, \mathsf{pig}, \mathsf{pig}, \mathsf{straw}, \mathsf{sticks}, \mathsf{wolf} \vdash A} \multimap L(r_1)}{\Delta \vdash A} \multimap L(r_3)$$

Again reading the named rules used in the proof from bottom to top, we can see we have a shorter story wherein after each pig builds their house, the wolf simply visits the brick house first and is eliminated from the narrative state there, leaving all three houses standing.

Some computational narrative researchers would consider such a story to be poorly formed, particularly according to popular Western story aesthetics: it can hardly be argued that this narrative has any *conflict*. Furthermore, two characters in the story (the straw house and brick house pig) are seemingly only referred to once, when they build their houses, and never interact with the narrative again, making their role in the story seem inessential. Riedl and Young [RY04] have investigated means of manipulating similar inference systems (planning) to introduce conflict by way of individual character goals that compete over shared resources. In future work we would like to investigate the applicability of their techniques to linear logic.

2.5 First-Order Linear Logic

One advantage of working with the logical methodology we have chosen is that it is easy to enrich the logic with orthogonal constructs. For instance, we can enrich our logic of atomic propositions to allow us to state more complex relationships between entities, such as $\mathsf{at\ pig\ straw\_house}$ to represent location or $\mathsf{has\ pig\ straw}$ to represent possession. If we additionally introduce a *quantifier* connective $\forall$ ("for all"), we can write a proposition modeling a *parametric* action where a pig in *any* location L where there is a building material M can be replaced by the pig possessing M:

$$\forall L, M. \mathsf{at\ pig}\ L \otimes \mathsf{at}\ M\ L \multimap \mathsf{has\ pig}\ M$$

For modeling pre-established narratives like The Three Little Pigs, modeling the story world at this level of detail and parametricity seems unnecessary. However, for the invention of more flexible story worlds prone to expressive collaboration from a human interactor, parametric rules like this one become essential.

The creators of social physics systems like Prom Week emphasize that creating a wide possibility space for narrative unfoldings is essential to the player's discovery of "emergent solutions and surprising, yet satisfying, outcomes." [MTS+11] Expressing the kinds of rules that make up Prom Week's social physics engine (Comme il Faut [MTS+10]) requires that they not be specialized to any particular character in the story but depend only on certain properties of characters and their relationships. The first-order extension of linear logic enables us to express rules of this form.

Most prior work on the use of linear logic to model stories and games focus *only* on the non-parametric, or *propositional*, fragment of the logic (see Section 2.6.2), and so its range of considered examples has mostly been limited to hand-authored, branching narratives. In Chapter 3, we will show how first-order linear logic scales to account for a richer narrative possibility space. We note that, as a side-effect, we will be able to use the same formalism to describe Twine games, command-line ("parser") interactive fiction, and multi-agent social models like Comme il Faut, and compare the narrative structures that arise from them.

2.6 Related Work

Bosser et al. [BCFC11] similarly identify structures in stories that can be identified as simultaneous and alternative, also modeled in linear lofic. In their case, simultaneous structure is represented by proof terms that record uses of the $\otimes R$ rule as "stories in parallel." However, such proofs still sequentialize story events that we identify as formally independent from one another. Our notion of independence aligns better with the permutability of proof rules than with the commutative structure of the linear context.

We identify two other formalisms that are very closely related to linear logic for story formalization: planning and Petri nets.

2.6.1 Planning

Interactive storytelling research communities have historically used planners (such as STRIPS [FN71] and IPOCL [YPM94]) to model and generate stories [You99, CMC01, RY10]. The planning approach to modeling actions, i.e. by designating facts as *preconditions*, *deleted* by the action, and *added* by the action, has a great deal in common with modeling actions as linear logic implications. In fact, Masseron et al. [Mas93] have shown how linear logic can be used to support planning and equate a proof to a plan. Others have demonstrated the use of linear logic for specific problems thought of as typical planning problems [DSB09, DST09].

So, in the interactive storytelling domain, linear logic can effectively be seen as a logic-based alternative to planning. We see the advantages to a basis in logic as twofold: first, that a long tradition of logics connecting to human epistemology means a wide range of epistemological extensions are available, and there is vast precedent for designing logics so that their different judgments (and corresponding connectives) may fit together, composing into richer formalisms. For example, this precedent is what allows us to seamlessly extend propositional linear logic with first-order quantification. Another, less-explored example is combining linear logic's resource orientation with logical connectives for *knowledge* of specific actors [GBB$^+$06], which we hypothesize could be quite fruitful for narrative modeling.

There are a few more technical differences between the expressivity of linear logic and planning as formalisms. For instance, in linear logic, the *multiplicity* of a fact matters: we cannot write brick_house $\multimap$ wolf to represent "deleting" the brick_house fact in the same way we can in planning, because there may be other copies of the brick_house resource available that we do not account for. (We would consider these statements equivalent, though, when an *invariant* of the program includes having exactly zero or one of a certain resource.) A consequence of this fact is that linear logic *can* refer to multiple instances of something (like money) without propositions indexed by a natural number, while planning cannot.

Another (related) difference is that fact deletion in planning does not require that the fact being deleted is actually *present* for the rule to fire. In linear logic, all "deleted" facts are also "preconditions." Finally, planners typically support *negation* of facts as preconditions for actions. Both of these constructs (unconditional deletion, and negation as failure) are inexpressible in linear logic, and in fact are inconsistent with purely logical characterization in general, because they do not maintain the transitivity of the consequence relation. However, when we consider the extension of our approach into a more practical programming language, we do add support for similar idioms (see Chapter 4).

2.6.2 Petri Nets

Petri nets [Mur89] are a formalism studied in the 1970s primarily for concurrent computation. They are easily interpreted in a graphical languages of nodes and arcs, where nodes may be either *places* (represented with a circle) or *transitions* (represented as a rectangle), where arcs connect places to transitions and transtions to place. Each place may have zero or more *markings* on it.

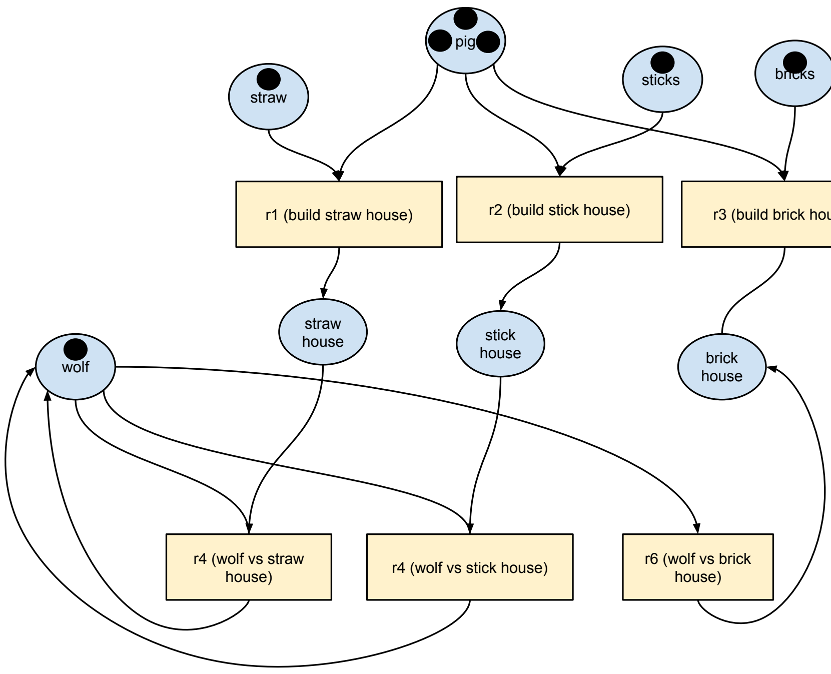
A given Petri net and configuration of markings may evolve to a new configuration as follows: if all of the places that point to (have an arc to) a given transition are marked, the transiton may *fire*, or remove one marking from each input place and add a marking to each output place. Places may have arcs to multiple transitions, suggesting nondeterminism in the system.

Petri nets may be understood as a restricted fragment of linear logic: a Petri net place is modeled by an atomic linear logic proposition, and a Petri net transition matches up to a linear logic implication of the form $!(a_1 \otimes \ldots \otimes a_n \multimap b_1 \otimes \ldots \otimes b_m)$, where each a_i is an input place to the transition and b_is are output places. Every rule must be persistent (thus the ! in front) so that it can model a transition that may fire arbitrarily many times.

The *reachability* problem in Petri nets—whether it is possible for a given configuration to evolve through arbitrarily many rule firings to another configuration—was shown equivalent to provability in this fragment of linear logic by Kanovich. [Kan95]

Note that this correspondence does not take into account the first-order extension of linear logic. For some first-order programs, it may be possible to *ground* them, i.e. give an equivalent propositional program, by instantiating every rule with every possible term given in the domain (assuming the domain is finite, which it may not be), but even when this is possible it results in a combinatorial explosion of the state space. This may

Figure 2.3: Petri net for the simultaneous Three Little Pigs story world.



make Petri nets as a formalism less scalable than first-order linear logic as an authoring tool.

Despite this limit in expressiveness, quite a few applications of Petri nets to games and interactive storytelling have been explored: first, Vega et al. investigate their use in game design [VGN04]; Araújo and Licínio use them to model game mechanics [AR09]; Dang et al. use the equivalent fragment of linear logic to validate interactive story scenarios [DHCS11, DCA13]; and my coauthors and I use that same fragment for generation of story variants [MBFC13] on the fabula of Flaubert's novel *Madame Bovary* [Fla01]. The main *benefit* of the limited expressiveness is that it is possible to do exhaustive analysis, since proof search on that fragment (i.e. reachability for Petri nets) is decidable. [May84]

Petri nets also have the advantage of enjoying a direct visual representation that nicely illustrates the relationship between alternative and simultaneous structure. The Petri net in Figure 2.3 illustrates the Three Little Pigs story world given in Section 2.4.2, and to find potential alternative structure we simply look for places (circles) with multiple out-edges.

Finally, we note that a Petri net-inspired system of markings and transitions, called Machinations, has been built as a visual game design tool [Dor11]. Later work formalized the core of this system and its semantics ("micro-machinations") [VRD14] to find that they differ substantially from Petri nets, losing the correspondence with linear logic.

2.7 Conclusion

We have presented linear logic as a formalism that supports two kinds of structure in narrative fabula, namely *alternatives* and *simultaneity*. We have built upon prior narrative modeling methodologies to map entities, relationships, states, events, stories, and story outcomes onto corresponding logical and proof-theoretic notions, completing the conceptual basis on which the remainder of this thesis builds.

In particular, we have demonstrated several core properties of narrative structure that correspond to the fundamental mechanisms of linear logic. Bosser et al. [BCC10] point out three such characteristics of computational stories: ***generativity***, or the derivability of a variety of different story unfoldings given the same beginning and ending conditions; ***variability***, or the open world assumption characterized by the story rules' independence from the initial story configuration; and ***narrative drive***, or the enforcement of certain story rules (scenes) being used in the proof. These properties emerge from the logic in the form of a nondeterministic inference system, the *hypothetical judgment* (i.e. presence of the context Δ as an essential part of sequents), and the absence of *weakening* making it possible to enforce the usage of assumptions.

We additionally observe the correspondence between linear logic connectives and *alternative* and *simultaneous* relationships between story events. Our observation of the logic's ability to model alternatives is effectively the same as the *variability* property mentioned above, but the *simultaneity* observation was mainly portrayed as causality in prior work. We posit that simultaneity offers a more operational reading of the narrative structure, in which we can imagine several moving parts to a complex interactive narrative, whose independent evolutions compose into a coherent story. Many works of interactive fiction already use such ideas by incorporating persistent state (such as player inventory and tracking of game history), but the simple alternative models of these fictions (e.g. Ashwell's diagrams in Section 2.4) do not depict the more complex story variability that arises from them. Reed and Garbe [RGWFM14] introduce the term *combinatorial narrative* to suggest centralizing the idea of complex narrative states in interactive story authoring, and we posit that linear logic would make a good candidate for modeling and reasoning about those works.

In the next chapter, we will show how to put computers to use on the mathematical formalism presented here in the form of *logic programming* to extract a computational interpretation of story modeling in linear logic. Namely, we extend our mapping to account for story world (or potential narrative) as a program that, when run, generates coherent stories according to the authored rules.

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:

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

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

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

where the notation $\Delta \rightarrow \Delta'$ 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:

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

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 $\Gamma; \Delta \rightarrow \Gamma'; \Delta'$, where $\rightarrow$ 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 $\Delta_0 =$

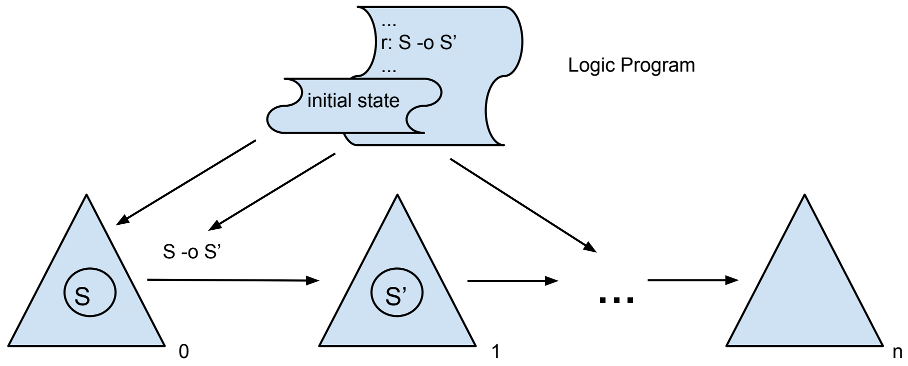
```
{ eros helena hermia, eros hermia helena
  eros helena lysander, eros lysander hermia }
```

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. $\forall x, y.\ \mathtt{p}\ x\ y \multimap \mathtt{q}\ x$.

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.

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

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

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

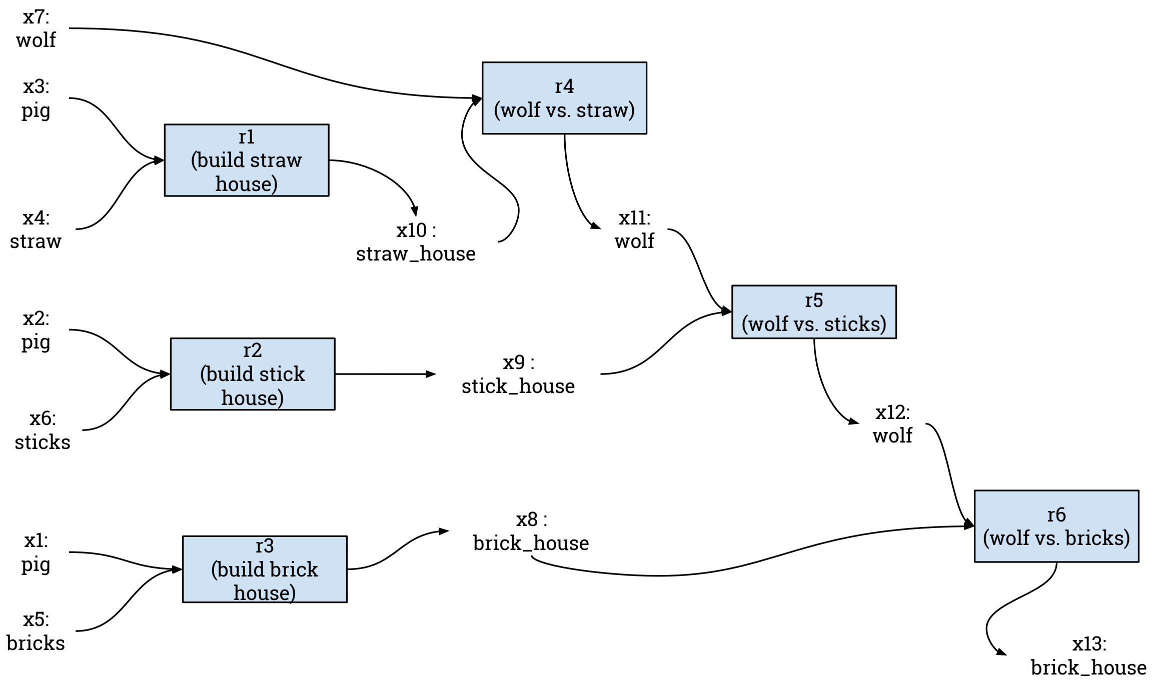
What this representation gives us is a way of describing dependency, and more importantly, *independence* between narrative actions. To see how, let us define an ϵ as a sequence of let bindings representing forward chaining proofs. A trace ϵ can either be empty ($\langle\rangle$), a single binding **let** $p = R$, 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:



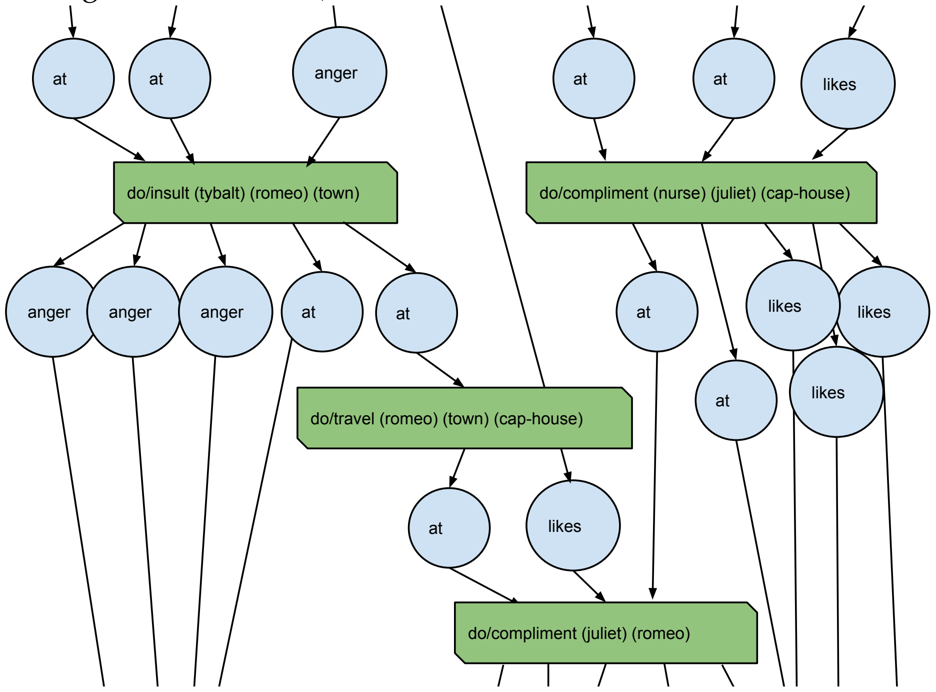
Here we can see that the narrative actions representing the pigs building their houses are independent, and thus the order in which they are told in the story is arbitrary. However, the resources representing the wolf (x_7, $x_1 1$, and $x_1 2$) are threaded through every interaction between house and 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):

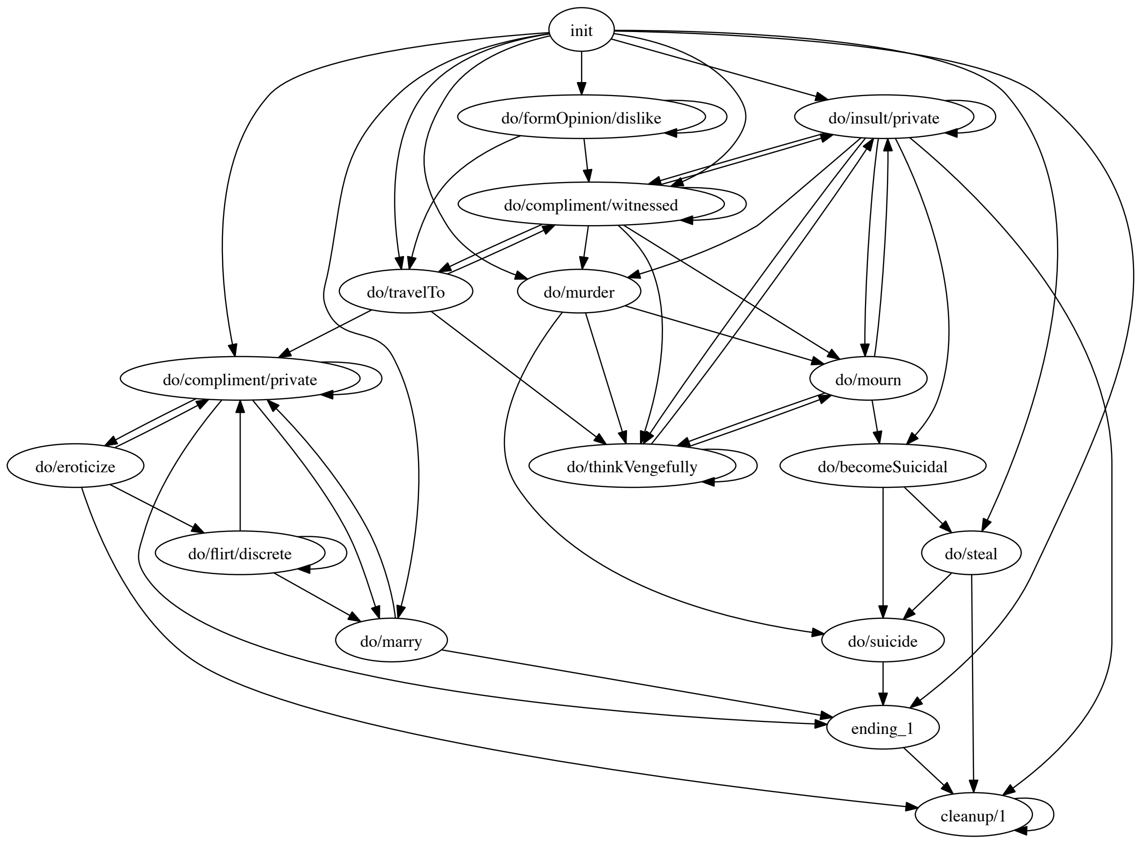

at
at
anger
at
at
likes
do/insult (tybalt) (romeo) (town)
do/compliment (nurse) (juliet) (cap-house)
anger
anger
anger
at
at
at
likes
likes
at
likes
do/travel (romeo) (town) (cap-house)
at
likes
do/compliment (juliet) (romeo)

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

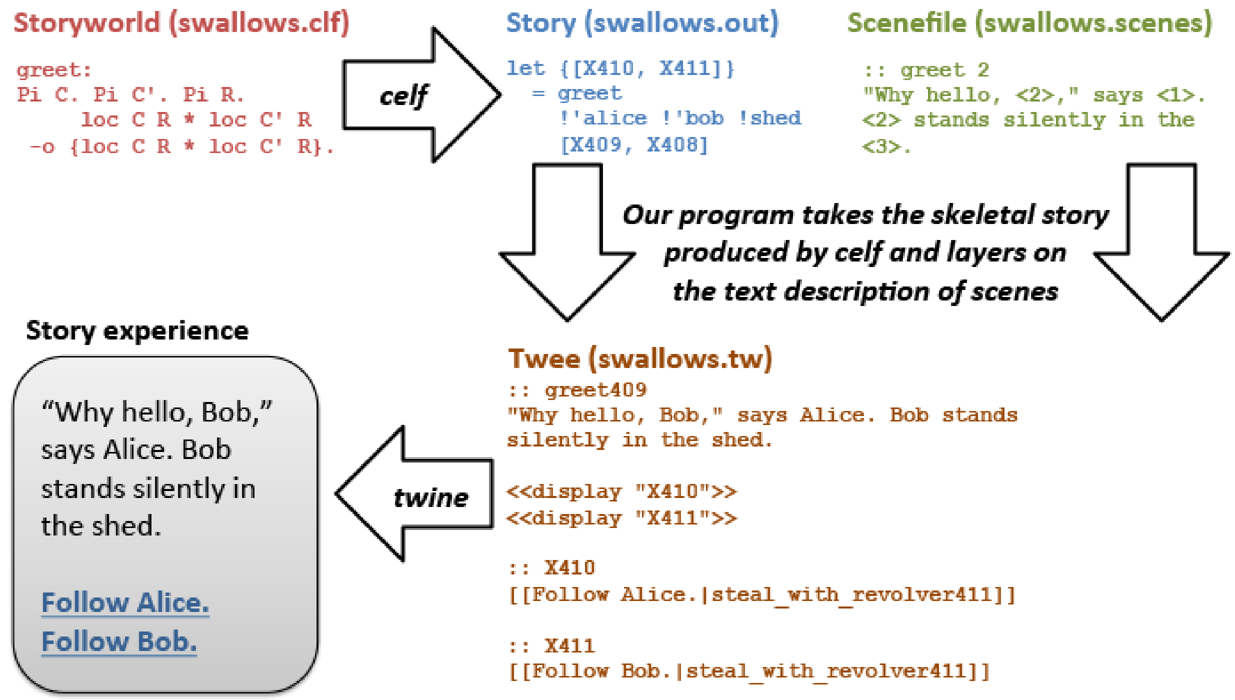
```
exists do/thinkVengefully && ~link do/thinkVengefully do/murder
```

(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):


Storyworld (swallows.clf)
celf
Story (swallows.out)
Scenefile (swallows.scenes)
Our program takes the skeletal story produced by celf and layers on the text description of scenes
Story experience
twine
Twee (swallows.tw)

```
greet:
Pi C. Pi C'. Pi R.
     loc C R * loc C' R
 -o {loc C R * loc C' R}.
```

```
let {[X410, X411]}
  = greet
    !'alice !'bob !shed
    [X409, X408]
```

```
:: greet 2
"Why hello, <2>," says <1>.
<2> stands silently in the
<3>.
```

```
:: greet409
"Why hello, Bob," says Alice. Bob stands
silently in the shed.

<<display "X410">>
<<display "X411">>

:: X410
[[Follow Alice.|steal_with_revolver411]]

:: X411
[[Follow Bob.|steal_with_revolver411]]
```

"Why hello, Bob," says Alice. Bob stands silently in the shed.

Follow Alice.
Follow Bob.

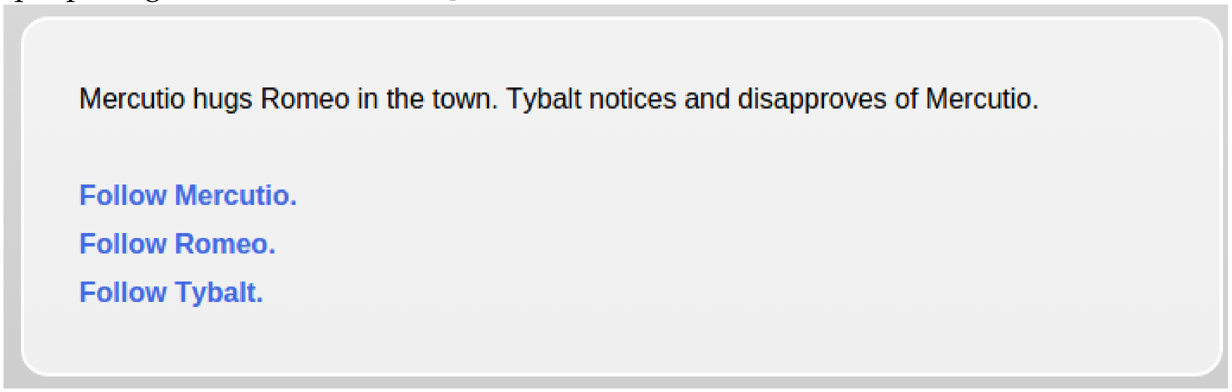
The story world author provides a *scene file* in addition to the Celf program that will generate the simultaneous narrative; the scene file provides the text to go along with the abstract story specifications. For example, here is an excerpt of the scene file for our Shakespearean tragedy world:

```
:: 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 mockingly, 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:


Mercutio hugs Romeo in the town. Tybalt notices and disapproves of Mercutio.

Follow Mercutio.
Follow Romeo.
Follow Tybalt.

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 doing 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.

Chapter 4

Ceptre: A Linear Logic Language for Interactive Programs

4.1 Background

At the writing of this thesis, game designers and developers have a wealth of tools available for creating executable prototypes. Freely-available engines such as Twine [KA08], Unity [Hig10], PuzzleScript [Lav13], and Inform 7 [Nel01] provide carefully-crafted interfaces to creating different subdomains of games. On the other hand, to invent interesting, novel mechanics in any of these tools typically requires the use of ill-specified, general purpose languages: in Twine, manipulating state requires dipping into JavaScript; in Unity, specifying any interactive behavior requires learning a language such as C# or JavaScript; Inform 7 contains its own general-purpose imperative, functional, and logic programming languages; and PuzzleScript simply prevents the author from going outside the well-specified 2D tile grid mechanisms.

The concept of *operational logics* [MWF09], the underlying substrate of meaningful state-change operations atop which mechanics are built, was recently proposed as a missing but critical component of game design and analysis. In most prototyping tools, there is a fixed operational logic (e.g. tile grids and layers in PuzzleScript, or text command-based navigation of a space in Inform 7) atop which the designer may be inventive in terms of *rules* and *appearance*, but all rules must be formed out of entities that mean something in the given operational logic (or use a much more complex language to subvert that default logic). We lack good specification tools for inventing *new* operational logics, or combining elements from several. In particular, many novel systems of play arise from the investigation of interactive storytelling [MS03], multi-agent social interaction [MTS$^+$10], and procedurally generated behavior [HZDR11], and no existing tools make it especially straightforward for a novice developer to codify and reason about the underlying logics of those systems.

Ceptre is a proposal for an operational-logic-agnostic specification language that may form the basis of an accessible front-end tool. Specifically, it embodies the following language design goals:

1. Has a logical foundation (correspondence to proof search) for the sake of clear, portable, and extensible semantics
2. Is general enough to describe a wide range of operational logics
3. Can describe both conjunctive and disjunctive narrative structure and support causal reasoning on the basis of that structure
4. Has the potential for accessible front-end tools to be built atop it

In the remainder of this chapter, we explain the features of the language by describing how to make the story world example from Chapter 3 into an interactive simulation, describe the language in full, and relate it to Celf to provide logical justification. In Chapter 5, we showcase the ability to support a wide range of operational logics by providing several example encodings as case studies. To support the "potential for accessibility" claim, we relate Ceptre to Inform 7 [Nel01], Kodu [Mac11], and PuzzleScript [Lav13]. We identify an essential idiomatic core to each of these languages, formalize it, and show how it can be expressed in Ceptre. What this work suggests is that each encoding could be packaged as a separate module or context *within Ceptre* – in which a novice designer could work exclusively, while the wider affordances of the language remain available to extend, combine, or (re)invent such models.

To summarize, the core contributions described in this chapter are (1) progress toward a computational and conceptual framework for creating games in a high-level, rule-based way without constraint by a particular underlying world logic; and (2) an account of several existing rule-based design frameworks in terms of this core language.

4.1.1 New Syntax

The following examples will introduce Ceptre through its concrete syntax, which differs subtly from the Celf notation we used in Chapter 3 for forward-chaining rules. In particular:

1. Forward-chaining, linear rules of the form `A -o {B}` are now written `A -o B`, because linear transitions are strictly forward-chaining.
2. We introduce the syntax `$A -o B` as sugar for `A -o A * B` to reduce the amount of duplication required for premises that are consumed and re-produced.

For example, the rule `do/compliment/private` from Chapter 3, which was originally written

```
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}.
```

can now be written

```
do/compliment/private
: $at C L * $at C' L * $philia C C' -o philia C' C.
```

Additionally, we introduce syntax for declaring types and predicates:

- Type declarations have the form `t : type`, where `t` is an identifier chosen by the programmer for the new type being defined.

- Predicate and term declarations have form `p t1 t2 : X` (generalized to an arbitrary number of arguments `t`), where `p` is the new predicate or term being defined, each `t` is the type of an argument to the predicate, and `X` is either `pred`, `bwd`, or a type `t'` previously defined with `t' : type`.
- The keywords `pred` and `bwd` represent linear resources and persistent facts defined by backward chaining, respectively.

As an example, we can define a set of colors, the unary natural numbers, and a predicate relating colors to numbers, as follows:

```
color : type.
red : color.
blue : color.

nat : type.
zero : nat.
succ nat : nat.

count color nat : pred.
```

Deeper language differences will be explained in the next sections, and the language is summarized comprehensively in Section 4.5.

4.2 Stages and Interactivity

Ceptre is, at its core, a restriction of Celf to forward-chaining linear rules (and, secondarily, backward-chaining persistent rules). The most notable addition is that Ceptre offers a mechanism to replace the default random nondeterminism with a choice from an external source, e.g. standard input. The syntax for adding interaction is to wrap all of the rules in a *stage*, or a named set of rules, then add an `#interactive` directive.

```
stage allrules = {
  do/compliment private : [...]
  [...]
}
#interactive allrules.
```

Annotating the stage with this directive means that every time at least one transition (rule applied to ground term arguments) can fire, all applicable transitions are shown as choices printed to standard output, and a choice between them may be entered through standard input.

We also include a `#trace` directive, similar to Celf's, that requires an initial stage and an initial context (rather than an initial predicate). For instance, the initial context for the story world example would be written as

```
context init =
{ 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,

  anger montague capulet, anger capulet montague,
  anger tybalt romeo, anger capulet romeo,
  anger montague tybalt,

  likes mercutio romeo, likes romeo mercutio,
  likes montague romeo, likes capulet juliet,

  has tybalt weapon, has romeo weapon, has apothecary weapon }.
```

And we invoke the program starting with the `allrules` stage and the initial state `init` with the directive `trace _ init allrules`. (The second argument is a numeric bound on the number of proof steps to take, or _ for no bound.)

Upon running this program, a user is first prompted with these choices, corresponding to all rules that apply in the initial state:

```
0: (do/insult/witnessed tybalt town romeo mercutio)
1: (do/insult/private tybalt town romeo)
2: (do/compliment/witnessed mercutio town romeo tybalt)
3: (do/compliment/private romeo town mercutio)
4: (do/compliment/private mercutio town romeo)
5: (do/formOpinion/dislike mercutio town juliet)
6: (do/formOpinion/dislike juliet town mercutio)
7: (do/formOpinion/like mercutio town juliet)
8: (do/formOpinion/like juliet town mercutio)
9: (do/travelTo montague romeo town mon_house)
10: (do/travelTo capulet juliet town cap_house)
11: (do/travelTo juliet nurse cap_house town)
12: (do/travelTo nurse juliet town cap_house)
```

By default, these choices are printed to the screen, followed by a prompt for a numeric selection. New choices are generated based on the prior selection and its change to the state. The state as it evolves is written to a log file, which can be read in a separate text buffer to inform player decisions. [1]

As an interactive storytelling system, this program has some peculiar features: for instance, it appears that we give the player control over not just characters' actions but also their *reactions*, which we would prefer to think of as involuntary, such as the `do/formOpinion` rules. We would like for some of these rules to run automatically without player intervention. In our next iteration of the program, we will use stages to transfer control between two components, an interactive one and a programmatic one: essentially, the player "takes turns" with the computer to build the story.

Stages compute until *quiescence*, at which point control may be transfered to another stage. We use outer-level rules of the form

[1]Ideally, these input and output mechanisms would be hooked up to a more intelligible, customizable user interface. We have attempted to make the implementation modular so that these mechanisms may be replaced in future work. One simple candidate would be to replace the transition print-outs with the legible scene descriptions described in Section 3.5.

```
qui * stage S [* ...] -o stage S' [* ...].
```

where [* ...] is meta-syntax denoting any additional tensored-together resources. The rule denotes, "At quiescence, if stage S is active (and potentially some additional state), transition to stage S' (and potentially remove/add some additional state)." The only new piece of syntax here is the `qui` predicate, which is a special token denoting quiescence of the program. All outer-level rules must have this form: upon quiescence, replace one `stage` resource with another (and possibly delete or add some additional state).

Concretely, the structure of our new program will be:

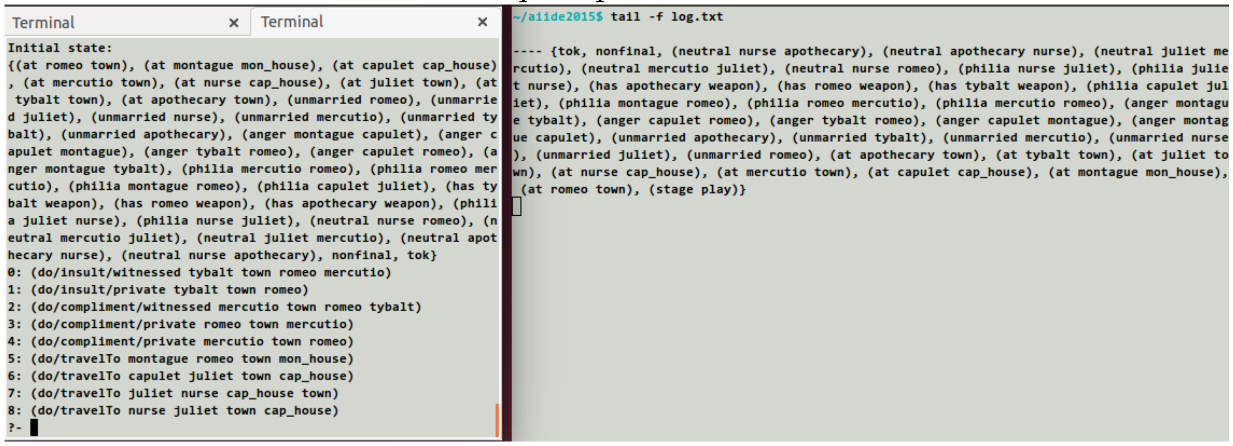
```
stage act = {
  % User-selected rules.
}
qui * stage act -o stage react.
stage react = {
  % Involuntary, reactive rules.
}
qui * stage react -o stage act.
```

In this formulation of the game, the player may act arbitrarily many times before reactions are calculated (and in fact must do so until quiescence of the act stage). We can slightly modify the program to enjoy a turn-taking idiom instead, offering one turn of input between potentially many steps of reactive computation:

```
tok : pred.

stage play = {

do/travelTo
: tok * $likes C C' * $at C' L' * at C L * accessible L L'
  -o at C L'.

  ...
}

#interactive play.
qui * stage play -o stage auto.

stage auto = {
  ...
}

qui * stage auto -o stage play * tok.
```

Each rule in the stage is edited to accept one additional premise `tok` (for "token"), which is generated whenever control is transfered to the stage. Turn tokens may be introduced to the reactive side of the program as well, if we wish to carry out a bounded number of reactive steps before returning control to the player. An example of this interface given by running Ceptre from a command prompt (and separately tracking the log file that depicts the evolving context) can be seen in Figure 4.1.

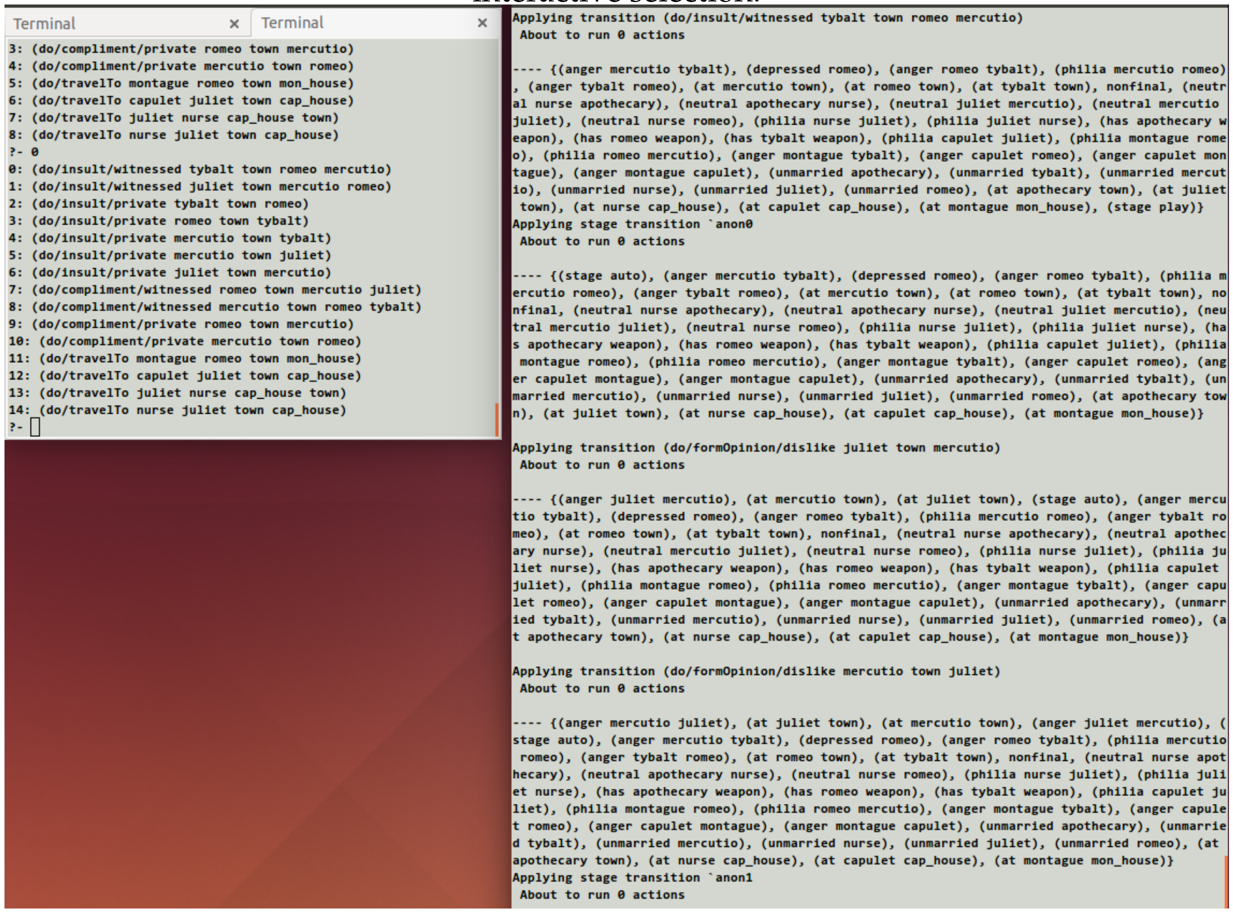
Figure 4.1: Screenshots of command-line interaction with Ceptre.
Initial prompt and context:



```
Initial state:
{(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), (unmarried romeo), (unmarrie
d juliet), (unmarried nurse), (unmarried mercutio), (unmarried ty
balt), (unmarried apothecary), (anger montague capulet), (anger c
apulet montague), (anger tybalt romeo), (anger capulet romeo), (a
nger montague tybalt), (philia mercutio romeo), (philia romeo mer
cutio), (philia montague romeo), (philia capulet juliet), (has ty
balt weapon), (has romeo weapon), (has apothecary weapon), (phili
a juliet nurse), (philia nurse juliet), (neutral nurse romeo), (n
eutral mercutio juliet), (neutral juliet mercutio), (neutral apot
hecary nurse), (neutral nurse apothecary), nonfinal, tok}
0: (do/insult/witnessed tybalt town romeo mercutio)
1: (do/insult/private tybalt town romeo)
2: (do/compliment/witnessed mercutio town romeo tybalt)
3: (do/compliment/private romeo town mercutio)
4: (do/compliment/private mercutio town romeo)
5: (do/travelTo montague romeo town mon_house)
6: (do/travelTo capulet juliet town cap_house)
7: (do/travelTo juliet nurse cap_house town)
8: (do/travelTo nurse juliet town cap_house)
?-
```

```
~/aiide2015$ tail -f log.txt

---- {tok, nonfinal, (neutral nurse apothecary), (neutral apothecary nurse), (neutral juliet me
rcutio), (neutral mercutio juliet), (neutral nurse romeo), (philia nurse juliet), (philia julie
t nurse), (has apothecary weapon), (has romeo weapon), (has tybalt weapon), (philia capulet jul
iet), (philia montague romeo), (philia romeo mercutio), (philia mercutio romeo), (anger montagu
e tybalt), (anger capulet romeo), (anger tybalt romeo), (anger capulet montague), (anger montag
ue capulet), (unmarried apothecary), (unmarried tybalt), (unmarried mercutio), (unmarried nurse
), (unmarried juliet), (unmarried romeo), (at apothecary town), (at tybalt town), (at juliet to
wn), (at nurse cap_house), (at mercutio town), (at capulet cap_house), (at montague mon_house),
 (at romeo town), (stage play)}
```

Prompt (left) and transition selection trace and context (right) after making one interactive selection:



```
3: (do/compliment/private romeo town mercutio)
4: (do/compliment/private mercutio town romeo)
5: (do/travelTo montague romeo town mon_house)
6: (do/travelTo capulet juliet town cap_house)
7: (do/travelTo juliet nurse cap_house town)
8: (do/travelTo nurse juliet town cap_house)
?- 0
0: (do/insult/witnessed tybalt town romeo mercutio)
1: (do/insult/witnessed juliet town mercutio romeo)
2: (do/insult/private tybalt town romeo)
3: (do/insult/private romeo town tybalt)
4: (do/insult/private mercutio town tybalt)
5: (do/insult/private mercutio town juliet)
6: (do/insult/private juliet town mercutio)
7: (do/compliment/witnessed romeo town mercutio juliet)
8: (do/compliment/witnessed mercutio town romeo tybalt)
9: (do/compliment/private romeo town mercutio)
10: (do/compliment/private mercutio town romeo)
11: (do/travelTo montague romeo town mon_house)
12: (do/travelTo capulet juliet town cap_house)
13: (do/travelTo juliet nurse cap_house town)
14: (do/travelTo nurse juliet town cap_house)
?-
```

```
Applying transition (do/insult/witnessed tybalt town romeo mercutio)
 About to run 0 actions

---- {(anger mercutio tybalt), (depressed romeo), (anger romeo tybalt), (philia mercutio romeo)
, (anger tybalt romeo), (at mercutio town), (at romeo town), (at tybalt town), nonfinal, (neutr
al nurse apothecary), (neutral apothecary nurse), (neutral juliet mercutio), (neutral mercutio 
juliet), (neutral nurse romeo), (philia nurse juliet), (philia juliet nurse), (has apothecary w
eapon), (has romeo weapon), (has tybalt weapon), (philia capulet juliet), (philia montague rome
o), (philia romeo mercutio), (anger montague tybalt), (anger capulet romeo), (anger capulet mon
tague), (anger montague capulet), (unmarried apothecary), (unmarried tybalt), (unmarried mercut
io), (unmarried nurse), (unmarried juliet), (unmarried romeo), (at apothecary town), (at juliet
 town), (at nurse cap_house), (at capulet cap_house), (at montague mon_house), (stage play)}
Applying stage transition `anon0
 About to run 0 actions

---- {(stage auto), (anger mercutio tybalt), (depressed romeo), (anger romeo tybalt), (philia m
ercutio romeo), (anger tybalt romeo), (at mercutio town), (at romeo town), (at tybalt town), no
nfinal, (neutral nurse apothecary), (neutral apothecary nurse), (neutral juliet mercutio), (neu
tral mercutio juliet), (neutral nurse romeo), (philia nurse juliet), (philia juliet nurse), (ha
s apothecary weapon), (has romeo weapon), (has tybalt weapon), (philia capulet juliet), (philia
 montague romeo), (philia romeo mercutio), (anger montague tybalt), (anger capulet romeo), (ang
er capulet montague), (anger montague capulet), (unmarried apothecary), (unmarried tybalt), (un
married mercutio), (unmarried nurse), (unmarried juliet), (unmarried romeo), (at apothecary tow
n), (at juliet town), (at nurse cap_house), (at capulet cap_house), (at montague mon_house)}

Applying transition (do/formOpinion/dislike juliet town mercutio)
 About to run 0 actions

---- {(anger juliet mercutio), (at mercutio town), (at juliet town), (stage auto), (anger mercu
tio tybalt), (depressed romeo), (anger romeo tybalt), (philia mercutio romeo), (anger tybalt ro
meo), (at romeo town), (at tybalt town), nonfinal, (neutral nurse apothecary), (neutral apothec
ary nurse), (neutral mercutio juliet), (neutral nurse romeo), (philia nurse juliet), (philia ju
liet nurse), (has apothecary weapon), (has romeo weapon), (has tybalt weapon), (philia capulet 
juliet), (philia montague romeo), (philia romeo mercutio), (anger montague tybalt), (anger capu
let romeo), (anger capulet montague), (anger montague capulet), (unmarried apothecary), (unmarr
ied tybalt), (unmarried mercutio), (unmarried nurse), (unmarried juliet), (unmarried romeo), (a
t apothecary town), (at nurse cap_house), (at capulet cap_house), (at montague mon_house)}

Applying transition (do/formOpinion/dislike mercutio town juliet)
 About to run 0 actions

---- {(anger mercutio juliet), (at juliet town), (at mercutio town), (anger juliet mercutio), (
stage auto), (anger mercutio tybalt), (depressed romeo), (anger romeo tybalt), (philia mercutio
 romeo), (anger tybalt romeo), (at romeo town), (at tybalt town), nonfinal, (neutral nurse apot
hecary), (neutral apothecary nurse), (neutral nurse romeo), (philia nurse juliet), (philia juli
et nurse), (has apothecary weapon), (has romeo weapon), (has tybalt weapon), (philia capulet ju
liet), (philia montague romeo), (philia romeo mercutio), (anger montague tybalt), (anger capule
t romeo), (anger capulet montague), (anger montague capulet), (unmarried apothecary), (unmarrie
d tybalt), (unmarried mercutio), (unmarried nurse), (unmarried juliet), (unmarried romeo), (at 
apothecary town), (at nurse cap_house), (at capulet cap_house), (at montague mon_house)}
Applying stage transition `anon1
 About to run 0 actions
```

At this point, the simulation may be fine-tuned to create the interactive experience desired by the author. For example, some rules may be modified not to require certain emotional states as premises, instead allowing agency on the part of the player to play that decision-making role. Thus concludes our first example of modeling an interactive, generative scenario in Ceptre.

4.3 Staged Logic Programming

Stages fundamentally give us the ability to *program with quiescence*. Above we sketched an example of their use for alternating automatic and human-mediated computation, but they can also be used for purely automatic computations that pure linear logic programming cannot express. For example, suppose our initial context is a bin of blue and red balls, and we want to write a program that exchanges blue for red, red for blue, and finally counts the number of balls of each color. Conceptually, there are just three rules,

```
red-to-blue : ball red -o ball blue.
blue-to-red : ball blue -o ball red.
count : count Color Number * ball color -o count Color (succ Number).
```

...but these rules interfere nondeterministically, such that computations are possible that produce the wrong answer, such as those that convert all red balls to blue and then count the blue balls before finishing their conversion. We can write this program correctly using three stages (and a temporary predicate):

```
temp : pred.
stage mark_blues = {
  blue-to-temp
  : ball blue -o temp.
}
do/flip_reds : qui * stage mark_blues -o stage flip_reds.

stage flip_reds = {
  red-to-blue
  : ball red -o ball blue.
}
do/flip_temps : qui * stage flip_reds -o stage flip_temps.

stage flip_temps = {
  temp-to-blue
  : temp -o ball red.
}
do/count : qui * stage flip_temps -o stage count.

stage count = {
  count_a_ball
  : count Color Number * ball Color
```

```
    -o count Color (succ Number).
}
```

By separating the computation into stages, such that certain rules are only accessible once others have quiesced, we in effect create an ordering among the rules and permit a greater range of computations to be expressed.

Next, we illustrate how to program a few common idioms using stages. Linear logic programming without stages has a number of limitations as a usable language. For instance, it can only refer to specific, positive instances of predicates, and rules may not be imbued with any particular relative priority ordering. While those limitations are a necessary facet of the direct correspondence with logic, stages break that correspondence at an abstraction boundary that makes certain conveniences possible while maintaining the ability to reason about a program logically within a given stage.

4.3.1 Rule Ordering

By default, rules within a stage are not ordered with respect to one another: when multiple rules apply to a state, there is no way to designate which should be prioritized. We deliberately do not introduce a primitive rule-ordering construct because it takes away the *locality* of reasoning about a program: one rule's meaning would depend on the meaning of all rules with higher priority, rather than being understandable in isolation. Rule priorities in general are *anti-compositional* in the sense that global program meaning cannot be broken down into smaller parts. However, with stages as a program structuring tool, we can actually recover rule ordering in a way that, in some sense, reveals its complexity.

Let us call an *ordered stage* one whose rules are given priority corresponding to their order in the program: only when rule i fails will rule $i + 1$ be considered. We can use stages to encode this ordering by placing each rule in its own stage, transfering control to the next rule's stage only when a certain premise has not been discharged. For instance, suppose we wanted to represent an ordered stage of the form

```
ordered stage OS = {
  r1 : S1 -o S1'.
  r2 : S2 -o S2'.
  ...
  rn : Sn -o Sn'.
}
qui * stage OS -o stage Next.
```

We can elaborate it as the following multi-stage program, using a similar idiom to the one we used to encode negation, adding to each rule a premise `no` to represent negative information (i.e. failure of any prior rule to take effect) and a consequent `yes` to represent success of a rule:

```
stage OS1 = {
  r1 : no * S1 -o S1' * yes.
}
```

```
qui * stage OS1 * no -o stage OS2.
qui * stage OS1 * yes -o stage OS1 * no.

stage OS2 = {
  r2 : no * S2 -o S2' * yes.
}
qui * stage OS1 * no -o stage OS3.
qui * stage OS1 * yes -o stage OS1 * no.
[...]
stage OSn = {
  rn : no * Sn -o Sn' * yes.
}
qui * stage OSn * no -o stage Next.
qui * stage OSn * yes -o stage OS1 * no.
```

This elaboration suggests that adding syntactic sugar to the language for designating a given stage as "ordered" would do little harm due to stages themselves providing a compositional unit of meaning, which un-ordered stages retain the property of local reasoning on rules.

4.3.2 Negation

To encode negation (absence) of a predicate `p`, which we might otherwise write `not p`, we can seed a stage with a distinct atom that is consumed by a rule only when `p` is present, thereby turning a negative check into a positive one. For example, to encode `not p -o q`, we do the following:

```
p : pred.
q : pred.
no_p_yet : pred.
notp : pred.

stage check_p = {
find : no_p_yet * p -o p.
}.
qui * stage check_p * no_p_yet -o stage do_q * notp.
qui * stage check_p * $p -o stage done.

stage do_q = {
  rule : notp -o q.
}

stage done = {}

context ff = {no_p_yet}.
context tt = {no_p_yet, p}.
```

```
#trace _ check_p ff.
#trace _ check_p tt.
```

The `check_p` stage consumes the `no_p_yet` predicate if `p` is in fact present, so that the rules that fire upon that stage's quiescence can branch, adding `notp` to the context if `no_p_yet` was never consumed.

4.3.3 Finite Set Comprehension

A common pattern that comes up in logic programming over indexed predicates is to do something *once per index*, when these are drawn from a finite set. For instance, in the story world, rather than presenting two rules for an interaction corresponding to whether it is observed or not— one which involves another party in the same room and one that does not— we might like to implement a *broadcast* action: once an action takes place, generate some fact that propagates to *every character in the same room*. In general, since types like `character` can be infinite, such a feature would not make sense, but we can implement it in a limited way with stages. In particular, we have linear resources corresponding to each character in a unique way, via the `at` predicate. So we can implement a two-stage program that first replaces all relevant instances of these resources with some new predicate ranging over the same indices, then replaces that predicate with the old one plus some additional information.

This trick is implemented below.

```
message : type.
says character message : pred.
knows character message : pred.

broadcast location message : pred.

stage say = {
  say : says C M * $at C L -o broadcast L M.
}
qui * stage say -o stage hear.

restore character location : pred.

stage hear = {
  hear : $broadcast L M * at C L -o knows C M * restore C L.
}
qui * stage hear -o stage restore.

stage restore = {
  restore : restore C L -o at C L.
}

alice : character.
bob : character.
```

```
charlie : character.
hall : location.
foyer : location.
hi : message.

context init =
{at alice hall, at bob hall, at charlie foyer, says alice hi}.

#trace _ say init.
```

We note that this trick is particularly inefficient and unwieldy compared to something like a form of quantification or direct set comprehension and suggest that a future version of the language might include a primitive, or at least syntactic sugar, for this idiom. The Meld programming language (and its linear-logic based analog, Linear Meld) includes a similar primitive for writing logic programs over graph structures [ARLG$^+$09, CRGP14].

4.3.4 Summarization

Finally, we illustrate a use of stages to transform a complex Celf program, which uses many predicates to represent invariants for essentially tracking stage information, into a much simpler staged logic program. We characterize this transformation as a form of *summarization*, or collecting information to quiescence for the sake of working with summarized information (such as a total quantity) in the next stage.

For example, in the use of linear logic to implement voting protocols [DS12], even the simplest "first-past-the-post" protocol has two stages: total all the ballots, the find the candidate with the maximum number. In the Celf implementation, stage information is tracked with the predicates `count-ballots` and `determine-max`. They take care only to transfer control from one stage to the next by maintaining a counter that, by invariant, must always equal the number of remaining (uncounted) ballots. Then, when the counter reaches zero, the predicate corresponding to the next stage is introduced. Below are the predicates and rules in this original Celf program.

```
uncounted-ballot : candidate -> type.
hopeful : candidate -> nat -> type.
defeated : candidate -> type.
elected : candidate -> type.
count-ballots : nat -> nat -> type.
determine-max : nat -> type.


%% FPTP axioms.

count/run : count-ballots (s U) H *
            uncounted-ballot C *
            hopeful C N
```

```
                -o {hopeful C (s N) *
                    count-ballots U H}.

count/done : count-ballots z H
                 -o {determine-max H}.

max/run : determine-max (s H) *
          hopeful C N *
          hopeful C’ N’ *
          !nat-lesseq N’ N
            -o {hopeful C N *
                !defeated C’ *
                determine-max H}.

max/done : determine-max (s z) *
           hopeful C N
             -o {!elected C}.
```

The programmer carefully maintains the invariant that the first index of `count-ballots U H` matches the number of `uncounted-ballot` instances in the program, and its second index matches the number of `hopeful` instances in the program. In Ceptre, this invariant is replaced by the ability to program with quiescence by stratification into stages.

```
stage count = {
  count_ballot : ballot C * hopeful C N -o hopeful C (s N).
}

- : qui * stage count -o stage pick * max z.

stage pick = {
  increase : max N * hopeful C N’ * lt N N’
               -o hopeful C N’ * max N’.
  eliminate : max N * hopeful C N’ * lt N’ N
               -o max N * !defeated C.
}
- : qui * stage pick * hopeful C _ -o !elected C.
```

A full code listing for this example is available in Appendix C.1.

4.4 Sensing and Acting Predicates

Interactive stages accept input at every rule selection step in a given stage, allowing the latent nondeterminism of the program to be mediated by the actor’s choices. By default, all program steps that are logically possible in the model are revealed to the external source—no more and no less. This design corresponds to the *branching story* interpretation of linear logic programs as described by Bosser [BCC10], and for interactive fiction, it can be seen as a sort of hypertext realization of a story model.

However, sometimes it is not desirable to expose every available choice to the player. In many instances of parser interactive fiction, for example, a large aspect of play is in discovering the *space* of reasonable actions with which to navigate the game—a mode of play afforded by its user interface, a simple command prompt at which the player types short imperative phrases.

To allow for a larger space of interaction models such as this one, we borrow an abstraction from the Meld language [ARLG$^+$09]: *sensing* and *acting* predicates. These predicates mediate input and output, respectively, to or from the runtime environment.

In our story-world model, we can declare a new term for actions and a new predicate `do` indicating player intent: `do <verb>`, where verb is an index type enumerated by the program and corresponding to all player-controlled commands. Now we add this predicate, indexed by the appropriate verb, on the left-hand side of every rule; for example

```
do/murder
: do (murder C C') *
  anger C C' * anger C C' * anger C C' * anger C C' *
  $at C L * at C' L  * $has C weapon
                -o !dead C'.
```

Action predicates may be used for output: for example, reporting that this rule successfully fired might involve a new predicate `report status verb : action` and the rule could be rewritten as

```
do/murder : ...  -o !dead C' * report success (murder C C').
```

The implementation of the action predicate is specified by an extensible part of the language runtime and is not part of the language definition. For instance, the report predicate could correspond to printing a particular string to standard output.

Finally, in order to coordinate the two halves of the program—one that waits for player input, and the other to process that input—we divide the program into stages as follows:

```
stage process_verb = {
  rulename : do A * ...
             -o ... * report success A.
  ...
}
qui * stage process_verb -o
  stage accept_verb * wait.

stage accept_verb = {
  listen : wait * input Verb -o do Verb.
}
% loop until verb is successfully received
qui * stage accept_verb * wait
  -o stage accept_verb * wait.
```

```
qui * stage accept_verb * $do Verb
  -o stage process_verb.
```

To tie the predicate do to a sensor for player commands and the predicate report to an action for printing command results, we provide corresponding directives:

```
#sensor input _BUILTIN_IF_COMMAND.
#action report _BUILTIN_IF_REPORT.
```

Reading the program rules in order, they say that when the process stage has quiesced, move to the accept stage along with a wait token. Then, if the accept stage finishes without consuming the wait token, return to the accept stage. Finally, if the accept stage quiesces and emits a verb intent (do Verb), go into the process stage and process the verb. In other words, this program implements a simple blocking input loop.

Note that we are not guaranteed that every do predicate issued by the player will result in a successful action. A character won't be able to murder someone unless they also have the required quantity of anger, for example. This problem of *coverage* of the possible action space comes up in all rule-based action systems, and it can be handled in several possible ways:

1. *Total coverage:* Require (and check) that the program satisfy coverage, i.e. that for every action A there is a rule that will consume do A.
2. *Failure feedback:* Check after-the-fact whether the action was consumed in rule processing, via quiescence rules, and enter a second "failure" pass, which then may offer feedback to the player about the reason for failure. This approach approximates Inform 7's rule processing, which we discuss further in Section 4.6.3.
3. *Intent handling:* Model player input not as effectful *actions* but effectless *intents*, which are then pre-processed to determine the effects that really take place, all of which should be expected to succeed. This method more closely matches PuzzleScript's approach, discussed in Section 4.6.2.

Failure feedback and intent handling can both be implemented via the stage system; total coverage would require a meta-program check that we believe would be feasible to implement, but is not currently part of Ceptre's design.

4.5 Language Definition

We have now established Ceptre's mechanisms—predicates, rules, sensing and acting predicates, and stages—through examples. In this section we give a definition of the language semantics, relating it to proof search in linear logic via a compilation to Celf.

We have three different base kinds: type for declaring new forms of data (such as characters, locations, numbers, and lists), pred for declaring predicates that may be generated by forward-chaining rules, and bwd for declaring predicates that may be defined by backward-chaining rules (and referred to in the premises of forward-chaining rules). All of these simply get compiled to type in the Celf interpretation and are used purely for the syntactic restrictions we wish to impose. Grammatically the kinds are specified as follows:

$$
\begin{array}{rcl}
P & ::= & \mathsf{pred} \mid \mathsf{bwd} \mid \tau \to P \\
K & ::= & \mathsf{type} \mid P
\end{array}
$$

The abstract syntax $a : \tau_1 \to \ldots \tau_n \to \mathsf{pred}$ stands for the concrete syntax `a t1 ... tn : pred` (where `ti` is the concrete syntax corresponding to τ_i), and similarly for bwd predicates and terms (described below).

All constant declarations of forward-chaining prediates, backward-chaining predicates, and types will be denoted with the metavariable a.

We have a simple term language that allows the definition of inductively defined types in the type kind. Term constants are denoted with the metavariable c.

$$
\begin{array}{rcl}
\tau & ::= & a \mid \tau \to \tau \\
t & ::= & c \mid t\, c
\end{array}
$$

So for example, we can define natural numbers and lists:

```
nat : type.
z : nat.
s nat : nat.

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

In Ceptre syntax,

```
cons nat list : list
```

gets interpreted as

```
cons : nat -> list -> list
```

Backward chaining rules take the following form:

$$
\begin{array}{rcl}
p & ::= & a\, t \ldots t \\
B & ::= & p \mid p \to B
\end{array}
$$

We will typically write $p \to q$ in ASCII using the backwards-arrow notation `q <- p`. So for example, a declaration of natural number addition looks like:

```
plus nat nat nat : bwd.
plus/z : plus z N N.
plus/s : plus (s N) M (s P)
       <- plus N M P.
```

This notation is similar to how such a predicate would look in Prolog or LF.

Linear, forward chaining rules A take the following form:

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

In concrete syntax, the tensor symbol $\otimes$ is spelled `*`, $\multimap$ is spelled `-o`, and 1 is spelled `()`. All capitalized identifiers are interpreted as logic variables Π-quantified over at the outside of the rule. For instance, `ballot C * hopeful C N  -o hopeful C (s N)` is written formally as $\Pi c{:}\text{cand}.\Pi n{:}\text{nat}.\text{ballot}\ c \otimes \text{hopeful}\ c\ n \multimap \text{hopeful}\ c\ (s\ n)$.

Stage contents Σ and stage declarations are defined as:

$$\begin{array}{rcl} \Sigma & ::= & \cdot \mid \Sigma, r : A \\ \mathit{stagedecl} & ::= & \mathsf{stage}\ \phi = \{\Sigma\} \end{array}$$

Linking rules L are the only forward-chaining rules that can appear outside of a stage, and they obey the grammar

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

where qui is a distinguished predicate for quiescence.

Linear contexts Δ and context declarations are defined as:

$$\begin{array}{rcl} \Delta & ::= & \cdot \mid \Delta, x : S \\ \mathit{ctxdecl} & ::= & \mathsf{context}\ C = \{\Delta\} \end{array}$$

With the syntax declared so far, which includes stage declarations but no means of transfering control between stages, we can take a program to be a collection of declarations

$$\Xi \quad ::= \quad \cdot \mid \Xi, c : \tau \mid \Xi, a : K \mid \Xi, b : B \mid \Xi, \mathit{stagedecl} \mid \Xi, \mathit{ctxdecl} \mid \Xi, r : L$$

We can now give meaning to a small example in terms of CLF. The following example consists of two stages, one that replaces all instances of `red` with `blue` and one that counts the number of `red` and `blue` instances.

```
red : type.
blue : type.

nat : type.
z : nat.
s nat : nat.

rcount nat : pred.
bcount nat : pred.

context init = {rcount z, bcount z}.

stage change = {
  change : red -o blue.
}
```

```
stage count = {
  count_red  : rcount N * red  -o {rcount (s N)}.
  count_blue : bcount N * blue -o {bcount (s N)}.
}
```

The translation of this example into CLF works simply by stripping out the stage declarations and adding a stage-tracking predicate to each side of every rule:

```
st : type.
change : st.
count : st.

stage : st -> type.

% program-specific type families.

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

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

As written, the two subprograms are isolated, and we have not yet provided a top-level to determine which stage starts or when it is appropriate to change stages. In general, we want the ability to program *behavior at quiescence*. In Ceptre source programs, we have a runtime-generated predicate `qui`, which pops into existence at global quiescence of the program, allowing us to write global rules like

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

We want to give the programmer access to such a predicate in a controlled way—it only makes sense on the left of a transition, for instance, and we expect rules using it to maintain an invariant pertaining to the stage predicate: we want to always consume a stage token and replace it with another one. That is, exactly one stage is always active. This property is enforced by the grammar for linking rules, i.e.

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

The quiescence token qui does not exist in Celf, but a particular quirk of the semantics of the forward chaining monad $\{A\}$ turns out to give us the behavior we need: Celf, when searching for a proof of $A \multimap \{B\}$, ignores the goal proposition B until quiescence. Thus Celf actually has the exact same crack in its correspondence with logical provability that Ceptre has: valid proofs of $A \vdash \{B\}$ in linear logic that refer to the right-hand side of the sequent before all left rules have been tried will not be discovered by Celf. By mixing forward- and backward-chaining linear rules in Celf, therefore, we can implement Ceptre's semantics.

Specifically, a backward-chaining rule of the form

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

can be operationally described as populating the context with S, running all forward-chaining rules to quiescence, and then searching for a proof of S'. 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 stages together at quiescence in the following way:

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

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

```
qui * stage change -o {stage count}.
```

can be compiled to Celf as two rules:

```
r1 : run change
      o- (stage change -o {run count}).
r2 : run count
      o- stage change
      o- (stage count -o {stage count * rcount R * bcount B}).
```

So far, this only demonstrates how to transfer control unilaterally from one stage 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" stage) on unparseable input, rather than moving on to the evaluation stage.

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 stage.

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

For every `P` for which there are rules of the form

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

add rules

```
- : run P o- (stage 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 interpretation we can fully describe the expressive structure of multi-stage programs, including looping and branching patterns.

The remainder of the language semantics is described by the linear logic proof search operationalization given in Chapter 3. The typing rules for terms and predicates are standard, and are fully described in Appendix B. This concludes our definition of the language semantics.

4.6 Interpreting Other Systems

Ceptre can be understood as an underlying framework in which more specialized, simpler authoring tools can be embedded. To illustrate its generality, we show how to recover the central idioms of three successful rule-based, operational-logic-specific tools: Kodu, PuzzleScript, and Inform 7. (See Section 4.1 for citations of publications and documentation for these tools.)

4.6.1 Kodu

Kodu is a rule-based programming language designed for children to make movement-based games on an Xbox 360. The language takes inspiration from behavior-based robotics, which closely reflects Ceptre's "sensing and acting" model described above: the language includes *sensors*, *filters*, *selectors*, *actuators*, and *modifiers*. A rule is simply one element of each of these categories, some of which are optional, associated with a specific game entity (such as the player character). For example, a rule directing an entity to move quickly toward green objects is:

```
see - green - move - towards - quickly
```

In this rule, *see* is a sensor, *green* is a filter, *move* is an actuator, *toward* is a selector, and *quickly* is a modifier (according to the formal grammar given by Stolee [Sto10]).

Interpreting Kodu programs in Ceptre is essentially a matter of interpreting sensors and selectors as sensing predicates, actuators as acting predicates, filters as ordinary predicates, and all the rest as term-level arguments to these predicates. Additionally, instead of explicitly associating rules with a single actor, we index each predicate with an actor term. Thus the above rule can be written in Ceptre as follows:

```
see X Y * $property Y green -o move X Y toward quickly.
```

The type header and runtime bindings associated with the predicates in this rule are as follows:

```
see entity entity : pred.
property entity filter : pred.
move entity entity selector modifier : pred.

#sensor see  _BUILTIN_KODU_SEE.
#action move _BUILTIN_KODU_MOVE.
```

One of the observations one can make in the collections Kodu examples presented in papers is that, with certain exceptions, they *come in pairs*: a rule whose actuator is movement, for example, can be thought of as connecting to (or causing) a rule whose sensor involves location, such as `bump`, which triggers when the actor is right next to something described by the filter. For instance, an actor that moves toward green things and eats them could be modeled by pairing the above rule with

```
bump - green - eat
```

Or in Ceptre:

```
bump X Y * $color Y green -o eat X Y.
```

One of the things that gets in the way of reasoning about these specifications in Kodu is that these causal relationships are not explicit in the rule syntax: it's not obvious that actuating `move toward` would eventually result in a `bump`. That fact only exists within the implementation of the sensors and actuators, which are black boxes with respect to the programmer's view. They transform the program constructs into a change to the underlying implementation, then produce new program constructs. In Ceptre we can use this same mechanism by passing the state update to the runtime through sensing and acting predicates.

But on the other hand, we might find it more informative to elaborate some of the movement predicates as a logic program. The following code could take the place of the builtin declarations for `bump`, `see`, and `move`:

```
sense/bump : $loc X L * $loc Y L' * !adj L L' -o bump X Y.
sense/see  : $loc X L * $loc Y L' * !visible L L' -o see X Y.
act/eat    : eat X Y * loc X L * loc Y L' -o loc X L'.
act/move_t : move toward X Y * loc X L * $loc Y L' *
              !adj L Ngh * !dist L L' D1 * !dist Ngh L' D2 *
              !lt D2 D1
              -o loc X Ngh.
act/move_a : move away X Y * loc X Y * $loc Y L' *
              !adj L Ngh * !dist L L' D1 * !dist Ngh L' D2 *
              !lt D1 D2
              -o loc X Ngh.
```

This program appeals to a number of geometric constructs such as `adj` for adjacency of locations, `visible` for visibility between locations, and `dist` for the distance between locations. Writing the rules this way forces us to expose more assumptions, such as the discretizability of game space, which are not the only sensible model for the rules given. This attempt to explicate the "black box" processes can help us understand the relationships between sets of rules in terms of the predicates that they consume and produce, and it can also reveal the underlying operational logic, exposing it to the possibility of modification.

Extensions to Kodu have investigated its potential to support richer operational logics. For example, Fristoe et al. [FDM+11] investigated the possibility of enabling social simulation mechanics by incorporating emotional state and dialogue sensors and actuators into the system. We note that these modifications are trivial to integrate in Ceptre and require only minimal extensions to the runtime processes, since the logic programming environment already supports the maintenance of state such as designations of friendship and fear between entities. In contrast, formal definitions of Kodu such as that given by Stolee [Sto10] attempt to encapsulate all of the language constructs at the same, hard-coded level, failing to distinguish the essential language structure (conditions and actions) from the highly contingent domain constructs (such as scoring and shooting).

One of Kodu's program structuring tools is the notions of *pages*, or different sets of rules that can be switched between, which closely resembles Ceptre's stages. Every rule belongs to a page, and a rule can trigger a "flip" to a different page. While Ceptre rules don't include the ability to change stages prior to program quiescence, the way we model sensing and acting means that the program will quiesce after every actor takes its turn, introducing an opportunity for a page-turning rule. That is, the Kodu rule belonging to page `p`

```
Condition Action switch page PageNumber
```

can be interpreted by generating a unique token `uniqtok` as a consequence of the corresponding rule, and changing pages at quiescence in the presence of that token:

```
stage p = {
  ...
  |Condition| -o |Action| * uniqtok
}
qui * stage p * uniqtok -o stage |PageNumber|
```

where `|-|` is the mapping from Kodu construct to Ceptre predicate outlined above.

4.6.2 PuzzleScript

PuzzleScript is a browser-based scripting language created by independent game creator Stephen Lavelle. Its target domain is 2D, tile-based puzzle games, whose rules transform sections of tiles related through the geometry of the grid, such as adjacency and row or column alignment.

The "hello world" example of PuzzleScript is *Sokoban*, in which the player controls an avatar that can push crates into empty spaces and typically advances by pushing crates onto targets. Blocks cannot pass through walls or other crates, nor can the player. It is codified in PuzzleScript through a single rule:

```
[> player | crate] -> [> player | > crate]
```

The > symbol represents a movement intent (in the present author's language, not Lavelle's) and the rule can be seen as propagating the player's intent (expressed through pressing a movement control key) on to a crate immediately adjacent in the intended direction. (The rule is generic over directions).

The avatar's free movement through empty space, and correspondingly, the disallowing of passage through blocks and walls, are codified not as rules but in the interpretation of *collision layers* over tile maps: entities on distinct layers may be superimposed and thus share a tile, but entities on the same layer cannot share a tile.

```
================
COLLISIONLAYERS
================
Background
Target
Player, Wall, Crate
```

By default, movement intents result in movement, as long as they are not blocked by the presence of some entity in the same layer.

Putting all these ideas together, the game can be codified in Ceptre as follows:

```
at layer location entity intention : pred.
adjacent location direction location : pred.

% [...]

% This stage contains the rules that PuzzleScript authors would write.
stage processIntentions = {
  % propagate player intention to crate
  push_crate :
    turn *
    $at Layer Loc player (go Dir) * adjacent Loc Dir Loc'
      * at Layer Loc' crate _
    -o at Layer Loc' crate (go Dir).
}
qui * stage processIntentions -o stage carryOutIntentions.

% resolve intentions in a nondeterministic order.
stage carryOutIntentions = {
  move :
    at Layer L Ent (go Dir) * adjacent L Dir L'
       * empty Layer L' -o at Layer L' Ent stationary * empty Layer L.
}
qui * stage carryOutIntentions -o stage cleanup.

% [...]
```

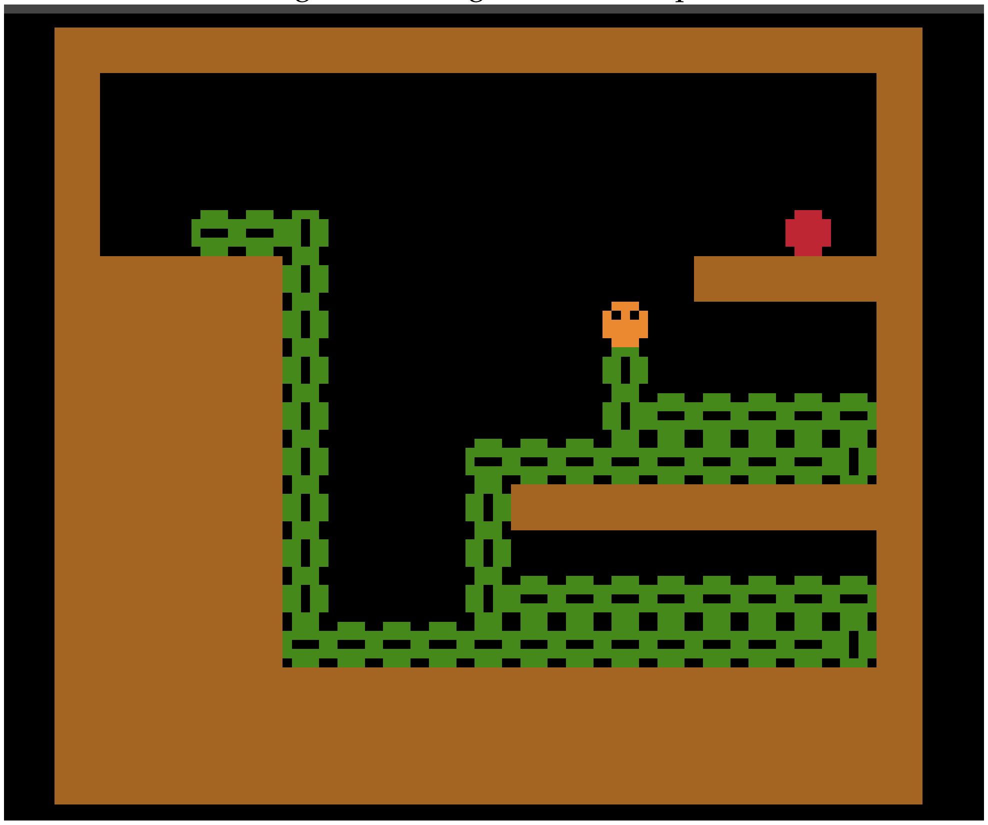
The full code listing for a runnable Sokoban implementation in Ceptre is found in Appendix C.2.

Next, we illustrate one of PuzzleScript's "intermediate" examples, a game called Lime Rick in which the player navigates a caterpillar-like avatar to targets. The avatar begins as a single-tile head, but as it moves, it leaves behind a trail of solid entities. The head is subject to gravity, and when it falls it also leaves behind such a trail. It can reach upward, but only three tiles at a time. See Figure 4.2 for a screenshot depicting a typical intermediate game state via Lime Rick's mechanics.

The game is available in PuzzleScript's built-in example collection through the web-based editor. [2]. The rules given there are:

[2]To view the code and interact with the game, go to http://www.puzzlescript.net/editor.html

Figure 4.2: A level in Lime Rick after several moves.
The avatar head is orange, indicating the current upward reach of two tiles.



```
UP [ UP PlayerHead4 ] -> [ PlayerHead4 ]
UP [ UP PlayerHead3 | No Obstacle ] -> [ PlayerBodyV | PlayerHead4 ] sfx1
UP [ UP PlayerHead2 | No Obstacle ] -> [ PlayerBodyV | PlayerHead3 ] sfx0
UP [ UP PlayerHead1 | No Obstacle ] -> [ PlayerBodyV | PlayerHead2 ] sfx0

Horizontal [ > Player | Crate | No Obstacle ]
        -> [ PlayerBodyH | PlayerHead1 | Crate ] sfx2

Horizontal [ > Player | No Obstacle ] -> [ PlayerBodyH  | PlayerHead1 ] sfx2

[ Player Apple ] [ PlayerBody ] -> [ Player Apple ] [ ]
[ Player Apple ] -> [ Player ]

[ > Player ] -> [ Player ]

DOWN [ Player | No Obstacle ] -> [ PlayerBodyV | PlayerHead1 ]
DOWN [ Crate | No Obstacle ] -> [  | Crate ]
```

We implement these rules in Ceptre using direct movement manipulation, rather

and select “Lime Rick” from the “Load Example” menu than intentions, as it is done in the PuzzleScript implementation as well. For brevity we omit the stage structure of this example.

```
move/up/1 : move up * at L (player_head red)
                -o at L (player_head red).
move/up/2 : move up * at L (player_head orange) * adj L up L'
                * empty L'
                -o at L player_body * at L' (player_head red).
move/up/3 : move up * at L (player_head yellow) * adj L up L'
                * empty L'
                -o at L player_body * at L' (player_head orange).
move/up/4 : move up * at L (player_head lime) * adj L up L'
                * empty L'
                -o at L player_body * at L' (player_head yellow).

move/r : move right * at L (player_head C) * adj L right L'
            * empty L'
            -o at L player_body * at L' (player_head lime).
% similarly move left

gravity : at L (player_head C) * adj L down L' * empty L'
             -o at L player_body * at L' (player_head C).
```

This example illustrates how Ceptre enables codifying "mechanics" in the sense of player-triggerable rules at the same level as internal game logic rules, such as gravity, with the only difference being that the rule for gravity does not consume a player-induced sensing predicate.

4.6.3 Featherweight Inform 7

Our final example is a boiled-down ("featherweight") version of the Inform 7 language for creating command-based interactive fiction (referred to as "Parser IF" by hobbyists). The language was designed to support authoring of works inspired by those released by Infocom in the 1980s, and accordingly it supports several verbs common to that genre, such as looking around, examining an object, taking an object, moving in cardinal directions, speaking to non-player characters, and inspecting one's inventory (see Montfort [Mon05] for a description of these games' conventions). Items in one's inventory often had game-specific verbs associated with them, such as "wield" or "attack [someone] with" a weapon, or "drink" a potion. In some cases, entities have their own internal state, such as a door being open or closed; entity-specific verbs like "open" and "close" affect this state. Inform 7 supports a large library of common verbs and entity types natively, and it also provides authors with mechanisms for creating new ones.

In the present author's experience, a great deal of Inform 7's power comes from the tremendous library of defaults, as well as the great care it takes to provide authors with options for seamless natural language processing and generation as well as other aspects of player experience. Although we recognize the importance of sensible defaults and polished aesthetics, we do not claim that Ceptre can capture these aspects of the tool. We instead describe a minimal target language capable of representing the range of verbs described above, purely in terms of the underlying world model.

Even ignoring the substantial natural language tooling, there is still quite a bit of complexity in faithfully describing Inform 7's rule processing pipeline, as might be inferred by a glance at its diagram in Figure 4.3. As before, the most subtle part is handling failure: how should the engine respond when the player types a command that fails to apply in the current world state? The answer will depend on what sort of failure we mean.

In parser IF, one form of failure is a parse error. That is, before we can determine whether a command is *applicable*, we need to determine whether it is *sensible*. We punt this form of failure to the implementation of the `do` sensing predicate, which is indexed by an abstract action, already in the form that our program can make sense of.

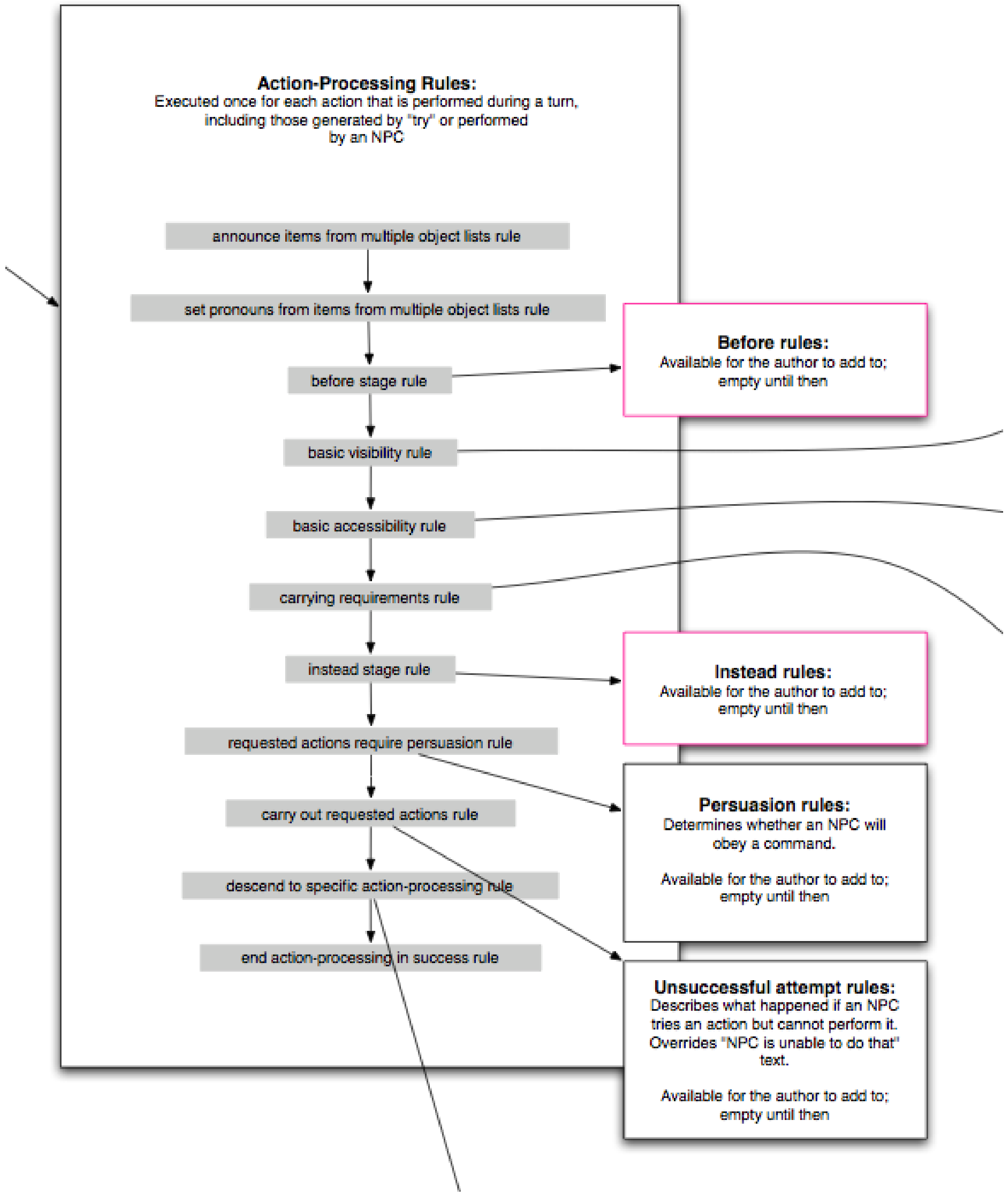
First we will give a type header for the world entities. Aside from verbs, another form of sensing predicate is *visibility*, which could be implemented in terms of the player's location relative to an object, the open or closed states of containers, darkness, and so on. We choose to leave it abstract to allow for variety in its implementation. We also include an action predicate called `success` for reporting the consequences of a player's action.

```
visible object : sense.
do verb : sense.
success verb message : action.
portable object : predicate.
at object location : predicate.
description object message : predicate.
adjacent object direction object : predicate.
  % rooms are objects
```

Now we can give some verb implementations, assuming success:

```
stage DoVerbs = {
  do/take : do (take Thing) * visible Thing
          * $portable Thing * at Thing Location
            -o at Thing player_inventory * success (take Thing) taken.
  do/look : do look * $at player Room
          * $description Room Description
            -o success look Description.
  do/examine/visible
          : do (examine Thing) * $visible Thing
          * $description Thing Description
            -o success (examine Thing) Description.
  do/examine/inventory
          : do (examine Thing) * $at Thing player_inventory
            * $description Thing Description
            -o success (examine Thing) Description.
  do/go   : do (go Direction) * at player Room
          * $adjacent Room Direction Room' * $description Room' Desc
```

Figure 4.3: Inform 7 Action Processing Diagram
From `http://inform7.com/learn/documents/Rules%20Chart.pdf`.


Action-Processing Rules:
Executed once for each action that is performed during a turn, including those generated by "try" or performed by an NPC
announce items from multiple object lists rule
set pronouns from items from multiple object lists rule
before stage rule
basic visibility rule
basic accessibility rule
carrying requirements rule
instead stage rule
requested actions require persuasion rule
carry out requested actions rule
descend to specific action-processing rule
end action-processing in success rule
Before rules:
Available for the author to add to; empty until then
Instead rules:
Available for the author to add to; empty until then
Persuasion rules:
Determines whether an NPC will obey a command.
Available for the author to add to; empty until then
Unsuccessful attempt rules:
Describes what happened if an NPC tries an action but cannot perform it. Overrides "NPC is unable to do that" text.
Available for the author to add to; empty until then

```
            -o at player Room’ * success (go Direction) Desc.
}
```

Some of the above rules should fail if certain circumstances apply. We implement those early failures as a stage that takes place before `doVerbs` (and introduce another action predicate for failure):

```
stage PreCheck = {
  check/take/already : do (take Thing) * at Thing player inventory
    -o fail (take Thing) alreadyHave.
  check/take/fixed : do (take Thing) * $visible Thing *
      $fixed Thing -o fail (take Thing) fixedInPlace.
  check/look : do look * at player Room * state Room dark
    -o fail look (darkness Room).
  check/examine : do (examine Thing) * at Player Room
    * state Room dark
    -o fail (examine Thing) (darkness Room).
  check/go : do (go Dir) * $at player R * $at R Door
    * door Door Dir closed -o fail (go Dir) (closedDoor Door).
}
qui * stage PreCheck * $do V -o stage DoVerbs.
qui * stage PreCheck * $fail V Msg -o stage Report.
```

On the other hand, some of the failure modes are only describable as the *negation* of certain things, such as visibility, which we cannot easily negate. (A sensing predicate is fundamentally *positive* information, and it would be impractical to enumerate everything one cannot see.) We address these failures by also including a *post*-processing failure stage, which we enter whenever `doVerbs` failed to consume the `do` predicate:

```
qui * stage DoVerbs * $success Verb Msg -o stage Report.
qui * stage DoVerbs * $do Verb -o stage PostCheck.

stage postCheck = {
  check/take/notHere : do (take Thing)
    -o fail (take Thing) notHere.
  check/go : do (go Dir) -o fail (go Dir) noExit.
  check/default : do V * default -o fail V defaultFail.
}

qui * stage postCheck * $do V -o default * stage postCheck.
qui * stage postCheck * $fail Verb Msg -o stage Report.
```

Although this is a drastically simplified version of the Inform 7 rules processing engine, we expose its structure in such a way that the IF author may change and potentially enrich it, in addition to simply adding new verb implementations and failure conditions. Furthermore, we have implemented standard interactive fiction verbs in the same framework in which we implemented Kodu’s robot-like sensors and action rules, and PuzzleScript’s grid-based tile manipulation rules. What we hope to have demonstrated by this point is that Ceptre offers a very flexible and composable way of writing rules for a variety of representative game types.

4.7 Implementation

Ceptre is implemented in Standard ML [Mil97] and is available as an open source project at `http://www.github.com/chrisamaphone/interactive-lp`. The implementation is approximately 2,300 lines of code, including a number of built-in sensing and acting predicates (as well as whitespace and comments). The makefile builds a binary called `ceptre` that can be run on a source file; examples may be found in the `examples` subdirectory (with `.cep` file extensions).

In addition to the core language described above, we also provide a few *directives*, or meta-linguistic operators that are placed in the source code but do not affect the underlying language meaning. One of these we have seen already is the `#trace` directive, which runs a given stage on a given initial context. The directives and their meaning are described below:

1. `#trace limit stage context.`: run the program starting in stage `stage` with initial context `context` until limit `limit` (either a number or _ for no limit) is reached.
2. `#interactive stage.`: Declare stage `stage` to block on choices from standard input when one or more of its rules applies.

4.8 Related Work

Formal *game description languages* have been a subject of investigation for some time, with unsolved problems in the area posed as recently as 2013 [ELL$^+$13]. Our work has very similar goals to those listed in the Dagshtul paper proposing VGDL (a video game description language), except that we de-emphasize the automatic generation of games as a criterion for success, emphasizing expressiveness of the language as an authoring tool instead. Additionally, our formalism is more flexible in terms of entity relationships, explicitly aiming to support abstract mechanics such as narrative and dialogue rather than focusing on object collisions in 2D space.

Smith et. al. have investigated computer-aided authoring of games through such means as constraint specification and procedural content generation [SNM09, Smi12], which shares with our approach a use of logic programming as its foundation. In this setting, game rules are treated as *term-level objects* over which program rules are written. This treatment separates the semantics of the logic program itself from the semantics of the game, meaning as a consequence that the object of formal analysis is distinct from the playable artifact. Ceptre aims to create a closer link between the logic itself and the game specification, making the language semantics itself the target of programmatic analysis as well as informal reasoning and playtesting. Our work also prioritizes the expression of exploratory mechanics rather than goal-driven ones (such as puzzle games), making exhaustive search over the rules for the sake of validation or game generation less of a concern for us. (That said, we do discuss the possibility of checking invariants over game states in Chapter 6.) A third endeavor in the sphere of executable formalisms for games is Versu [ES], which we note as a useful reference point in terms of our approach, defining Ceptre as a core formalism (c.f. Versu's Praxis) atop which a friendlier user interface (c.f. Prompter) might be built. Of course, our approach with Ceptre favors breadth (generality) rather than the depth of story- and text-specific tools of Versu.

4.9 Conclusion

We have presented a linear logic-based framework for authoring and reasoning about generative game designs, and an implementation of these ideas in the Ceptre logic programming language, which models gameplay as proof search. We have used a Shakespeare-inspired story world example to demonstrate the use of this language as a generative system to which one may selectively add interactive components. We have carried out several additional case studies with this tool, including board game, garden and dialogue simulators, illustrating Ceptre's potential for inventing new operational logics. We are actively developing the language in the open at `http://www.github.com/chrisamaphone/interactive-lp` and encourage contributions of examples from the game design community at large.

Going forward, we aim to expand the range of analytical tools for Ceptre specifications. We have in progress several candidate algorithms for checking programmer-specified game invariants, and tools for visualizing and manipulating causally-structured traces. These tools put to use in the context of the generative game examples well-described in Ceptre could lead to novel approaches to expressive range analysis [SW10], for example.

Linear logic has been used independently to study numerous phenomena, such as memory management, concurrency, game theory, and quantum physics. Its continual rediscovery in these many domains leads us to believe it is one of the more "permanent ideas" in computer science, robust to advances the technology industry that might otherwise affect the way we formulate something as culturally and technologically contingent as video game design. By providing a logical underpinning to techniques used in planning-based and ad-hoc approaches to game description, we aim to provide a basis for extension and interoperability of game specifications at the language level.

Chapter 5

Case Studies

We now give several case studies of the modeling and programming methodologies described so far. This chapter presents the results of the research; we aim to illustrate the concrete payoff that linear logic programming can have across a breadth of ludonarrative domains. We argue that the specifications presented here are much more concise and readable than they would be in a general-purpose language like JavaScript or C++, while allowing for a great deal more flexibility in expression than domain-specific languages for any of the game genres represented here. Additionally, these specifications allow for augmentation by player strategies for fuzzing, analysis in the form of visualization for causal dependencies in play traces, and the potential for deeper analysis in terms of high-level design concepts such as game balance.

In an effort to demonstrate breadth, we give the code for five complete case studies of various worlds:

- Refining an Interactive Social Story World, or `BuffyWorld`: building on the narrative modeling ideas presented in Chapters 2 and 3, we model an interactive social story world based on the television show *Buffy the Vampire Slayer*.
- Dungeon Crawler, or `DungeonWorld`: we model a "hack-and-slash" roleplaying combat mechanic with resource-management feedback loops.
- Garden Simulator, or `GardenWorld`: we model a similar mechanic to `DungeonWorld` in terms of resource management, but using a nurturing rather than violent context to inspire player affordances and autonomous world evolution.
- Settlers of Catan, or `SiedlerWorld`: we model a minimal version of the board game Settlers of Catan, in which players compete for territory and strategize about resource exchange feedback loops.
- Tamara: finally, we show an experiment that follows on from the narrative generation work in Chapter 3: we use Ceptre to model a work of participatory theater called *Tamara* [Kri89] where scenes take place in multiple rooms simultaneously. In this example, we use the result of program execution, i.e. the proof term, as the interactive artifact, rendered as a Twine text adventure that replicates the participation mechanic in the original play (that is, following characters when they exit).

5.1 Refining an Interactive Social Story World

This case study uses a similar story world to the narrative generation example from Chapter 3: characters and locations are the basic types; predicates include character locations and relationships between one another; story actions change and depend on the states described by those predicates.

The main new idea in this case study is an *action/reaction* dichotomy: in an attempt to chain together less random sequences of story, characters may perform up to one action per turn of simulation, but they also must *react* to any action performed toward them by another character.

Additionally, we let the player pick one of the characters to control for the duration of the interaction rather than letting them choose arbitrary character actions, hopefully lending a stronger sense of continuity to the experience.

The combination of these two axes, act/react and player/non-player character, make up the four stages of the program.

5.1.1 Story World Overview

Buffy the Vampire Slayer is a television franchise created by Joss Whedon, airing 1997-2003. The basic premise is that Buffy, a teenage girl who just wants an ordinary life, is the "chosen one," or Slayer, obliged to protect the world from supernatural evils lurking in the California town of Sunnydale. Throughout the show's seven seasons, the cast of characters and specific threats of evil vary substantially, but a few things remain constant rules for the Buffy universe. Here are a few that we use to shape our story world:

- Some characters are *evil*, e.g. by virtue of being vampires or demons. The slayer is obliged to destroy evil creatures.
- Evil people feel no remorse about killing, but will generally not do so at random; only if a particular goal (like destroying the slayer) is achieved. Vampires and demons are usually evil.
- Xander and Willow are Buffy's best friends; Giles is the slayer's appointed *watcher* (similar to a mentor).
- Dawn is Buffy's younger sister, who frequently drives the plot forward by getting kidnapped or accidentally casting spells that Buffy and her friends have to thwart.
- *Witches* are humans who have mastered some control over the supernatural powers that be, and can use them to various good or ill effects.

In this encoding, we include a few other members of the Buffy universe: Anya, a no-longer-evil demon (dating Xander); Tara, another witch (dating Willow); Spike, a vampire who develops a romantic obsession with Buffy; and an abstract, unspecified villain who hunts Dawn.

5.1.2 Code

Code for this section is available at `https://github.com/chrisamaphone/interactive-lp/blob/master/examples/buffy.cep`.

Below we give the characters and predicates relating characters to each other. The predicate `npc C` marks `C` as a "non-player character."

```
character : type.
buffy : character.
willow : character.
xander : character.
anya : character.
tara : character.
giles : character.
dawn : character.
spike : character.
villain : character.

npc character : pred.

evil character : pred.
witch character : pred.
slayer character : pred.
watcher character : pred.

weak character : pred.
strong character : pred.

friends character character : pred.
crush character character : pred.
dating character character : pred.
dislike character character : pred.
hunting character character : pred.
hurt_by character character : pred.
```

There are six central locations we will refer to: the crypt in the graveyard (Spike's home), the graveyard itself, Buffy's house, the Magic Box (a home for Giles' personal library and occult collection, disguised as a novelty shop), Dawn's high school, and the Bronze, a local hangout and bar. The world map is specified below.

```
place : type.
crypt : place.
graveyard : place.
buffy_house : place.
magic_box : place.
villain_lair : place.
high_school : place.

adjacent place place : bwd.
adjacent crypt graveyard.
adjacent graveyard crypt.
adjacent crypt magic_box.
adjacent magic_box crypt.
adjacent graveyard buffy_house.
adjacent buffy_house graveyard.
adjacent buffy_house magic_box.
adjacent magic_box buffy_house.
adjacent magic_box graveyard.
adjacent graveyard magic_box.
adjacent magic_box villain_lair.
adjacent villain_lair magic_box.
adjacent buffy_house villain_lair.
adjacent villain_lair buffy_house.
adjacent buffy_house high_school.
adjacent high_school magic_box.

at character place : pred.
```

The following intermediate states result from actions taken by characters; they represent an ongoing action sequence that needs to be resolved by some reaction.

```
attacking character character : pred.
hitting_on character character : pred.
spellcast character character : pred.
joking_with character character : pred.
accusing character character : pred.
pushing_away character character : pred.
acting_affectionate character character : pred.
bickering character character : pred.
```

The initial state specifies initial relationships between characters, as well as individual characteristics that may determine the range of actions available, for the particular season of the show and (perhaps) episode. (This one is loosely based off of Season 6.)

```
context init_season =
{ friends buffy willow, friends willow buffy,
  friends buffy xander, friends xander buffy,
  friends willow xander, friends xander willow,
  friends dawn buffy, friends buffy dawn,
  friends dawn xander, friends xander dawn,
  friends dawn willow, friends willow dawn,
  friends dawn tara, friends tara dawn,
  dating willow tara, dating tara willow,
  dating xander anya, dating anya xander,
  friends giles buffy, friends buffy giles,
  friends willow giles, friends giles willow,
  dislike giles spike, dislike spike giles,
  dislike xander spike, dislike spike xander,
  evil spike, crush spike buffy, dislike buffy spike,
  dislike villain buffy, evil villain,
  witch willow, slayer buffy, watcher giles, witch tara,
  strong buffy, strong willow, strong villain,
  weak xander, weak giles, weak anya, weak tara, weak dawn, weak spike}.

context init_episode =
{ hunting villain dawn, crush buffy spike,
  at dawn high_school,
  at buffy graveyard, at willow buffy_house, at tara buffy_house,
  at xander magic_box, at anya magic_box, at giles magic_box,
  at spike crypt, at villain villain_lair
}
```

The following machinery allows the player to select a character to be the "PC" (player character) from among the pool of NPCs (non-player characters).

```
no_pc : pred.
pc character : pred.
turn : pred.
end : pred. % end of story.
go : pred. % not end of story.

context init_game =
{npc buffy, npc willow, npc xander, npc anya, npc tara,
npc giles, npc spike, npc villain, npc dawn,
no_pc}.

stage select_character = {
  pick : npc C * no_pc -o pc C.
}
#interactive select_character.
select_character_to_pc_choices
: qui * stage select_character -o stage pc_choices * turn.
```

Next, we describe the choices available to the player character as actions. These include moving from place to place, and acting on other characters, specifically by attacking, making romantic advances, showing affection, joking with them, or accusing them of causing hurt. Witches can also cast spells that affect whether a character is *strong* or *weak* (which in turn affects their outcomes in combat).

The player character may also choose to end the story.

```
stage pc_choices = {
  act/move
      : turn * $pc C
      * at C L * adjacent L L'
          -o at C L' * go.

  act/kidnap/evil
```

```
    : turn * $pc C
    * $evil C
    * at C L * at C' L * weak C'
    * adjacent L L'
      -o at C L' * at C' L' * go.

act/attack/evil
    : turn * $pc C
    * $evil C
    * $at C L * $at C' L
        -o attacking C C' * go.

act/attack/slayer
    : turn * $pc C
    * $slayer C * $at C L * $at C' L * $evil C'
        -o attacking C C' * go.

act/attack/revenge/friend
    : turn * $pc C
    * $at C L * $at C' L
    * $friends C Victim * hurt_by Victim C'
    * $evil C'
    -o attacking C C' * go.

act/attack/revenge/lover
    : turn * $pc C
    * $at C L * $at C' L
    * $dating C Victim * hurt_by Victim C'
    * $evil C'
    -o attacking C C' * go.

act/hit_on
    : turn * $pc C
    * $at C L * $at C' L * $crush C C'
      -o hitting_on C C' * go.

act/affection
    : turn * $pc C
    * $at C L * $at C' L * $dating C C'
      -o acting_affectionate C C' * go.

act/bicker
    : turn * $pc C
    * $at C L * $at C' L * $dating C C'
    -o bickering C C' * go.
    act/spellcast/weak_to_strong
        : turn * $pc C * $witch C
        * $at C L * $at C’ L
        * weak C’
          -o strong C’ * spellcast C C’ * go.

    act/spellcast/strong_to_weak
        : turn * $pc C * $witch C
        * $at C L * $at C’ L
        * strong C’
          -o weak C’ * spellcast C C’ * go.

    act/watcher/push_slayer_away
        : turn * $pc C * $watcher C
        * $at C L * $at C’ L * $slayer C’
        * $strong C’
        -o pushing_away C C’ * go.

    act/accuse
        : turn * $pc C
        * $at C L * $at C’ L
        * $hurt_by C C’
        -o accusing C C’ * go.

    act/joke_with
        : turn * $pc C
        * $at C L * $at C’ L
        -o joking_with C C’ * go.

    act/none : turn -o go.

    end_story : turn -o end.
  }
  #interactive pc_choices.

  pc_to_npc :
  qui * stage pc_choices * go
    -o stage npc_reactions.

  pc_to_done :
  qui * stage pc_choices * end
    -o stage done.

  stage done = {}
```

Next, we describe how NPCs may *react* to character actions. Extending the analogy of “social physics,” this technique metaphorically reflects Newton’s Third Law—every intermediate state introduced by (appearing on the right-hand-side of) an action must be processed by (appear on the left-hand-side of) a reaction. Reactions to a given action may be nondeterministic and may depend on other parts of social state.

```
stage npc_reactions = {

  react/attacking/flee
    : $npc C
    * attacking C’ C * $weak C * at C L * adjacent L L’ -o
        at C L’ * hunting C’ C.
  react/attacking/hurt
    : $npc C
    * attacking C’ C * $weak C
        -o hurt_by C C’.
  react/attacking/fight/lose
    : $npc C
    * attacking C’ C * $strong C -o hurt_by C C’.
  react/attacking/fight/win
    : $npc C
    * attacking C’ C * $strong C -o hurt_by C’ C.

  react/hit_on/date
    : $npc C
    * hitting_on C’ C * $crush C C’
        -o dating C C’ * dating C’ C.
  react/hit_on/dislike
    : $npc C
    * hitting_on C’ C * $dislike C C’
        -o hurt_by C’ C.
  react/hit_on/reciprocate_crush
    : $npc C
    * hitting_on C’ C
        -o crush C C’.

  react/spellcast/thanks
      : $npc C * spellcast C’ C * $strong C
          -o friends C C’ * friends C’ C.
  react/spellcast/wtf
      : $npc C * spellcast C’ C
          -o accusing C C’.
  react/spellcast/bad
      : $npc C * spellcast C’ C * $weak C
          -o dislike C C’ * accusing C C’.

  react/accusing/counter
      : $npc C * accusing C’ C
          -o accusing C C’.
  react/accusing/apologize
      : $npc C * accusing C' C
          -o ().
  react/accusing/breakup
      : $npc C * accusing C' C * dating C C' * dating C' C
         -o hurt_by C C' * hurt_by C' C.

  react/joke_with/shoot_down
      : $npc C * joking_with C' C * $dislike C C'
            -o hurt_by C' C.
  react/joke_with/laugh
      : $npc C * joking_with C' C -o ().

  react/affection/good
    : $npc C * acting_affectionate C' C -o ().
  react/affection/bad
    : $npc C * acting_affectionate C' C * $hurt_by C C'
        -o hurt_by C' C.

  react/hurt_accuse
    : $npc C * hurt_by C C' * hurt_by C C'
        -o accusing C C'.
}
```

Next we give the NPC action stage, with a little bit of setup so that each character only takes one turn:

```
  qui * stage npc_reactions
    -o stage gen_npc_turns.

  stage gen_npc_turns = {
    npc C -o npc_turn C.
  }
  qui * stage gen_npc_turns -o stage npc_choices.
```

NPC actions mirror PC actions except that, not being guided by player choice, they sometimes require additional premises to drive their behavior in a coherent way.

```
  stage npc_choices = {
    act/move_toward_friend
      : npc_turn C
      * at C L * $at C' L' * $friends C C' * adjacent L L'
        -o at C L' * npc C.

    act/move_toward_enemy
      : npc_turn C
      * at C L * $at C' L' * $hunting C C' * adjacent L L'
        -o at C L' * npc C.
```

```
act/attack_hunting
  : npc_turn C
  * $at C L * $at C' L * hunting C C'
    -o attacking C C' * npc C.

act/evil_attack_dislike
  : npc_turn C
  * $evil C * $at C L * $at C' L * $dislike C C'
      -o attacking C C' * npc C.

act/evil_attack_slayer
  : npc_turn C
  * $evil C * $at C L * $at B L * $slayer B
    -o attacking C B * npc C .

act/slayer_attack_evil
  : npc_turn C
  * $slayer C * $at C L * $at V L * $evil V
    -o attacking C V * npc C.

act/attack/revenge/friend
  : npc_turn C
    * $at C L * $at C' L
    * $friends C Victim * hurt_by Victim C'
    -o attacking C C' * npc C.

act/hit_on_crush
  : npc_turn C
  * $at C L * $at C' L * $crush C C'
    -o hitting_on C C' * npc C.

act/affection_date
  : npc_turn C
  * $at C L * $at C' L * $dating C C'
    -o acting_affectionate C C' * npc C.

act/spellcast/weak_to_strong
  : npc_turn C
    * $witch C
    * $at C L * $at C' L
    * weak C'
      -o strong C' * spellcast C C' * npc C.

act/spellcast/strong_to_weak
  : npc_turn C
    * $witch C
```

```
      * $at C L * $at C' L
      * strong C'
        -o weak C' * spellcast C C' * npc C.

  act/accuse
    : npc_turn C
      * $at C L * $at C' L
      * $hurt_by C C'
      -o accusing C C' * npc C.

  act/joke_with
      : npc_turn C
      * $at C L * $at C' L
      * $friends C C'
      -o joking_with C C' * npc C.

  act/none : npc_turn C -o npc C.

}
```

Finally, we give the stage for player character *reactions*. Again, these basically mirror the NPC reactions, except that they sometimes have fewer premises (allowing the player to act out of their own volition and playful goals). Reactions must be selected until no more actions on the player are left to be reacted to. After the player has chosen all reactions, we go back to the `pc_choices` stage for the player to make new actions.

```
qui * stage npc_choices -o stage pc_reactions.

stage pc_reactions = {
  %%%% Player Reactions %%%%

  react/attacking/fight
      : $pc C * attacking C' C
          -o attacking C C'.
  react/attacking/flee
      : $pc C * attacking C' C * at C L * adjacent L L'
          -o at C L' * hunting C' C.

  react/hitting_on/reciprocate
      : $pc C * hitting_on C' C
          -o dating C C' * dating C' C.
  react/hitting_on/shoot_down
      : $pc C * hitting_on C' C
          -o hurt_by C' C.

  react/joke_with/shoot_down
      : $pc C * joking_with C' C
            -o dislike C C'.
```

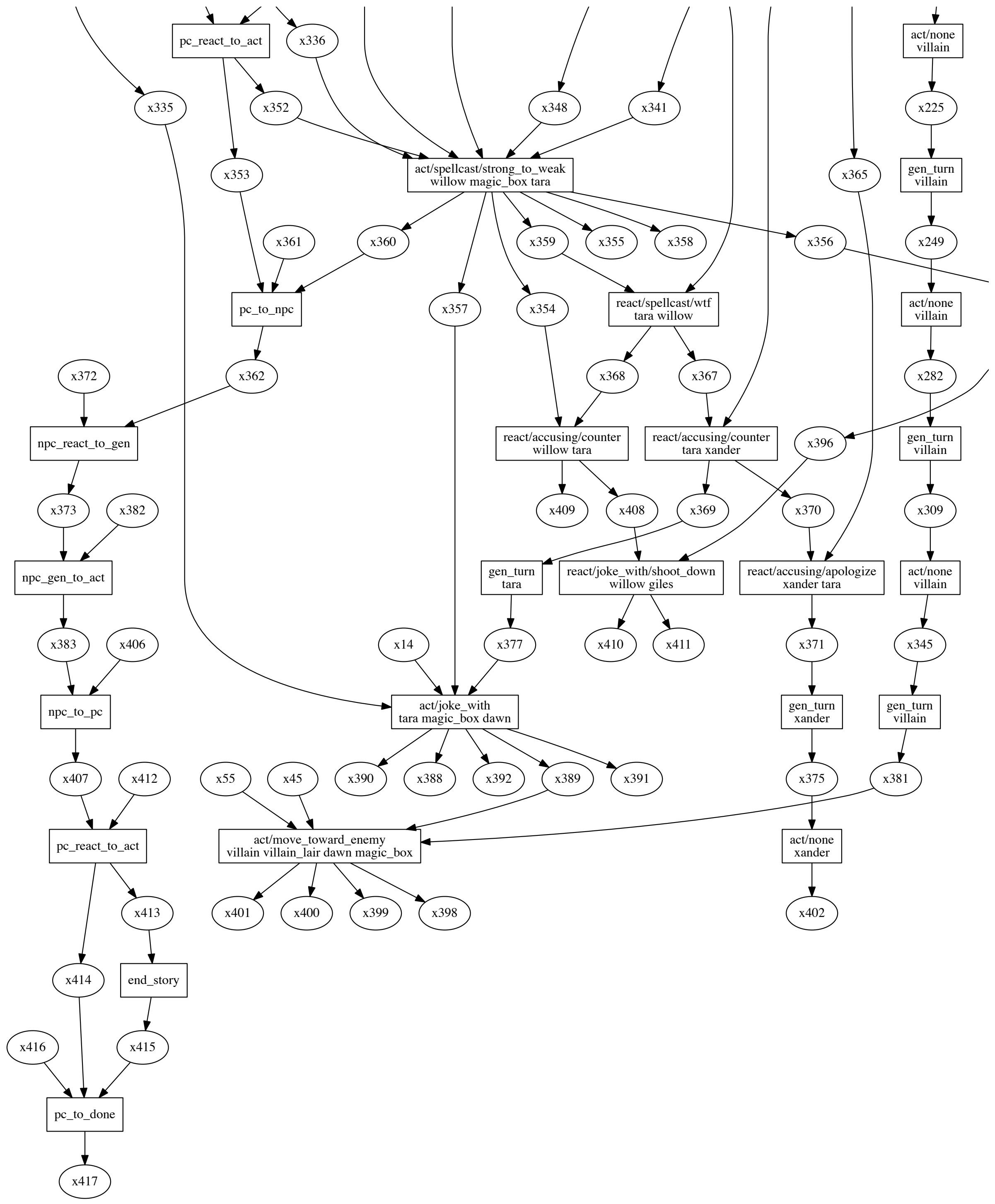
```
    react/joke_with/laugh
        : $pc C * joking_with C' C
              -o friends C' C * friends C C'.

    react/spellcast/thanks
        : $pc C * spellcast C' C * $strong C
            -o friends C C' * friends C' C.
    react/spellcast/wtf
        : $pc C * spellcast C' C
            -o accusing C C'.
    react/spellcast/bad
        : $pc C * spellcast C' C * $weak C
            -o dislike C C' * accusing C C'.

    react/accusing/counter
        : $pc C * accusing C' C
            -o accusing C C'.
    react/accusing/apologize
        : $pc C * accusing C' C
            -o ().
    react/accusing/breakup
        : $pc C * accusing C' C * dating C C' * dating C' C
            -o hurt_by C C' * hurt_by C' C.

    react/acting_affectionate/happy
        : $pc C * acting_affectionate C' C
            -o ().
    react/acting_affectionate/shoot_down
        : $pc C * acting_affectionate C' C
            -o hurt_by C' C.
}
#interactive pc_reactions.
qui * stage pc_reactions -o stage pc_choices * turn.

#trace _ select_character {init_story, init_locations, init_game}.
```

5.1.3 Sample Interaction

To illustrate interaction with the program, rather than giving a transcript, we show graphical output of a few interesting play traces (automatically generated with GraphViz [1]).

In the first interaction, we select the character Willow. In the interactive stages for determining her actions, we choose transitions that correspond to her casting spells, arguing with Tara, and joking with her friends. A portion of the resulting trace follows:

[1]http://www.graphviz.org/



This graph depicts the action of Willow casting a spell on Tara, Tara reacting negatively, and other actions by the non-player characters. One of the final scenes (transitions whose output resources have no out-edges) depicts the villain pursuing Dawn. A side argument between Xander arises, due to Tara having hurt Willow (Xander's friend) in a prior argument.

Re-enacting an Episode

We created an augmented version of the story world where every character may be controlled by interaction for the sake of re-enacting a specific *Buffy* episode: the critically acclaimed "musical" episode, featuring a demon who causes the town to express their inner thoughts through song and dance.

In the episode, Dawn is kidnapped by the dance demon, Anya and Xander's relationship falters, Tara discovers Willow used a spell to erase her memory of them arguing, and Giles pushes Buffy away from his mentorship. The episode culminates on Buffy's charge to rescue Dawn, where she is eventually followed by all of her friends for a climactic final scene.

The character interactions involved in this climactic build-up can be seen in Figure 5.1.

5.1.4 Discussion

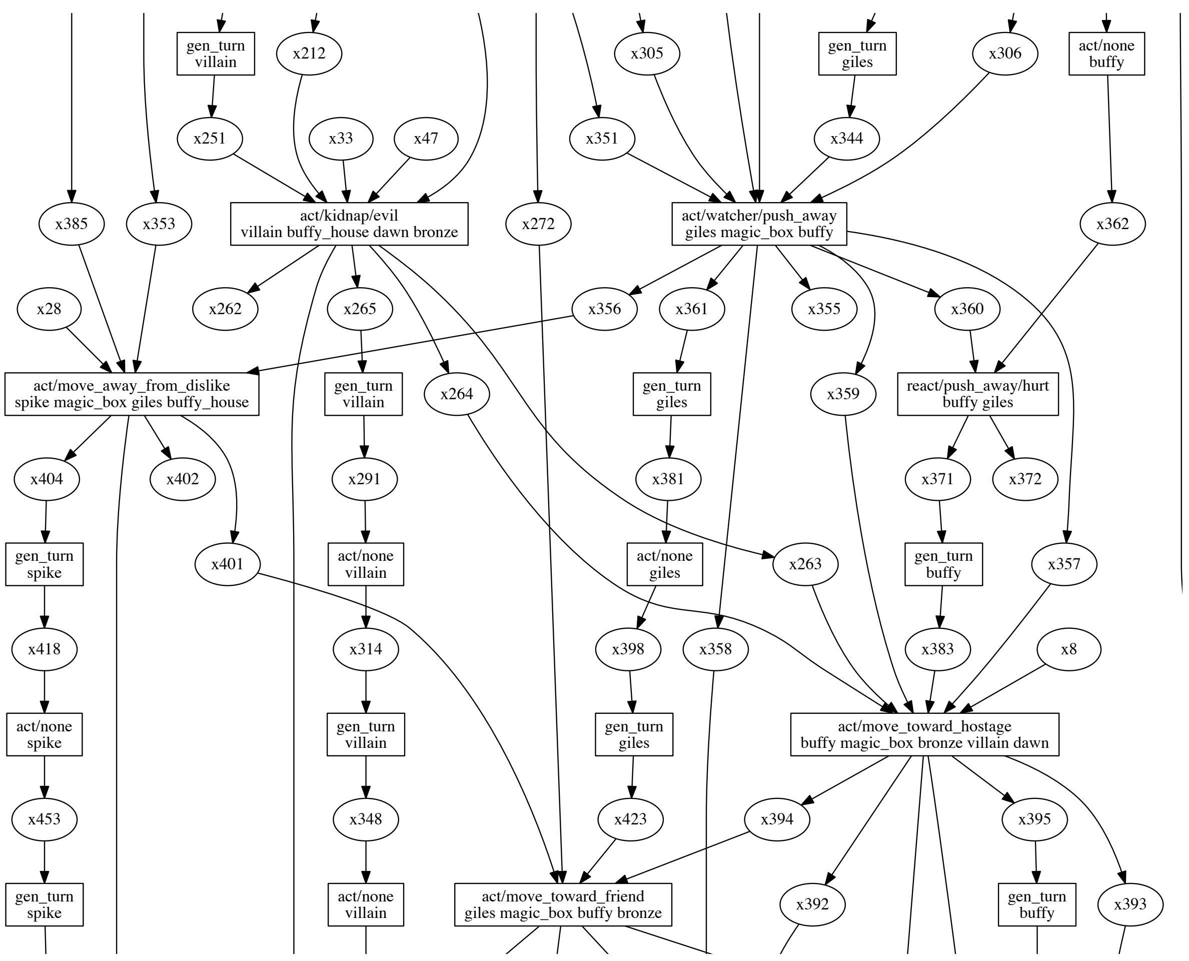
This case study has presented Ceptre code for a collaborative storytelling system between a human player and several NPCs, who all may act and react according to similar logic. It does not contain any scripted scenes or endings and is in that sense a "sandbox" or "open world" narrative system, but it is also somewhat asymmetric in that characters are assigned different traits and abilities. We could imagine easily extending this example to model supernatural powers such as vampiric siring, summoning demons, and casting more intricate spells.

The development of this example also serves to suggest the limitations of linear logic programming for narrative expression. This action/reaction model serves as substitute for characters having their own goals pertaining to the narrative and solve for those goals to create plans using the same inference mechanisms available to the logic itself. The use of planning for interactive storytelling includes that kind of character-specific inference [CCM02, RY10], though a logical formalism for both projective inference of this kind *and* state change would need at least the fully-reflexive (dependently-typed) capabilities of CLF in which predicates may refer to proofs as terms. We leave further investigations of this potential logical correspondence to future work.

Another limitation of the exemplified technique, though not necessarily of the programming language, is that each action or reaction must specify the complete set of characters it involves; any social interactions involving three or more characters would require additional rules. We believe that the "broadcast" pattern shown in Section 4.3.3 could be a basis for an interaction models wherein all characters are given the opportunity to react to any action taking place, say, in the same room as them, but it could substantially complicate the program.

As future work, we would also like to investigate the potential to use "social practices" a la Versu [ES] to constrain social actions. The basic idea is that at any given time, a finite number of social practices, or sets of roles and rules for those roles, apply to the world state, and characters select their actions on the basis of their roles within those practices. As an example, Buffy might only attack someone during a *combat* social

Figure 5.1: A trace representing part of the “Once More, With Feeling” episode of *Buffy*.



practice, in which she inhabits her role as the slayer, but she might navigate that role separately from (or simultaneously with) navigating her role as a supportive friend to Willow in the social practice of friends confiding in one another.

In summary, we have presented an investigation into the use of Ceptre to specify an interactive social story world based on *Buffy the Vampire Slayer*, using an action/reaction model and allowing the interactor to select a character to play throughout the interaction. This concludes the case study.

5.2 Dungeon Crawler

Next, we examine the task of specifying a "dungeon crawler"-inspired game design. This game might have state such as player character skill level, equipment such as weapons and armor, treasure that can be spent, player character health, and NPC health. It might have actions like adventuring or combat (in which randomness as well as player-determined properties can influence the outcome), resting to recover from damage, shopping to spend money on weapons or armor, and training to increase skill level.

We choose this domain as an exemplar of a common and popular operational logic: for example, the basic idea can be found in analog games like *Dungeons and Dragons* [GAH74] and digital games ranging in time from *Rogue* and *Hack* to the *Legend of Zelda* series to contemporary games made by large studios, like Blizzard's *Hearthstone*. Representing this operational logic, at the core of so many existing games, should be a good benchmark for the expressiveness of Ceptre.

5.2.1 Game Design Goals

We speculate that the reason this operational logic (and common mechanics created with it) has enjoyed so much success has to do with the *dynamics* afforded by feedback loops in the mechanics: easy, small tasks (training) lead to medium-sized successes (defeating an enemy), which has payoff (treasure) that accumulates until larger successes (purchasing a shiny new weapon) may be enjoyed—and those successes feed back into the ease of the smaller and medium-sized tasks until a satisfying feeling of *momentum* is achieved. Typically, as rewards scale, they also enable more *strategic agency*: they present choices (e.g. between a large number of upgrade items for purchase) that give the player not just a sense of difficulty in selecting the "right" one, but also a sense of personalization in choosing between several equally-good alternatives (particularly when those choices are reflected in the form of an avatar).

In our experiment, we do not aspire to creating the same kind of aesthetic enjoyment made possible by these mechanics, but we do hope to create at least a similar-feeling *dynamic* of play: the sense that *increase in momentum* occurs when the player finds a working strategy. That goal will inform the choices we make.

5.2.2 Iteration 1

Code for this section is available at `https://github.com/chrisamaphone/interactive-lp/blob/master/examples/rpg-tiny.cep`.

We start with a minimal iteration of the game wherein resources—treasure, stamina, damage, and skill—are represented as atomic predicates.

```
treasure : pred.
stamina : pred.
damage : pred.
skill : pred.
```

A few intermediate predicates are used as well. (We use these by invariant as Boolean values—at every program step, each of these is either in the context once or zero times.)

```
alive : pred.
fighting : pred.
turn : pred.
```

The `do` stage is controlled by the player, and contains actions for training, fighting, healing, resting, and upgrading a weapon (which is in effect simply training without consuming stamina).

```
stage do = {
  train : turn * stamina * $alive -o skill.
  fight : turn * stamina * $alive -o fighting.
  heal : turn * $alive * damage -o stamina.
  rest : turn * $alive -o stamina.
  buy_weapon : turn * treasure * treasure * treasure * $alive -o skill.
}
#interactive do.
qui * stage do -o stage happen.
```

The `happen` stage processes the player's selected action, nondeterministically resolving fights (though they are more likely to result in a hit the more skill a player has) and determining when the player gets tired or dies.

```
stage happen = {
  turn -o ().
  fight/hit : fighting * $skill -o treasure.
  fight/miss : fighting -o damage.
  get_tired : $damage * stamina * $alive -o ().
  die : damage * damage * damage * alive -o ().
}
qui * stage happen -o stage do * turn.
```

We can run the simulation with a starting configuration like the one below, and fiddle with the amount of starting stamina if needed:

```
context init = { stamina, stamina, stamina, alive, turn}.
#trace _ do init.
```

5.2.3 Iteration 2

Code for this section is available at `https://github.com/chrisamaphone/interactive-lp/blob/master/examples/rpg.cep`.

In our second iteration, we will use numerically-indexed predicates rather than multiple copies of an atomic predicate, so that we may do simple arithmetic on their values.

Ceptre allows the definition of permanent facts via backward-chaining predicates, so for instance, we can define the arithmetic operation of addition capped at a certain maximum value as a predicate `cplus A B Cap C` which can be read as `A` plus `B` capped at `Cap` is `C`. We omit the definition of the predicate for brevity, but make use of it in later rules. We also use backward-chaining predicates to define a few constants, such as the player's maximum health and the damage and cost of various weapons:

```
max_hp 10.  damage sword 4.  cost sword 10.
```

We then define a initial context and an initial stage that sets up the game's starting state:

```
context init_ctx = {init_tok}.
stage init = {
  i : init_tok * max_hp N
      -o health N * treasure 0 * ndays 0
          * weapon_damage 4.
}
```

We define the rest of the game using a "screen" idiom, with predicates representing the main, rest, adventure, and shop screens. (Some type header information is omitted.)

```
qui * stage init -o stage main * main_screen.

stage main = {
  do/rest : main_screen -o rest_screen.
  do/adventure : main_screen -o adventure_screen.
  do/shop : main_screen -o shop_screen.

  do/quit : main_screen -o quit.
}
#interactive main.

qui * stage main * $rest_screen -o stage rest.
qui * stage main * $shop_screen -o stage shop.
qui * stage main * $adventure_screen -o stage adventure.
qui * stage main * quit -o ().
```

The `rest` and `shop` stages allow recharging health (at the cost of an increment to the number of days) and upgrading one's weapon damage in exchange for treasure, respectively:

```
stage rest = {
  recharge : rest_screen
              * health HP * max_hp Max * recharge_hp Recharge
```

```
                * cplus HP Recharge Max N
                * ndays NDAYS
               -o health N * ndays (NDAYS + 1).
}
qui * stage rest -o stage main * main_screen.

stage shop = {
  leave : shop_screen -o main_screen.
  buy : treasure T * cost W C * damage_of W D * weapon_damage _
          * subtract T C (some T')
          -o treasure T' * weapon_damage D.
}
#interactive shop.
qui * stage shop * $main_screen -o stage main.
```

The adventure stages are the most complex part of the code, involving the random generation of a monster and random spoils are collected upon player victory. Spoils are only added to the treasure bank if the player does not flee from a fight in progress. To do all of this, we need an adventure initialization stage (`init`), a monster generating stage (`fight_init`), a way of responding to player actions in context (`fight_auto`), and a choice for the player between fighting and fleeing (`fight`):

```
stage adventure = {
  init : adventure_screen -o spoils 0.
}
qui * stage adventure -o stage fight_init * fight_screen.

% drop_amount M N means a monster of size M can drop N coins
drop_amount nat nat : bwd.
drop_amount X X. % for now

stage fight_init = {
  init : fight_screen -o gen_monster * fight_in_progress.
  gen_a_monster : gen_monster * monster_size Size
                    -o monster Size * monster_hp Size.
}
qui * stage fight_init -o stage fight * choice.

try_fight : pred.
fight_in_progress : pred.
stage fight_auto = {
  fight/hit
    : try_fight * $fight_in_progress * monster_hp MHP * $weapon_damage D
        * subtract MHP D (some MHP')
        -o monster_hp MHP'.
  win
    : fight_in_progress * monster_hp MHP * $weapon_damage D
        * subtract MHP D none
        -o win_screen.
  fight/miss
    : try_fight * $fight_in_progress * $monster Size * health HP
           * subtract HP Size (some HP')
        -o health HP'.
  die_from_damages
    : health 0 * fight_in_progress -o die_screen.
  fight/die
    : try_fight * fight_in_progress * monster Size * health HP
           * subtract HP Size none
           -o die_screen.
}
choice : pred.
qui * stage fight_auto * $fight_in_progress -o stage fight * choice.
qui * stage fight_auto * $win_screen -o stage win.
qui * stage fight_auto * $die_screen -o stage die.

stage fight = {
  do_fight : choice * $fight_in_progress -o try_fight.
  do_flee  : choice * fight_in_progress -o flee_screen.
}
#interactive fight.
qui * stage fight * $fight_in_progress -o stage fight_auto.
qui * stage fight * $flee_screen -o stage flee.
```

Choosing `flee` takes you back to the main screen without any spoils:

```
stage flee = {
  % lose spoils
  do/flee : flee_screen * spoils X * monster _ * monster_hp _
             -o ().
}
qui * stage flee -o stage main * main_screen.
```

Finally, we need stages for winning and dying in combat:

```
go_home_or_continue : pred.
stage win = {
  win
    : win_screen * monster Size * drop_amount Size Drop
         -o drop Drop.
  collect_spoils
    : drop X * spoils Y * plus X Y Z
      -o spoils Z * go_home_or_continue.
  go_home
    : go_home_or_continue
      * spoils X * treasure Y * plus X Y Z
      -o treasure Z * main_screen.
  continue
    : go_home_or_continue -o fight_screen.
}
#interactive win.
qui * stage win * $main_screen -o stage main.
qui * stage win * $fight_screen -o stage fight_init.

end : pred.
stage die = {
  quit : die_screen -o end.
  restart : die_screen * monster_hp _
              * spoils _ * ndays _ * treasure _
              * weapon_damage _ -o init_tok.
}
#interactive die.

qui * stage die * end -o ().
qui * stage die * $init_tok -o stage init.
```

The program can then be run with the directive `#trace _ init init_ctx`.

The development of this example benefited extensively from the ability to specify interactivity modularly, occasionally making non-player-directed stages (like `fight_auto`) interactive to debug them. Additionally, we can test the design by "scripting" certain player strategies. For instance, we could augment the two rules in the `fight` stage to be deterministic, fighting when the monster can't kill us in one turn and fleeing otherwise:

```
stage fight = {
  do_fight :
    choice * $fight_in_progress
    * $monster Size * $health HP * Size < HP
            -o try_fight.
  do_flee  :
    choice * fight_in_progress
    * $monster Size * $health HP * Size >= HP
      -o flee_screen.
}
```

If we remove interactivity from this stage, then we get automated combat sequences that should never result in the player's death.

5.2.4 Discussion

In summary, this case study illustrates the use of Ceptre to program simple combat strategy mechanics found in several dungeon-crawler, RPG, and roguelike game genres. We use stages extensively to coordinate not just interaction between the game and player, but also different autonomous parts of the game.

This case study suggests potential analyses that could be carried out on gameplay traces. The ability to script player strategies means that we could draft a few and compare them, perhaps by charting the various numeric quantities over time (program steps). Such analyses are suggested by Dormans [Dor11] in the development of strategy game mechanics to study things like feedback loops.

5.3 Settlers of Catan

Our next case study is an encoding of a *non-digital* game, *The Settlers of Catan*. Settlers of Catan is a strategic board game designed by Klaus Teuber and published in 1995 [SCS10], a notable, award-winning exemplar of a trend of similarly-styled board games achieving popularity in the 1990s.

At a high level, Settlers is a game in which players take the role of colonizers of an island, Catan, and position their settlements in such a way as to maximize their benefit from the (somewhat randomized) production of resources. Resources can then be used to build more settlements, and ultimately to achieve victory (through a points system).

We choose Settlers as an object of study due to its popular success as well as its amenability to formalization in Ceptre due to its use of resource exchange as a central mechanic. It gives us an opportunity to illustrate how we can codify non-digital processes in a computational way such that one could imagine prototyping such rules in Ceptre before physical implementation. It allows us to illustrate operational logics like rolling dice, placing pieces, and drawing cards from a shuffled, but pre-determined deck. This will also be our first formalization of a multiplayer game, drawing attention to turn-taking in a new light and raising some interesting research questions about peer-to-peer protocols (as in trading) and information hiding (as in secret hands of cards).

Below we summarize the game rules described by Teuber [Teu07].

5.3.1 Game Rules

The Settlers of Catan is a 2-4-player board game played on a grid of hexagonal tiles (hexes), where each hex represents some terrain which may produce *resources*—wool, lumber, grain, brick, or ore—which players need to collect in order to build roads, settlements, and cities on the grid. Roads are placed along edges of the hexes, settlements are placed at intersections of hexes, and cities are *upgrades* of settlements.

Every turn, the player whose turn it is rolls a pair of six-sided dice to determine resource production. Every hex is marked with a number corresponding to a possible roll of the dice, and when that number comes up, those hexes produce whatever resources correspond to their terrain. Every player who has a settlement next to one of those hexes receives a copy of the resource, and every player who has a city next to one of them receives *two* copies.

After resource production, the player whose turn it is may *trade* resources with other players. They do so by making an offer, such as "I will trade one *wool* for either *lumber* or *grain*," which may or may not be accepted by another player. Other players can make their own counter-offers as well. Trade takes place when (and only when) another player accepts the offer.

Finally, after trading, the player whose turn it is may *build* within the limits of their resources. They may build a road in any empty edge that connects to one of their pre-existing roads, settlements, or cities; they may build a settlement in any empty intersection that connects to one of their roads; and they may build a city as an upgrade to a settlement, replacing it on the game board. Building costs resources, summarized in the table below:

Building	Ore	Grain	Lumber	Wool	Brick
Road	-	-	1	-	1
Settlement	-	1	1	1	1
City	3	2	-	-	-
Development card	1	1	-	1	-

A *development card* is drawn from a shuffled stack of *knight*, *progress*, and *victory point* cards. Victory points are how the game ends: a player wins once they accrue 10 or more victory points (see the Victory Points section below). We consider the game's core mechanics to be characterized by the above rules where the *only* development cards are victory points.

The other development cards, and more complex, special-case mechanics are described next. We list these mainly to supply the palette of game design ideas from which we will draw to give interesting use cases for formalization.

Game Setup

The official rules of the game supply two possibilities for setting up the game: one in which a provided board determines the locations of the hexes and their dice roll values (see Figure 5.2), the other of which is for a *variable* board, recommended for advanced players, wherein hexes and their corresponding dice roll values are randomly determined.

In either case, the players also do two rounds of initial settlement placement: they get to place one settlement and one road adjacent to that settlement in an open intersection and edge during each round. For more details on initial placement, see the rulebook [Teu07].

The Robber

A single "desert" hex tile always appears on the board. This tile produces no resources. A piece called the *robber* is placed initially in the desert. It moves when the dice roll totals to 7 (which does not otherwise trigger any resource production). Whoever makes that dice roll decides which adjacent hex the robber should move to. Whenever the robber is on a hex, that hex does not produce any resources.

Knights

A *knight* is a development card that a player may keep in their hand and play on their turn. It allows them to move the robber to an adjacent hex.

Figure 5.2: Settlers of Catan beginner's starting map.

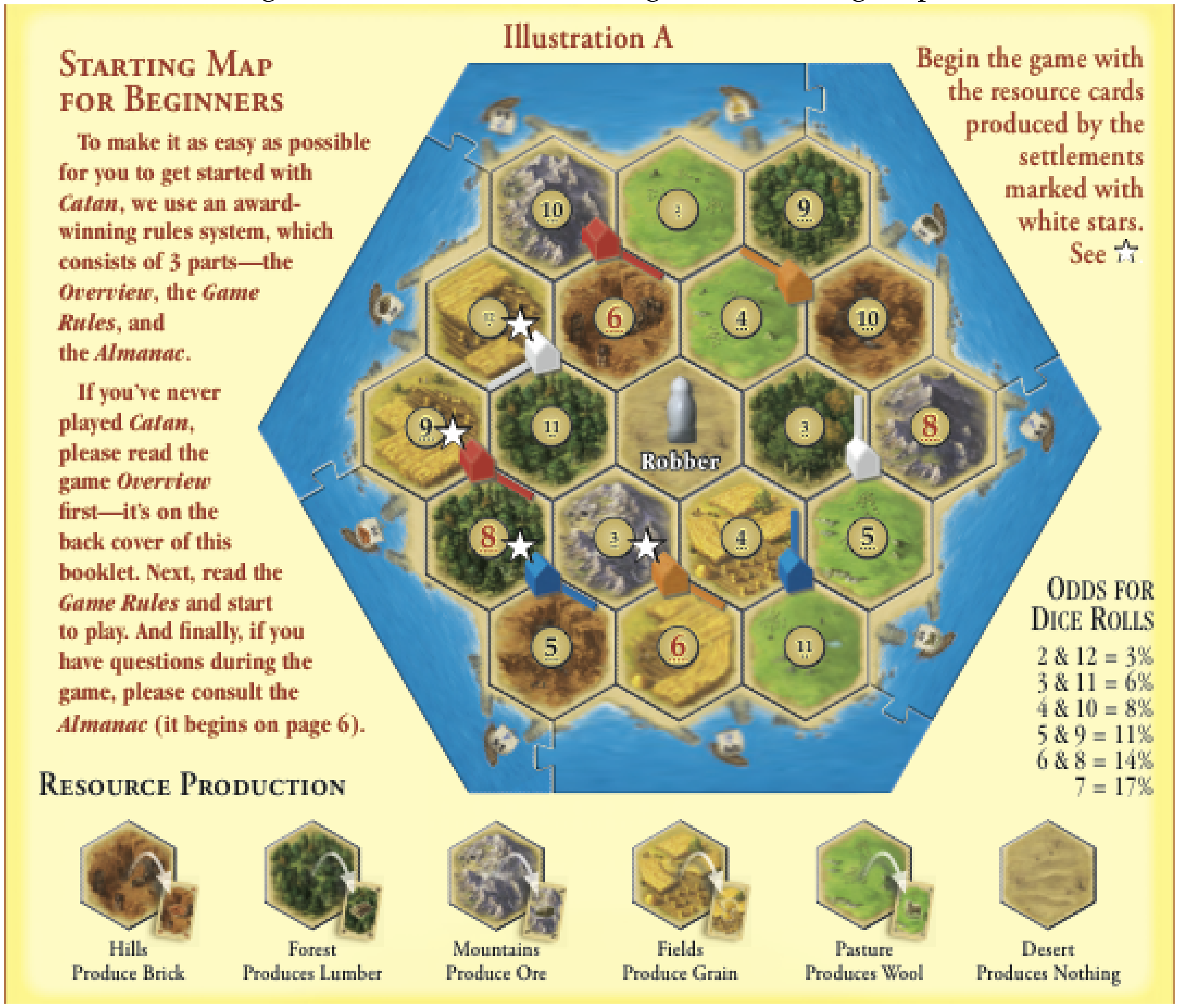


Illustration A

STARTING MAP FOR BEGINNERS

To make it as easy as possible for you to get started with *Catan*, we use an award-winning rules system, which consists of 3 parts—the *Overview*, the *Game Rules*, and the *Almanac*.

If you've never played *Catan*, please read the game *Overview* first—it's on the back cover of this booklet. Next, read the *Game Rules* and start to play. And finally, if you have questions during the game, please consult the *Almanac* (it begins on page 6).

Begin the game with the resource cards produced by the settlements marked with white stars. See ☆

ODDS FOR DICE ROLLS

2 & 12 = 3%
3 & 11 = 6%
4 & 10 = 8%
5 & 9 = 11%
6 & 8 = 14%
7 = 17%

RESOURCE PRODUCTION

Hills	Forest	Mountains	Fields	Pasture	Desert
Produce Brick	Produces Lumber	Produce Ore	Produce Grain	Produces Wool	Produces Nothing

Maritime Trade

On a player's turn, they can trade resources without involving another player at an exchange rate of $4 : 1$—that is, they may trade four identical resource cards (to the ambient supply) for one resource card of their choice. *Harbors*, or settlements built at special points on the edge of the game board, can enable more effective maritime trade rates. (See the rulebook for details [Teu07].)

Progress Cards

Another kind of card that can be drawn from the pool of development cards is a *progress* card, each of which grant the player some one-time special ability when they play it (which they are able to do on their turn). Progress cards include:

- Road Building: When the player plays this card, they may immediately place two free roads on the board (according to normal building rules).
- Year of Plenty: When the player plays this card, they may immediately take any two resource cards from the supply.
- Monopoly: When the player plays this card, they must name one type of resource, and then the other players must give them all of the resource cards of that type from their hands.

Victory Points

Several facets of the game count toward victory points. Players win when it is both their turn *and* they have 10 or more victory points.

- A settlement is worth one victory point.
- A city is worth two victory points.
- Whoever has the longest road (of at least five individual road pieces) has two additional victory points (this designation can change throughout the game).
- Whoever has the "largest army" (number of Knight cards played) of at least three plays has two additional victory points (similar to the longest road designation).
- A victory point card is worth one victory point.

5.3.2 Discussion of the Game's Design

We have described the game's rules in more detail than necessary for our case study, but we want to give the reader a sense of appreciation for the numerous subtleties and edge cases. At its heart, Settlers is a simple game of territory acquisition and resource management; undoubtedly, its designer started with those ideas and recognized the need for more intricate rules through iteration and playtesting.

Therefore, to study its design pragmatically, we start with a codification of a single central mechanic. We identify the exchange of resources for building settlements, cities, and roads as such, but there are other candidates: the randomness of resource production is another (amplified by other randomness-introducing elements, like the Robber); the socioeconomics of the *trading* mechanic is yet another; and finally, the intricate dynamics created by the geometry of the board is another—all of the faces, edges, and intersections formed by the hexagons are interdependent and relevant to play.

5.3.3 Iteration 1

Code for this section is available at `https://github.com/chrisamaphone/interactive-lp/blob/master/examples/siedler_core.cep`.

In the first iteration, we model the central mechanic of *exchanging resources for development* from the point of view of a single player. We will illustrate the dichotomy between player choices (in the building phase) and the "external" choice of the random deck draw in the selection of development cards. However, we will not model the meaning of development cards in this iteration.

Resources are represented as predicates (i.e. resources in the meta-logic, conveniently enough).

```
brick : pred.
lumber : pred.
road : pred.
wool : pred.
grain : pred.
settlement : pred.
ore : pred.
city : pred.
development_card : pred.
knight : pred.
progress : pred.
victory_point : pred.
```

We have two predicates to sequentialize turn taking between the player and the random draw from the deck.

```
turn : pred.
success : pred.
```

The first stage is a collection of rules for transforming basic resources (brick, lumber, wool, grain, and ore) into settlements, cities, roads, and development cards.

```
stage building = {

build_road : turn * brick * lumber -o road * success.
build_settlement : turn * brick * lumber * wool * grain
  -o settlement * success.
build_city : turn * ore * ore * ore * grain * grain
  -o city * success.
build_development : turn * ore * wool * grain
  -o development_card * success.
}
#interactive building.
```

When the building stage quiesces, we transition to the development card drawing stage to process any possible development card draws. There is one rule in this stage per possible development card.

```
qui * stage building * success -o stage draw_dev_card.

stage draw_dev_card = {

dev_knight : development_card -o knight.
dev_prog : development_card -o progress.
dev_victory_point : development_card -o victory_point.

}
qui * stage draw_dev_card -o stage building * turn.
```

Finally, we specify an initial state with a somewhat arbitrary collection of initial resources for testing.

```
context init =
{turn,
  brick, brick, brick, lumber, lumber, lumber, wool, wool, wool,
  grain, grain, grain, ore, ore, ore}.

#trace _ building init.
```

Sample Interaction

In Figure 5.3, we show two distinct ways of partitioning the available resources according to Settlers' building mechanics.

5.3.4 Iteration 2

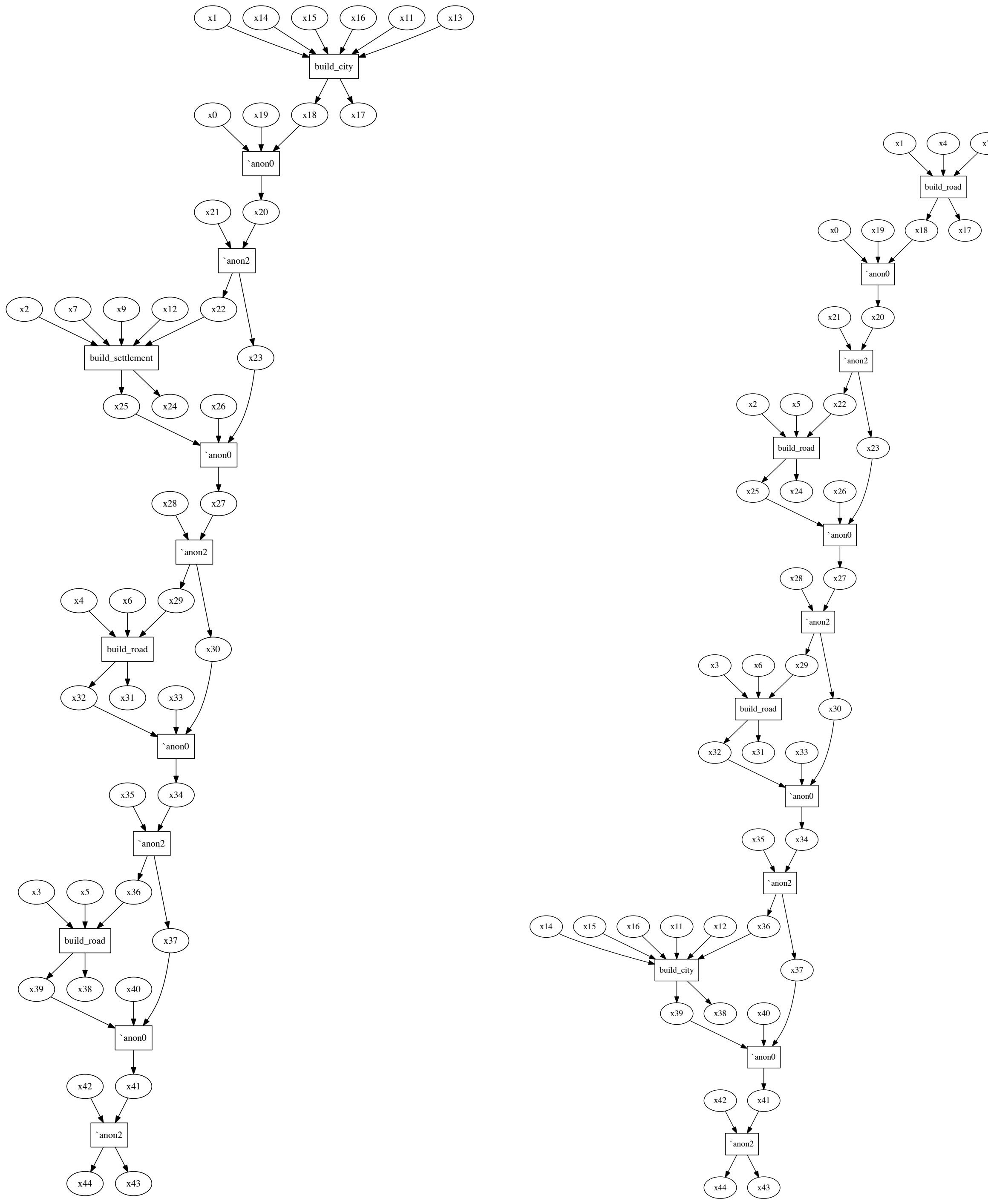
Code for this section is available at https://github.com/chrisamaphone/interactive-lp/blob/master/examples/siedler1.cep.

In the second iteration of our formalization of Settlers, we introduce multiple players with hands of resources, the production of resources by settlements and cities, and trading. The building of settlements is still not constrained by the geometry of the board, but the production of resources is. We will use a miniature version of the game board, too (but large enough to provide variability in play).

We model players as inhabitants of a player type related by a turn ordering. The turn and done predicates are used to manage turn taking.

```
player : type.
red : player. blue : player. yellow : player.
```

Figure 5.3: Two possible play traces under the Settlers building mechanics.



```
% turn order
next player player : bwd.
next red blue.
next blue yellow.
next yellow red.

turn player : pred.
done player : pred.
```

Basic resources are now terms rather than predicates; their predicate form is additionally indexed by a player.

```
resource : type.
brick : resource.
lumber : resource.
wool : resource.
grain : resource.
ore : resource.

holds player resource : pred.
```

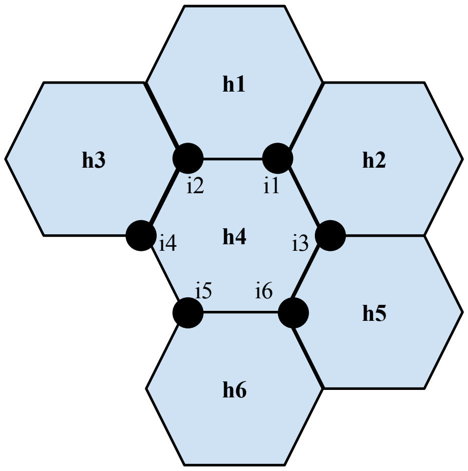
We establish a small board made of six hexes, each individual inhabitants of the `hex` type. We associate with each hex the resource that it produces.

```
hex : type.
h1 : hex. h2 : hex. h3 : hex. h4 : hex. h5 : hex. h6 : hex.

produces hex resource : bwd.
produces h1 grain.
produces h2 lumber.
produces h3 ore.
produces h4 wool.
produces h5 grain.
produces h6 brick.
```

We introduce six intersections, each at one point of the central hex. The locations of these intersections relative to the hexes are specified by associating each hex with a *list* of intersections it touches. The map we represent corresponds to the one shown in Figure 5.4.

```
intersection : type.
i1 : intersection.
i2 : intersection.
i3 : intersection.
i4 : intersection.
i5 : intersection.
i6 : intersection.

ilist : type.
nil : ilist.
```

Figure 5.4: Sample map for second iteration of Settlers of Catan encoding.



```
cons intersection ilist : ilist.

adjacent hex ilist : bwd.
adjacent h1 (cons i1 (cons i2 nil)).
adjacent h2 (cons i1 (cons i3 nil)).
adjacent h3 (cons i2 (cons i4 nil)).
adjacent h4
  (cons i1 (cons i2 (cons i3 (cons i4 (cons i5 (cons i6 nil)))))).
adjacent h5 (cons i3 (cons i6 nil)).
adjacent h6 (cons i5 (cons i6 nil)).
```

For each intersection, we track whether it is available (empty) or has a settlement or city at it.

```
available_land intersection : pred.
has_settlement player intersection : pred.
has_city player intersection : pred.
```

We specify the starting board with all intersections available. We specify (without loss of generality) the red player starting first, and give each player a token for initialization. In this minimal version of the game, each player gets to place only *one* settlement, as codified in the `setup` stage:

```
context board =
{ available_land i1,
  available_land i2,
  available_land i3,
  available_land i4,
  available_land i5,
  available_land i6 }.
```

```
init_settlement player : pred.
context players =
{turn red,
  init_settlement red,
  init_settlement yellow,
  init_settlement blue }.

context init = {board, players}.

stage setup = {
  % everyone gets one settlement to start,
  % going in random order.
  settle
  : init_settlement P * available_land X
    -o has_settlement P X.
}
#interactive setup.
```

For dice rolling, we introduce a `die` predicate (an unrolled die) and a stage with as many rules as possible rolls, each causing the die to transform into an indication of a certain hex.

```
die : pred.
setup_to_rolling : qui * stage setup -o stage roll * die.

roll hex : pred.
stage roll = {
  roll1 : die -o roll h1.
  roll2 : die -o roll h2.
  roll3 : die -o roll h3.
  roll4 : die -o roll h4.
  roll5 : die -o roll h5.
  roll6 : die -o roll h6.
}
roll_to_prod : qui * stage roll -o stage produce.
```

Next, we do rule production by propagating the die roll to every intersection adjacent to the corresponding hex. The `mete` rules process the list of adjacent intersections, stopping when there are none left, skipping empty ones, and introducing new resources into players' hands when there are cities or settlements there.

```
prod intersection : pred.
mete resource ilist : pred.
stage produce = {
  process_roll :
    roll H * produces H R * adjacent H Is
    -o mete R Is.
  mete/nil : mete R nil -o ().
  mete/skip : mete R (cons I Is) * $available_land I
              -o mete R Is.
  mete/settlement
    : mete R (cons I Is) * $has_settlement Player I
              -o holds Player R * mete R Is.
  mete/city
    :  mete R (cons I Is) * $has_city Player I
              -o holds Player R * holds Player R
                  * mete R Is.
}
```

Next is the trading stage. We model trading with a single rule indicating that any two cards in someone's hands can swap places.

```
prod_to_trade : qui * stage produce -o stage trade.

stage trade = {
  do/trade : $turn P * holds P R * holds P' R'
           -o holds P R' * holds P' R.
  donetrading : turn P -o done P.
}
#interactive trade.
```

This simplistic model of trading fails to capture a few important details, such as the ability to trade many-for-one or one-for-many cards. Also, the default presentation of all available rules with the `#interactive` mechanism reveals all of the resource cards in a player's hand (information that is not actually secret except by virtue of the limitations of human memory, but still intended to be kept hidden). However, for our second iteration, still meaning only to capture the essence of the game, this model suffices.

The next stage after trading is building. The building rules look similar to those in Iteration 1, except that they refer to the availability of an intersection (for settlements) and to the prior establishment of a settlement (for cities). The player may build arbitrarily many times, so the turn token is preserved in each rule application, until they explicitly choose to be done with the `finished` rule.

```
trade_to_build : qui * stage trade * done P -o stage build * turn P.

stage build = {
  build_settlement :
    $turn P
      * holds P brick * holds P lumber * holds P wool * holds P grain
      * available_land L
    -o has_settlement P L.

  build_city :
    $turn P
      * has_settlement P I * holds P ore * holds P ore * holds P ore
        * holds P grain * holds P grain
    -o has_city P I.

  finished : turn P -o done P.
}
#interactive build.
```

Finally, the building stage changes the turn to the next player upon quiescence and transfers control back to the `roll` stage.

The top-level program begins in the `setup` stage and with the initial context provided at the top of the program.

```
build_to_roll :
  qui * stage build * done P * next P P'
   -o stage roll * turn P' * die.

#trace _ setup init.
```

Sample Interaction

We can play through this version of the game with the following transition sequence, starting with setup and concluding with the red player building a second settlement:

```
(settle red i1)
(settle blue i6)
(settle yellow i4)
roll6
(process_roll h6 brick (cons i5 (cons i6 nil)))
(mete/skip brick i5 (cons i6 nil))
(mete/settlement brick i6 nil blue)
(mete/nil brick)
(donetrading red)
(finished red)
roll2
(process_roll h2 lumber (cons i1 (cons i3 nil)))
(mete/settlement lumber i1 (cons i3 nil) red)
(mete/skip lumber i3 nil)
(mete/nil lumber)
(donetrading blue)
(finished blue)
roll2
(process_roll h2 lumber (cons i1 (cons i3 nil)))
(mete/settlement lumber i1 (cons i3 nil) red)
(mete/skip lumber i3 nil)
(mete/nil lumber)
(donetrading yellow)
(finished yellow)
roll4
(process_roll h4 wool (cons i1 (cons i2 (cons i3 (cons i4 (cons i5 (cons i6 nil)))))))
```

```
(mete/settlement wool i1 (cons i2 (cons i3 (cons i4 (cons i5 (cons i6 nil))))) red)
(mete/skip wool i2 (cons i3 (cons i4 (cons i5 (cons i6 nil)))))
(mete/skip wool i3 (cons i4 (cons i5 (cons i6 nil))))
(mete/settlement wool i4 (cons i5 (cons i6 nil)) yellow)
(mete/skip wool i5 (cons i6 nil))
(mete/settlement wool i6 nil blue)
(mete/nil wool)
(do/trade red lumber blue brick)
(donetrading red)
(finished red)
roll3
(process_roll h3 ore (cons i2 (cons i4 nil)))
(mete/skip ore i2 (cons i4 nil))
(mete/settlement ore i4 nil yellow)
(mete/nil ore)
(donetrading blue)
(finished blue)
roll1
(process_roll h1 grain (cons i1 (cons i2 nil)))
(mete/settlement grain i1 (cons i2 nil) red)
(mete/skip grain i2 nil)
(mete/nil grain)
(donetrading yellow)
(finished yellow)
roll5
(process_roll h5 grain (cons i3 (cons i6 nil)))
(mete/skip grain i3 (cons i6 nil))
(mete/settlement grain i6 nil blue)
(mete/nil grain)
(donetrading red)
(build_settlement red i5)
```

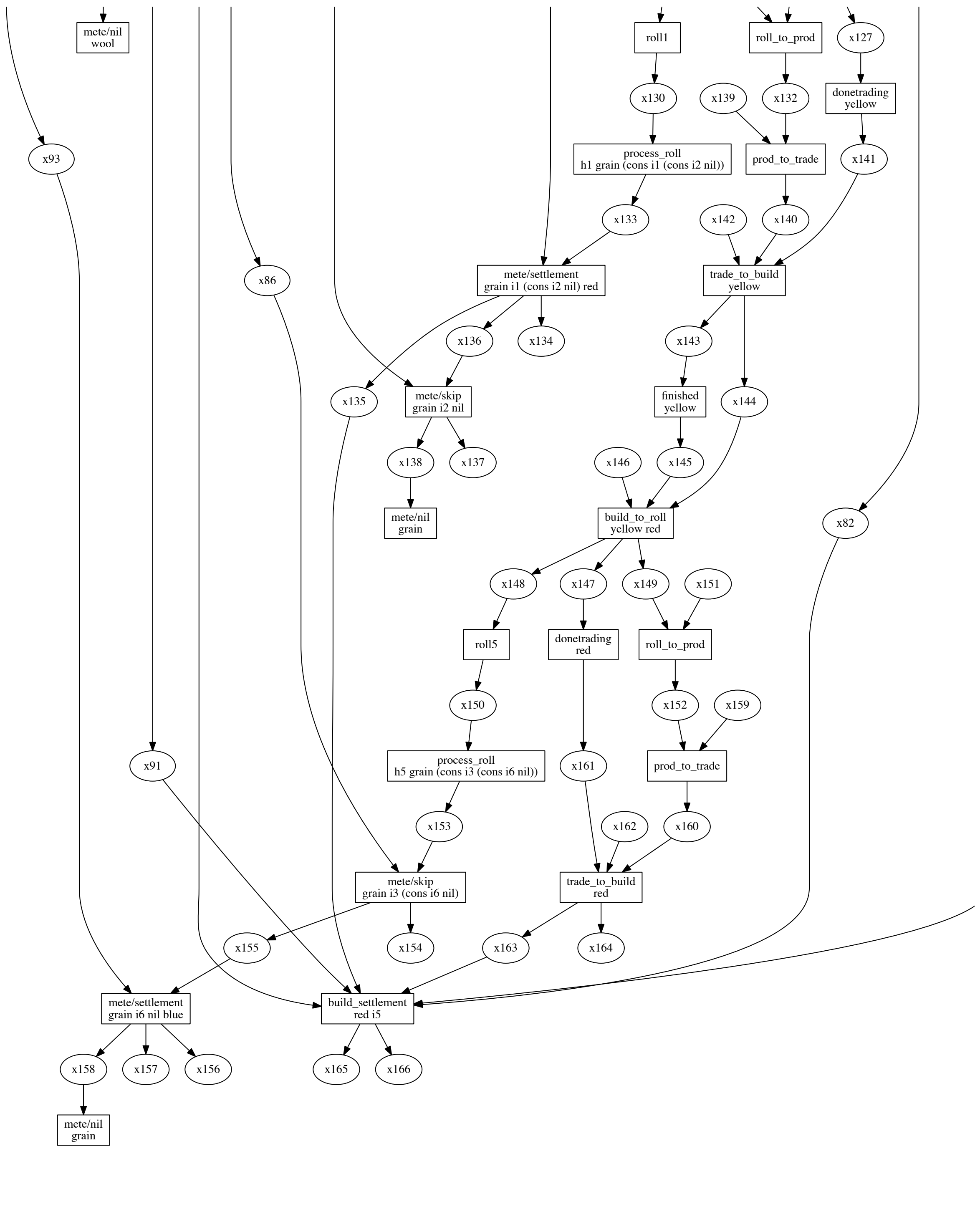
The causal structure of the final sequence can be found in Figure 5.5.

5.3.5 Discussion

The second iteration we have presented is a playable prototype of all of the game's core rules, amenable to further iteration. There are a few ways in which it fails to capture some of the subtleties of Settlers' mechanics:

- *Trade*: as mentioned briefly, the formalization of the trading mechanic does not quite model what is interesting about trading in Settlers. If we think of hands of cards as secret information, then trading really emphasizes a model of distributed information, which would map onto distributed proof search in the formalism. An interesting future research direction might be to investigate epistemic modal logic as a basis for modeling this knowledge distribution.

Figure 5.5: Trace graph for part of a Settlers of Catan playthrough.



Additionally, the trading stage of Settlers follows a *protocol* of offering and accepting trades that is not modeled in this formalism. An offer might be rejected or replaced with a counter-offer, and only once both parties agree will it be accepted. We could potentially extend this formalization to account for the peer-to-peer player interaction in more detail.

- *Building*: we have not expressed the constraints on building described by the Settlers rule book, i.e. the requirement that new settlements be built next to roads owned by the building player and the requirement that new roads be built next to anything owned by the building player. It should be straightfoward to introduce those constraints in a future iteration, although tracking *edges* between hexes in addition to intersections requires careful modeling.

Additionally, there are several aspects of Settlers' play that could be built into further iterations on this encoding, each of which might only be introduced to balance some other shortcoming of the simpler set of mechanics, determined through playtesting:

- *Board map*: being able to express arbitrarily large and complex board layouts, at least up to the size and expressiveness of the physical board game, should be possible. We would need to introduce another stage and several more predicates to create the appropriate geometric constraints.
- *Win conditions*: we have not modeled victory points nor ending conditions based on them, but this should be straightforward, especially since they can only take effect for a player during their turn.
- *Random factors*: things like the robber, development cards, and alternative routes to victory points (longest road and largest army) add additional unpredictability into the game that can lead to subtle strategies.
- *Maritime trade*: we have not modeled maritime trade in this iteration of the game, though it would be straightforward to add it.
- *Finite resource pools*: in Settlers, each player can only build up to the allowance of their initially-alotted supply of cities, settlements, and roads. Our encoding currently allows for building as many of these as players' hands of cards allow.

5.3.6 Potential Analyses

Even with the simplicity of our second iteration compared to the full game, we can still learn from it. By removing the `#interactive` directives from the stages and generating enough random play traces, we could perform analysis to answer the following questions:

- Is there a set of initial conditions, e.g. board size, number of tiles of each resource, et cetera, that leads to more balanced games? One way to answer this would be to randomly generate those conditions in the setup stage, then measure "balance" by the proportion of wins by each player on a sufficiently large playthrough sample. This technique resembles that used by the Ludi system [Bro08] for the procedural generation of game rules via evolutionary algorithms.

- If we were to build a physical version of the game, what is the right number of cities, settlements, and resource cards to include? We can answer this question by doing a resource usage analysis of the game playthroughs, noting the maximum number of each predicate in the context corresponding to the physical resource we are interested in (as well as averages of those numbers).

This concludes our Settlers of Catan case study.

5.4 Garden Simulator

Our next case study is an original simulation design built using an iterative process like that described in Section 5.3. This study is an open-ended simulation that puts the player in the role of a gardener.

5.4.1 Game Idea

At a high level, we want to use autonomous proof search to model the growth of plants in a garden, and interactive proof search to tend the garden. We would like to create interesting tension between expanding the variety and size of the garden and keeping all of the plants healthy. There should be several types of plant, each of which takes up some soil space, grows if given the proper amount of water, and dies otherwise. Once plants have reached maturity, they may be harvested for new seeds.

To introduce tension between expansion and plant health, plants need to require just enough attention from the gardener (player) that on any given turn, the decision between planting something new and tending to the needs of existing plants is difficult. Real-world phenomena that we might want to model include the presence of weeds, overgrowth of some plants we are trying to grow (that might impede the growth of other plants in the garden), sensitivity to overwatering, and plant pests or diseases.

5.4.2 Iteration 1

Code for this section is available at `https://github.com/chrisamaphone/interactive-lp/blob/master/examples/garden-small.cep`.

For the first iteration of the game, we will just model the basic mechanic: planting, watering, and harvesting plants as player actions; and growth and death of plants as autonomous actions.

We start with three types of plants that will not be distinguished by the mechanics.

```
plant_type : type.
flower : plant_type.
herb : plant_type.
vine : plant_type.
```

Plants have three growth phases (seed, seedling, and mature) and three health states (healthy, wilting, and dead).

```
plant_phase : type.
seed : plant_phase.
seedling : plant_phase.
mature : plant_phase.

next plant_phase plant_phase : bwd.
next seed seedling.
next seedling mature.

health : type.
healthy : health.
wilting : health.
dead : health.
```

A plant can be harvested, in seed form (not planted), or planted at a particular place in the garden with a particular growth phase and health, as modeled by the following predicates.

```
harvested plant_type : pred.
have_seed plant_type : pred.
plant plant_type plant_phase health : pred.
```

The `play` stage models gardener actions: planting, watering, harvesting, clearing away dead plants, and extracting seeds from harvested plants. Planting requires soil space. The stage is provided with one `step` token, which is consumed and re-produced by every action except for `done`, meaning that we can perform as many actions as we want before ending our turn and passing control back to the autonomy of the garden.

```
step : pred.
soil_space : pred.

stage play {
  get_seeds : $step * harvested Type -o have_seed Type * have_seed Type.
  plant : $step * have_seed Type * soil_space -o plant Type seed healthy.
  harvest : $step * plant Type mature healthy -o harvested Type * soil_space.
  clear : $step * plant Type _ dead -o soil_space.
  water : $step * plant Type Phase wilting -o plant Type Phase healthy.
  done : step -o ().
}
#interactive play.

qui * stage play -o stage grow * step.
```

The `grow` stage may also take an arbitrary number of steps before passing control back to the gardener. Its possible actions are growing, wilting, and dying.

```
stage grow = {
  grow : $step * plant Type Phase healthy * next Phase Phase'
           -o plant Type Phase' healthy.
  wilt : $step * plant Type Phase healthy
```

```
            -o plant Type Phase wilting.
  die : $step * plant Type Phase wilting
            -o plant Type Phase dead.
  done : step -o ().
}
qui * stage grow -o stage play * step.
```

We test the game with three starting seeds and four soil spaces.

```
context init = {have_seed flower, have_seed herb, have_seed vine, step,
                  soil_space, soil_space, soil_space, soil_space}.
#trace _ play init.
```

Sample Interaction

In Figure 5.6, we give a partial trace graph for a sample interaction with the system.

5.4.3 Iteration 2

Code for this section is available at https://github.com/chrisamaphone/interactive-lp/blob/master/examples/garden-med.cep.

In our next iteration, we introduce some additional synchronization in the form of a day counter.[2]

```
nat : type.
z : nat.
s nat : nat.

day nat : pred.
```

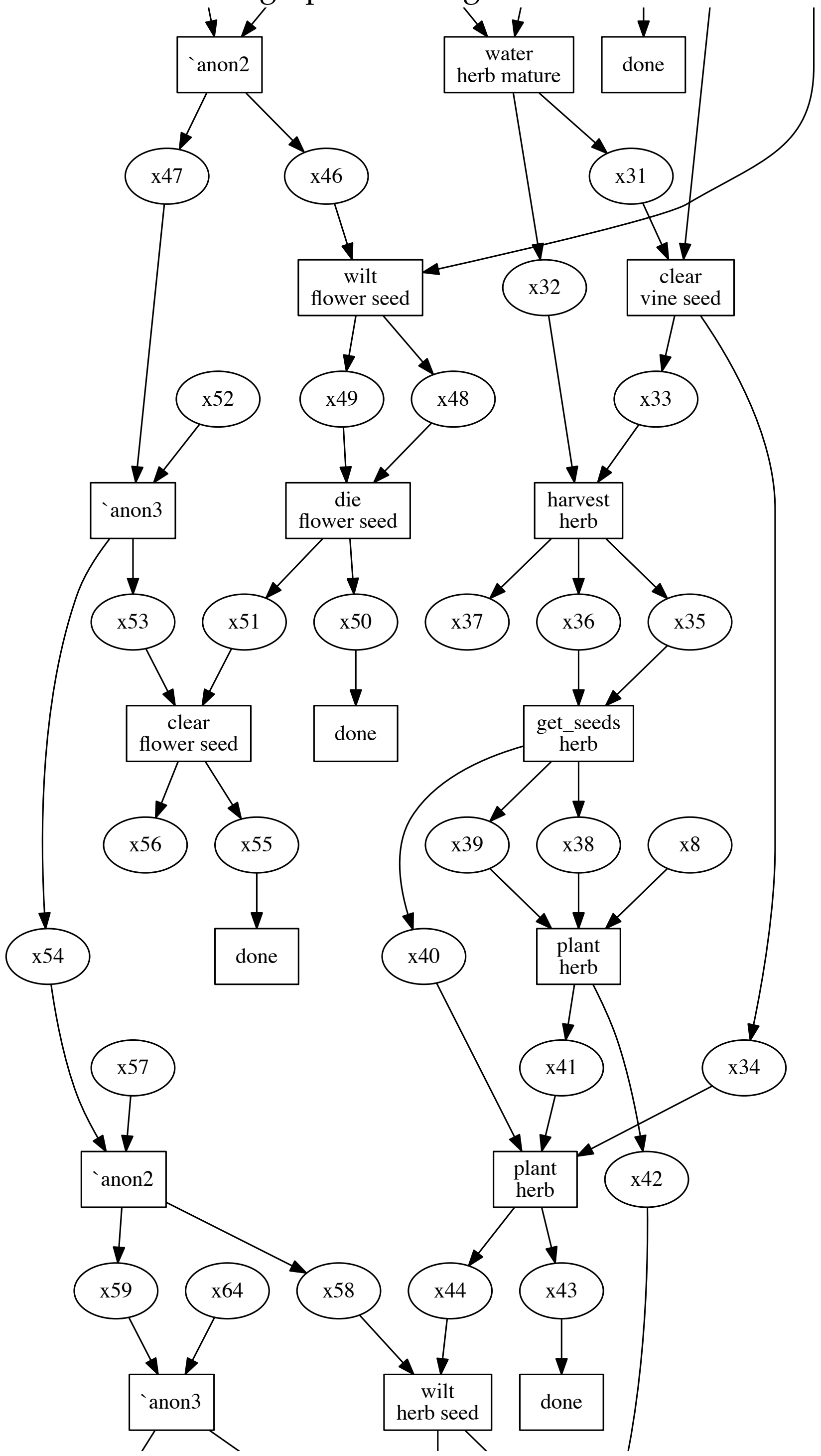
The plant types, health and maturity levels are the same as before. And, as before, a plant can either be harvested, in seed form, or planted, but the plant predicate now tracks some additional information for synchronization: the last day it has taken a step in the grow stage.

```
harvested plant_type : pred.
have_seed plant_type : pred.
% last var is for synchronization
plant plant_type plant_phase health nat : pred.
```

We give the player three turns in this iteration. Planting a new seed marks its last argument with the current day D.

[2]In this example, we will represent numbers in unary notation: z for 0 and s N for N+1. Ceptre's implementation allows for the designation of user-defined natural numbers in this fashion to be mapped to built-in integers, thus giving the programmer some flexibility in which notation to use—note that the unary notation is convenient for the sake of pattern-matching in the premises to rules, but treating numbers as built-in integers has the advantage that operators like multiplication and addition can be written in-line without defining them as logic programs.

Figure 5.6: Partial trace graph for first garden simulator iteration.



```
step : pred.
turns nat : pred.
soil_space : pred.

stage play {
  get_seeds :
    turns (s N) * harvested Type
      -o have_seed Type * have_seed Type * turns N.

  plant :
    turns (s N) * have_seed Type * soil_space * $day D
      -o plant Type seed healthy D * turns N.

  harvest : turns (s N) * plant Type mature healthy _
      -o harvested Type * soil_space * turns N.

  clear : turns (s N) * plant Type _ dead _
      -o soil_space * turns N.

  water : turns (s N) * plant Type Phase wilting D
      -o plant Type Phase healthy D * turns N.

  done : turns _ -o ().
}
#interactive play.
qui * stage play -o stage advance_day * step.
```

Before the growth stage, we advance the day.

```
stage advance_day = {
  advance : day N * step -o day (s N).
}
qui * stage advance_day -o stage grow.
```

In the growth stage, each plant only takes a step if its day index lags behind the current day, and the step updates it to be current. This technique enforces that each plant takes exactly one turn of growth or ailment in this stage. (This technique for enforcing that behavior is an alternative to the `npc_turn` trick used in Section 5.1).

```
stage grow = {
  grow : plant Type Phase healthy D * next Phase Phase' * $day (s D)
            -o plant Type Phase' healthy (s D).
  wilt : plant Type Phase healthy D * $day (s D)
            -o plant Type Phase wilting (s D).
  die : plant Type Phase wilting D * $day (s D)
            -o plant Type Phase dead (s D).
}
qui * stage grow -o stage play * turns (s (s (s z))).
```

```
context init = {day z,
  have_seed flower, have_seed herb, have_seed vine, have_seed succulent,
  soil_space, soil_space, soil_space, soil_space, turns (s (s (s z)))}.
#trace _ play init.
```

Sample Interaction

In Figure 5.7, we give a partial trace graph for a sample interaction with the system.

5.4.4 Iteration 3

Code for this section is available at `https://github.com/chrisamaphone/interactive-lp/blob/master/examples/garden-iter3.cep`.

In our third iteration, we introduce concrete terms for the spaces that plants can inhabit and allow the player to trade harvested plants for space upgrades. We also fix the bug from the second iteration by adding a bit of extra state (watered/needing water) and adding a drinking action to the autonomous growth stage.

The structure of this iteration now resembles the action/reaction model from the social story world described in Section 5.1.

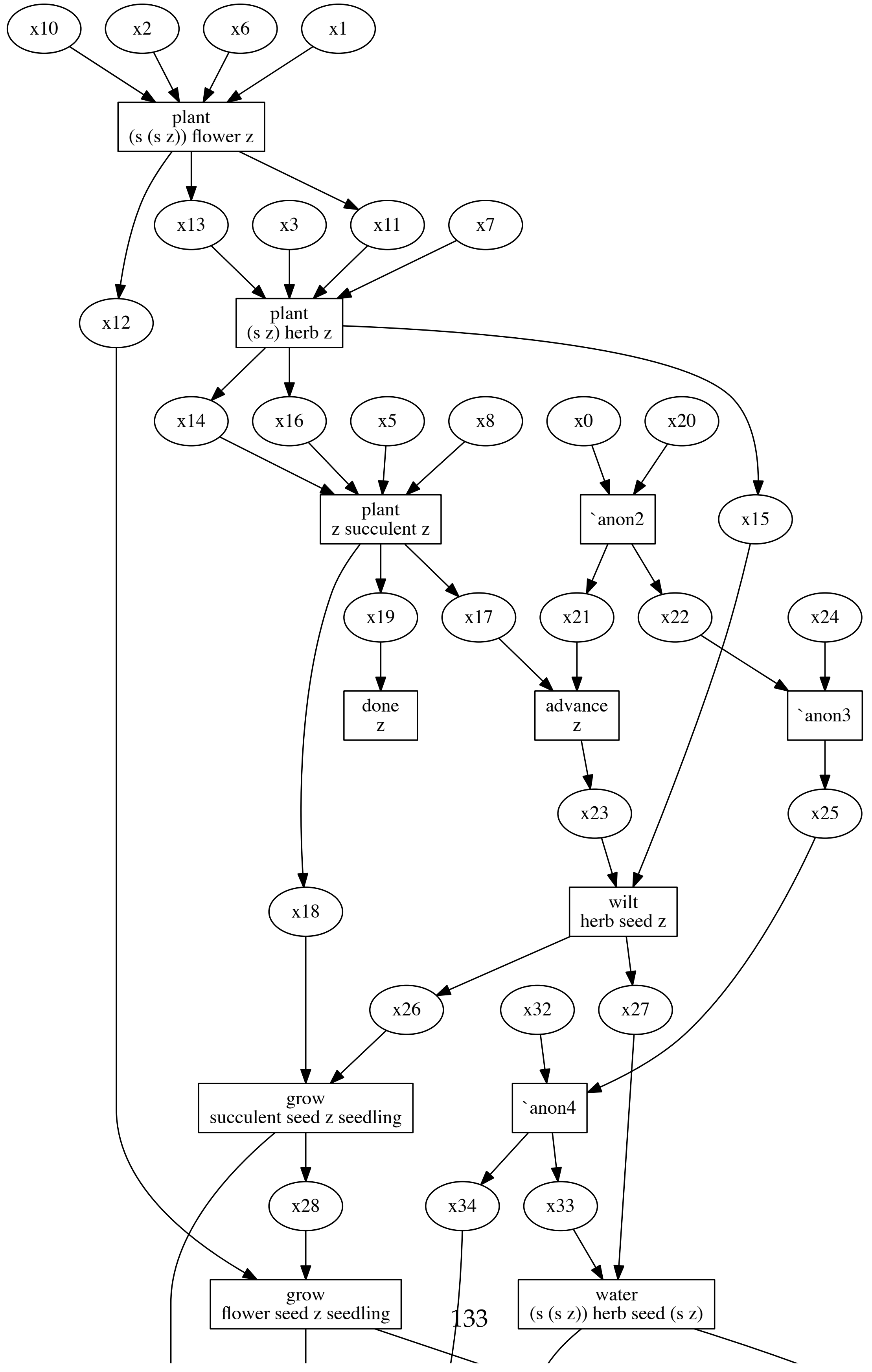
The code below starts by introducing a type for garden space and giving the predicates over spaces (omitting the type header common to prior iterations):

```
space : type.
s1 : space. s2 : space. s3 : space. s4 : space.
s5 : space. s6 : space. s7 : space. s8 : space.

max_space : pred.

next_space space space : bwd.
next_space s4 s5.
next_space s5 s6.
next_space s6 s7.
next_space s7 s8.

empty space : pred.
plant_at space : pred.
plant_type_at space plant_type : pred.
plant_phase_at space plant_phase : pred.
health_at space : pred.
needs_water space : pred.
watered space : pred.
```

Note that individual properties of the plant space may now be specified independently, using the identifier of the soil space as a tag to link all of those properties.

The play stage now includes an option to buy more soil space, and watering introduces a `watered` fact distinct from the plant's health:

Figure 5.7: Partial trace graph for second garden simulator iteration.



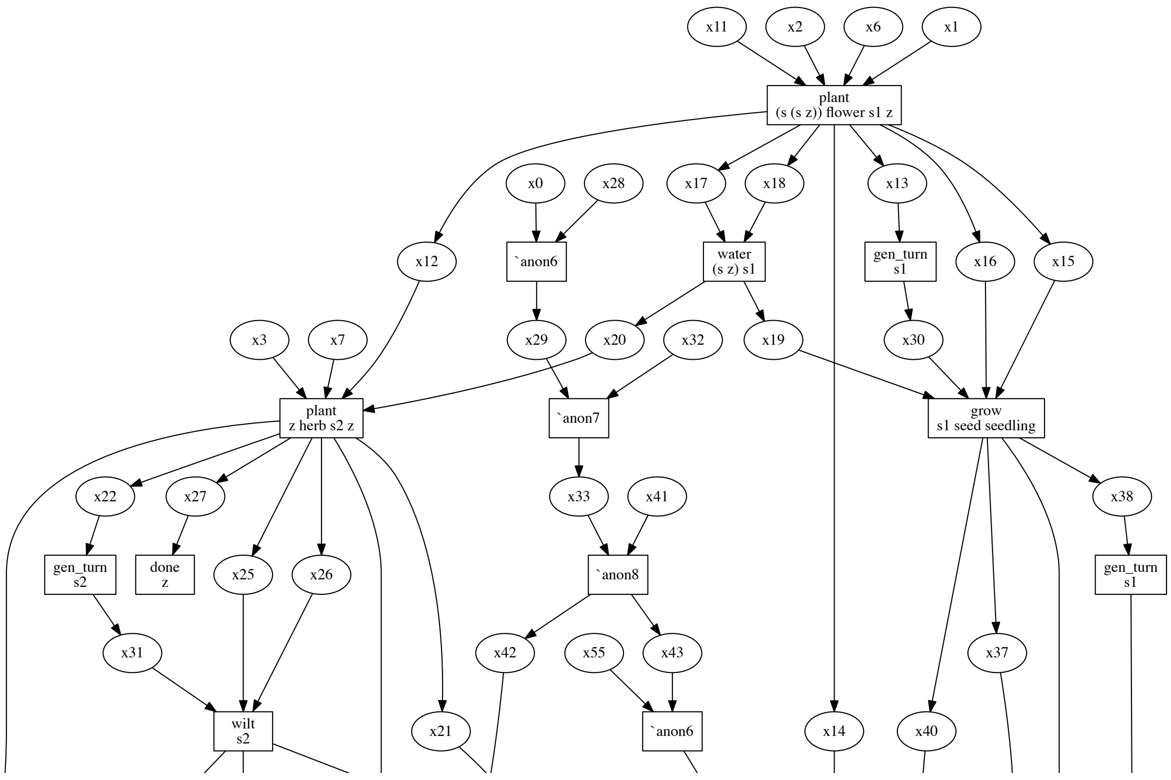
```
step : pred.
turns nat : pred.

stage play {
  get_seeds :
    turns (s N) * harvested Type
      -o have_seed Type * have_seed Type * turns N.

  plant :
    turns (s N) * have_seed Type * empty S * $day D
      -o plant_at S
          * plant_type_at S Type
          * plant_phase_at S seed
          * health_at S healthy
          * needs_water S
          * turns N.

  harvest :
    turns (s N) * plant_at S
    * plant_phase_at S mature * health_at S healthy
    * plant_type_at S Type
      -o harvested Type * empty S * turns N.

  clear :
    turns (s N) * plant_at S * health_at S dead
    * plant_phase_at S _ * plant_type_at S _
      -o empty S * turns N.

  water :
    turns (s N) * needs_water S
      -o watered S * turns N.

  buy_space :
    turns (s N) * harvested P
    * max_space S * next_space S S'
    -o empty S' * max_space S'.

  done : turns _ -o ().
}
#interactive play.
```

Now that we have separated out the plant type, plant phase, and health information of a soil space, it is easier to use the "generate one turn per plant" technique wherein one predicate (`plant_at`) is replaced by a temporary turn predicate (`plant_turn`), then consumed and replaced with the original predicate in each rule of the growth stage. Note that this trick depends on carefully maintaining the invariant that every plant will consume a `plant_turn` atom and produce a `plant_at` atom during the `grow` stage.

Maintaining this invariant was the source of several bugs found while developing the example.

```
qui * stage play -o stage advance_day.

plant_turn space : pred.
stage advance_day = {
  gen_turn : plant_at S -o plant_turn S.
}

qui * stage advance_day -o stage grow.

stage grow = {
  grow : plant_turn S * $health_at S healthy * watered S
       * plant_phase_at S Phase * next Phase Phase'
            -o plant_at S * plant_phase_at S Phase' * needs_water S.

  wilt : plant_turn S * health_at S healthy * $needs_water S
            -o plant_at S * health_at S wilting.

  drink : plant_turn S * health_at S wilting * watered S
            -o plant_at S * health_at S healthy * needs_water S.

  die : plant_turn S * health_at S wilting * needs_water S
            -o plant_at S * health_at S dead.

  none/dead
    : plant_turn S * $health_at S dead -o plant_at S.

  none/mature
    : plant_turn S * $health_at S healthy * watered S
      * $plant_phase_at S mature
      -o plant_at S * needs_water S.

  needwater/empty
    : $empty S * needs_water S -o ().

  watered/empty
    : $empty S * watered S -o ().
}
qui * stage grow -o stage play * turns (s (s (s z))).

context init = {day z,
  have_seed flower, have_seed herb, have_seed vine, have_seed succulent,
  empty s1, empty s2, empty s3, empty s4, max_space s4,
  turns (s (s (s z)))}.
#trace _ play init.
```

Figure 5.8: Partial trace graph for third garden simulator iteration.



Sample Interaction

In Figure 5.8, we give a partial trace graph for a sample interaction with the system.

5.4.5 Discussion

With this garden simulation case study, we have demonstrated an iterative design process for an original game mechanic based on interacting with a simulated, virtual world. Potential extensions that could be developed in future iterations include varying the plant types, such as making it so that succulents require less water, and modeling plant overgrowth, such as vines becoming arbitrarily long and threatening the health of other plants.

The plant growth stage can be seen as an application of generative methods, albeit a simple one akin to a state machine. Generative methods have perhaps been more applied to the domain of plant growth (e.g. tree generation) than any other domain [LD99, GPSL03, DHL^{+}98, Pru86], and we posit that more sophisticated techniques could be modeled as an extension to this prototype to interesting effect, especially with visual rendering.

This model could be considered a prototype for a more extensive game about the creation and maintenance of peaceful worlds, an under-explored space of design in mainstream games. We have aimed to illustrate with this example the use of Ceptre, and especially programming idioms that have been common to other examples, for the design of a novel operational logic based on simple generative techniques.

5.5 Tamara

Tamara is a work of participatory theater by John Krizanc [Kri89]. The form of participation is quite limited: guests are allowed to *follow* characters when they depart a scene, after which point the departing characters will participate in a different scene, either by joining an ongoing one or starting their own. We use this play as an example of interactive storytelling that does not involve digital computers, yet has *computational content* in a sense: many traversals through the play's story world are available by way of simultaneous scenes and the option to move between them when a character connects them.

This case study is unlike the others because we do not attempt to implement the interactive experience of attending *Tamara* as an interactive linear logic program. Instead, the study we have carried out pertaining to this play is twofold:

- First, we implement a transformation from the *script* of the play—in particular, information about which characters participate in which scenes, and where they go afterwards—to a dependency diagram that gives a holistic sense for which scenes may co-occur and where synchronization is needed. In some sense, this analysis could be considered a producer or stage manager's tool for orchestrating the logistics of the play. Let us refer to the dependency diagram as a *scene graph*.

  The transformation from script to scene graph is done by encoding the script as a linear logic program, then using proof search to create the scene graph (represented as a concurrently-structured proof term).

- Next, we implement a compiler that takes a proof term (representing the scene graph) along with a *scenes* file, which maps linear logic rules to the text (dialogue) of the scene, and turns it into a Twine game whose passages are scenes and whose links are choices to either follow a character who is exiting or remain with the other characters.

  In this way, linear logic programming is only used as an intermediate representation between the informal script and the interactive performance (Twine game).

5.5.1 Tamara's Story, Script, and Rules

Tamara is a historically-based tale of aristocrats and servants, set in the mansion of Italian poet Gabriele d'Annunzio in the 1920s, pivoting around the event of the Polish artist Tamara de Lempicka's arrival as an artist in residence. The characters involved are artists previously invited to the d'Annunzio estate, many of whom d'Annunzio took as lovers, and the serving staff.

The idea for the play came out of a discussion between Krizanc and his friends about a Chekhov play, wherein he expressed wishing he could follow the servants when they exited a scene, because they "were the only ones to know the underbelly of the social fabric." As such, the characters in Tamara include a large cast of servants —Emilia, the maid; Dante and Mario, servants; Finzi, the guard; and Aelis, the housekeeper—in addition to the five wealthy artists who live in or visit the d'Annunzio estate—Luisa, de Spiga, Carlotta, Tamara, and D'Annunzio himself.

From an audience member's perspective, the play's mechanics are as follows: after arrival, guests are assembled, then divided into three starting groups. Each group follows a different character, who leads them to a different room in the mansion. At that point, the play begins in earnest, and the standard rules apply: whenever a character exits a room, you may follow them. You must always be following or watching the action of a character—that is, if a character is the *only* one in the room when they exit, you are obligated to follow them. However, you need not follow the same character throughout the play.

The script for the play is organized into standard acts, which are further subdivided *sections* labeled with an alphabetic character $A \dots U$, each further subdivided into numerically-labeled scenes. Sections designate some approximation of concurrent structure: their endings usually just follow substantial divergence or fragmentation in the characters' story arcs, and their beginnings give an overview of the pursuant simultaneous scenes (see Figure 5.9).

Note that, despite the story's "branching" structure, it is actually deterministic, as in, the play is performed the same way every time. In our terminology from Chapter 2, the story is purely *simultaneous* and contains no *alternative* nonlinearity. Thus, the logic program that we write should be deterministic. Lacking metalogical checks on the program to ensure this fact for us, we must do so by maintaining carefully the invariant that no component of narrative state is consumed by multiple rules.

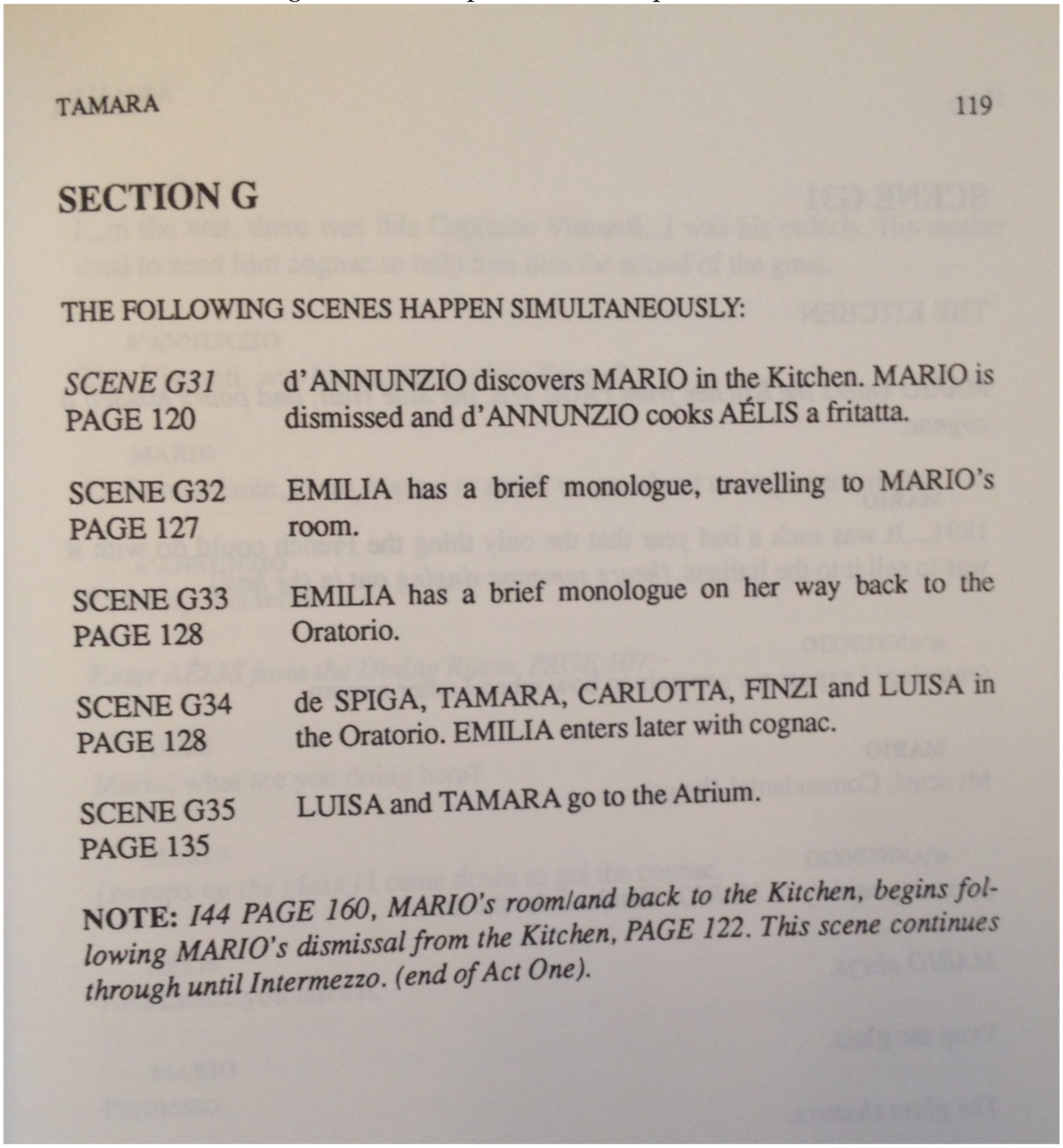
5.5.2 Encoding the Script Structure

Code for this section is available at `https://github.com/chrisamaphone/interactive-lp/blob/master/examples/tamara/tamara.cep`.

First, we must address the question of what a piece of narrative state should be. At first glance, it may seem that the answer is a *character* in the story. But since characters may occur in multiple scenes which need to be ordered with respect to one another, we will need to track more information in order to maintain determinism.

Previously, our approach has been to decompose the story into narrative resources that implicitly drive the causal model underlying the story. For instance, in the *Three Little Pigs* formalization from Chapter 2, we incorporated the narrative resource of the "brick house" state as an output of one scene (the building of the brick house) and input of another (the wolf confronting the pig in the brick house) to ensure that these scenes happened in a plausible causal order.

Figure 5.9: Excerpt from the script of *Tamara*



TAMARA 119

SECTION G

THE FOLLOWING SCENES HAPPEN SIMULTANEOUSLY:

SCENE G31 PAGE 120	d'ANNUNZIO discovers MARIO in the Kitchen. MARIO is dismissed and d'ANNUNZIO cooks AÉLIS a fritatta.
SCENE G32 PAGE 127	EMILIA has a brief monologue, travelling to MARIO's room.
SCENE G33 PAGE 128	EMILIA has a brief monologue on her way back to the Oratorio.
SCENE G34 PAGE 128	de SPIGA, TAMARA, CARLOTTA, FINZI and LUISA in the Oratorio. EMILIA enters later with cognac.
SCENE G35 PAGE 135	LUISA and TAMARA go to the Atrium.

NOTE: *I44 PAGE 160, MARIO's room/and back to the Kitchen, begins following MARIO's dismissal from the Kitchen, PAGE 122. This scene continues through until Intermezzo. (end of Act One).*

For this study, however, we are less interested in discovering emergent causal relationships: we just want to encode the one that is already present in the script for the play. Thus, to synchronize scenes deterministically, we make narrative states that represent characters *at a certain time and place*—where time is abstracted as a scene index. Concretely, we set up the encoding of sections A through C in the script as follows:

```
scene : type.
a1 : scene.  b2 : scene.
b3 : scene.  b4 : scene.  b5 : scene.
b6 : scene.  b7 : scene.  b8 : scene.
b9 : scene.  b10 : scene.  b11 : scene.
c12 : scene.  c13 : scene.  c14 : scene.
d15 : scene.  d16 : scene.  d17 : scene.

location : type.
sidehall      : location.
atrium        : location.
hall          : location.
oratorio      : location.
diningroom    : location.
luisaroom     : location.
dannunzioroom : location.
leda          : location.
above-atrium  : location.
finzi-office  : location. % 10 locations.
offstage      : location.

%% characters
tamara    scene location : pred.
luisa     scene location : pred.
emilia    scene location : pred.
finzi     scene location : pred.
despiga   scene location : pred.
aelis     scene location : pred.
dannunzio scene location : pred.
carlotta  scene location : pred.
dante     scene location : pred.
mario     scene location : pred. % 10 characters.
```

The initial setup positions every character offstage for scene $A1$ except for de Spiga, who starts the show, and Tamara, who does not enter until scene $D15$.

```
context
init = {
  tamara    d15 atrium,
  luisa     a1 offstage,
  emilia    a1 offstage,
  finzi     a1 offstage,
  despiga   a1 atrium,
  aelis     a1 offstage,
  dannunzio a1 offstage,
  carlotta  a1 offstage,
  dante     a1 offstage,
  mario     a1 offstage
}.
```

Now we can write rules that correspond to the character coordination that must take place for each scene. The first several rules manage the rapid entering-and-exiting of the introductory scene of the play, which the whole audience actually witnesses as in an ordinary stage play:

```
stage all = {
%%%% SECTION A %%%%

a1_dante_finzi
: dante a1 offstage * finzi a1 offstage -o dante a1 atrium * finzi a1 atrium.

a1_emilia_enters
: emilia a1 offstage -o emilia a1 atrium.

a1_dannunzio_enters
: dannunzio a1 offstage -o dannunzio a1 atrium.

a1_dannunzio_exits
: dannunzio a1 atrium -o dannunzio a1 dannunzioroom.

a1_emilia_exits
: emilia a1 atrium -o emilia a1 sidehall.

a1_aelis_enters
: aelis a1 offstage -o aelis a1 atrium.

a1_carlotta_enters
: carlotta a1 offstage -o carlotta a1 atrium.

a1_carlotta_and_aelis_exit
: aelis a1 atrium * carlotta a1 atrium
           -o carlotta b4 hall * aelis a1 diningroom.

a1_mario_enters
: mario a1 offstage -o mario a1 atrium.

a1_mario_exits
: mario a1 atrium -o mario d15 atrium.
```

Then the audience is split into three groups, each following one of Dante, Finzi, or de Spiga:

```
a1_groups_split
: dante a1 atrium * finzi a1 atrium * despiga a1 atrium
       -o  dante a1 luisaroom
           * finzi a1 diningroom
           * despiga b2 atrium .
```

At this point, the audience is divided and the story becomes truly simultaneous. Some of the groups subdivide further.

```
% Dante takes group 1 to Luisa and D'Annunzio, then heads to the atrium
% alone.
a1_grp1_luisa
: dante a1 luisaroom * luisa a1 offstage
-o dante a1 dannunzioroom * luisa b3 luisaroom.

a1_grp1_dannunzio
: dante a1 dannunzioroom * dannunzio a1 dannunzioroom
-o dante b10 atrium * dannunzio b6 dannunzioroom.

% Finzi takes group 2 to Aelis, then Emilia, then meets Carlotta in the
% hall.
a1_grp2_aelis
: finzi a1 diningroom * aelis a1 diningroom
-o finzi a1 sidehall * aelis b9 diningroom.

a1_grp2_emilia
: finzi a1 sidehall * emilia a1 sidehall
-o finzi b4 hall * emilia b7 leda.
```

Note the strict pattern that these rules follow: every character predicate on the premise side of the rule must match in scene and room identifier, and must be replaced on the consequent side of the rule with a corresponding character predicate in a later scene.

The remaining rules for sections B and C of the play follow this same pattern, and are produced below for completeness, although we do not expect them to be particularly meaningful to the reader without the script of the play itself.

```
%%%% SECTION B %%%%

% Group 3 stays in the Atrium with de Spiga.
b2_despiga_monologue
: despiga b2 atrium -o despiga b10 atrium.

b3_luisa_monologue
: luisa b3 luisaroom -o luisa b11 dannunzioroom.

b4_carlotta_kisses_finzi
: finzi b4 hall * carlotta b4 hall
-o finzi b5 hall * carlotta b8 leda.
```

```
b5_finzi_monologue
: finzi b5 hall -o finzi c12 atrium.

b6_dannunzio_monologue
: dannunzio b6 dannunzioroom -o dannunzio b11 dannunzioroom.

b7_emilia_monologue
: emilia b7 leda -o emilia b8 leda.

b8_emilia_carlotta
: emilia b8 leda * carlotta b8 leda
-o emilia c12 diningroom * carlotta c12 atrium.

b9_aelis_monologue
: aelis b9 diningroom -o aelis c12 atrium.

b10_dante_despiga
: dante b10 atrium * despiga b10 atrium
-o dante c12 atrium * despiga c12 atrium.

b11_dannunzio_luisa
: dannunzio b11 dannunzioroom * luisa b11 dannunzioroom
-o dannunzio c12 atrium * luisa c12 above-atrium.

%%%% SECTION C %%%%

c12_finzi_exit
: %% de Spiga, Dante, Aelis, Finzi, Carlotta, D'Annunzio
  despiga   c12 atrium
* dante     c12 atrium
* aelis     c12 atrium
* finzi     c12 atrium
* carlotta  c12 atrium
* luisa     c12 above-atrium
-o   despiga   c12 atrium
    * dante     c12 atrium
    * aelis     c12 atrium
    * finzi     c12 finzi-office
    * carlotta  c12 atrium
    * luisa     c12 above-atrium.

c12_gunshot
: luisa c12 above-atrium * finzi c12 finzi-office * emilia c12 diningroom
-o luisa c12 atrium * finzi c12 atrium * emilia c12 atrium.
```

```
c12_emilia_carlotta_aelis_luisa_leave_atrium
: emilia c12 atrium * carlotta c12 atrium * aelis c12 atrium
  * luisa c12 atrium
-o emilia c13 sidehall * carlotta c13 sidehall * aelis c13 sidehall
    * luisa c13 atrium.

c13_finzi_luisa_confrontation
: finzi c12 atrium * luisa c13 atrium
-o finzi d15 atrium * luisa c13 sidehall.

c13_emilia_aelis_carlotta_luisa_sidehall
: emilia c13 sidehall * aelis c13 sidehall * carlotta c13 sidehall
* luisa c13 sidehall
%% all exit back to atrium
-o emilia c13 atrium * aelis c14 atrium * carlotta c13 atrium
* luisa c13 atrium.

c12_take_presents_to_leda
: dante c12 atrium * carlotta c13 atrium * luisa c13 atrium
* emilia c13 atrium
-o dante c14 leda * carlotta c14 leda * luisa c14 leda
* emilia c14 atrium.

c14_emilia_enter_leda
: dante c14 leda * carlotta c14 leda * luisa c14 leda * emilia c14 atrium
-o dante c14 leda * carlotta c14 leda * luisa c14 leda * emilia c14 leda.

c14_all_exit_leda
: dante c14 leda * carlotta c14 leda * luisa c14 leda * emilia c14 leda
-o dante c14 atrium * carlotta c14 atrium * luisa c14 atrium
    * emilia c14 atrium.
%% n.b. technically they go "back" to scene c12, but this disrupts
%% a useful form of stratification...

c_final
: luisa c14 atrium * carlotta c14 atrium * dante c14 atrium
* emilia c14 atrium * aelis c14 atrium * dannunzio c12 atrium
* despiga c12 atrium
-o  luisa d17 oratorio * carlotta d17 oratorio
   * aelis d17 oratorio * despiga d17 oratorio
   * dante d15 atrium * dannunzio d15 atrium
   * emilia d15 diningroom.
}
```

The program produced above is simply a by-hand transliteration of information found in the script. With it, however, we can use proof search to generate the concurrent structure of the play.

The raw proof term generated by a query or trace on this program is always concurrently equal to the following:

```
let {[X2, [X3, [X4, [X5, [X6, [X7, [X8, [X9, [X10, X11]]]]]]]]]} = X1 in
let {X12} = a1_carlotta_enters X9 in
let {X13} = a1_dannunzio_enters X8 in
let {X14} = a1_mario_enters X11 in
let {X15} = a1_emilia_enters X4 in
let {X16} = a1_dannunzio_exits X13 in
let {[X17, X18]} = a1_dante_finzi [X10, X5] in
let {X19} = a1_aelis_enters X7 in
let {X20} = a1_emilia_exits X15 in
let {[X21, [X22, X23]]} = a1_groups_split [X17, [X18, X6]] in
let {X24} = a1_mario_exits X14 in
let {X25} = b2_despiga_monologue X23 in
let {[X26, X27]} = a1_carlotta_and_aelis_exit [X19, X12] in
let {[X28, X29]} = a1_grp1_luisa [X21, X3] in
let {X30} = b3_luisa_monologue X29 in
let {[X31, X32]} = a1_grp1_dannunzio [X28, X16] in
let {[X33, X34]} = b10_dante_despiga [X31, X25] in
let {X35} = b6_dannunzio_monologue X32 in
let {[X36, X37]} = b11_dannunzio_luisa [X35, X30] in
let {[X38, X39]} = a1_grp2_aelis [X22, X27] in
let {X40} = b9_aelis_monologue X39 in
let {[X41, X42]} = a1_grp2_emilia [X38, X20] in
let {X43} = b7_emilia_monologue X42 in
let {[X44, X45]} = b4_carlotta_kisses_finzi [X41, X26] in
let {[X46, X47]} = b8_emilia_carlotta [X43, X45] in
let {X48} = b5_finzi_monologue X44 in
let {[X49, [X50, [X51, [X52, [X53, X54]]]]]}
  = c12_finzi_exit [X34, [X33, [X40, [X48, [X47, X37]]]]] in
let {[X55, [X56, X57]]} = c12_gunshot [X54, [X52, X46]] in
let {[X58, [X59, [X60, X61]]]}
  = c12_emilia_carlotta_aelis_luisa_leave_atrium
    [X57, [X53, [X51, X55]]] in
let {[X62, X63]} = c13_finzi_luisa_confrontation [X56, X61] in
let {[X64, [X65, [X66, X67]]]}
  = c13_emilia_aelis_carlotta_luisa_sidehall [X58, [X60, [X59, X63]]] in
let {[X68, [X69, [X70, X71]]]}
  = c12_take_presents_to_leda [X50, [X66, [X67, X64]]] in
let {[X72, [X73, [X74, X75]]]}
  = c14_emilia_enter_leda [X68, [X69, [X70, X71]]] in
let {[X76, [X77, [X78, X79]]]}
  = c14_all_exit_leda [X72, [X73, [X74, X75]]] in
let {[X80, [X81, [X82, [X83, [X84, [X85, X86]]]]]]}
  = c_final [X78, [X77, [X76, [X79, [X65, [X36, X49]]]]]]
```

The visualized concurrent structure of this term is given in Figures 5.10 and 5.11.

Figure 5.10: Trace graph for *Tamara* encoding, part 1.

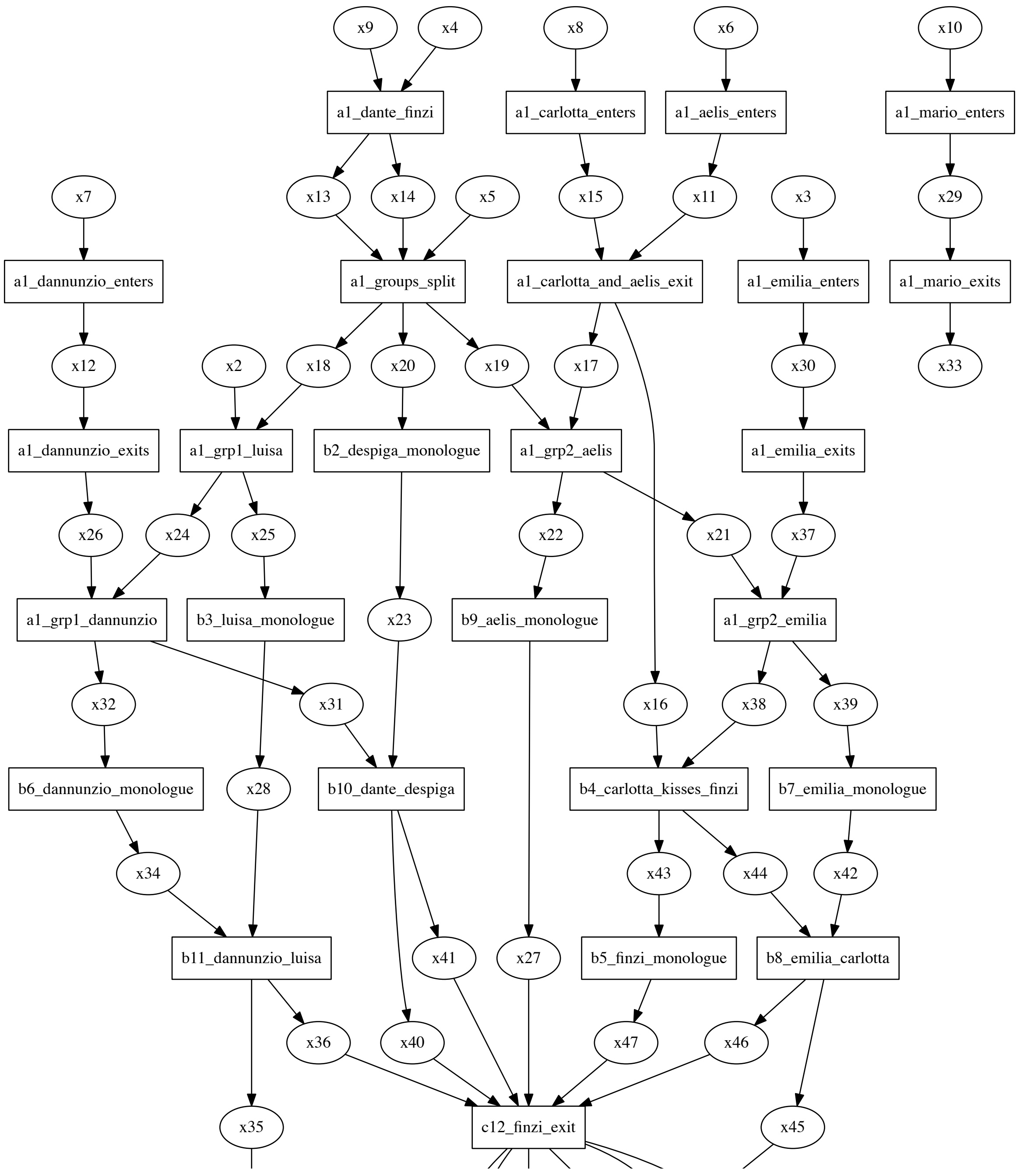
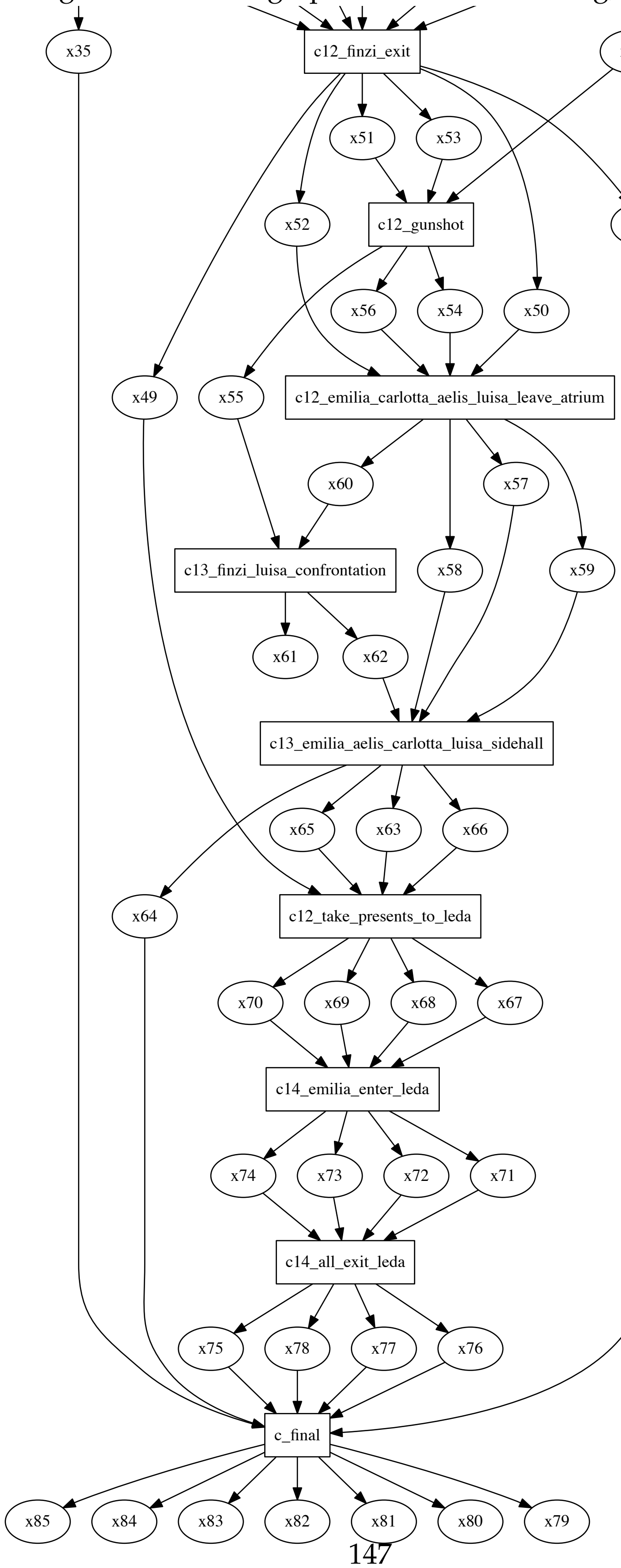


Figure 5.11: Trace graph for *Tamara* encoding, part 2.



In Celf, such a trace might result from the following query:

```
init -o
{ tamara    STamara  LTamara
* luisa     SLuisa   LLuisa
* emilia    SEmilia  LEmilia
* finzi     SFinzi   LFinzi
* despiga   SDeSpiga LDeSpiga
* aelis     SAelis   LAelis
* dannunzio SDannunzio LDannunzio
* carlotta  SCarlotta  LCarlotta
* dante     SDante     LDante
* mario     SMario     LMario}.
```

This query would additionally generate the final locations and scenes of every character:

```
#LAelis = oratorio
#LCarlotta = oratorio
#LDannunzio = atrium
#LDante = atrium
#LDeSpiga = oratorio
#LEmilia = diningroom
#LFinzi = atrium
#LLuisa = oratorio
#LMario = atrium
#LTamara = atrium

#SAelis = d17
#SCarlotta = d17
#SDannunzio = d15
#SDante = d15
#SDeSpiga = d17
#SEmilia = d15
#SFinzi = d15
#SLuisa = d17
#SMario = d15
#STamara = d15
```

5.5.3 Compilation to Twine

We next turn this structured trace, the CLF proof term, into a *playable drama* with the same form of interaction as the play itself: following (or staying with) characters. Doing so involves a very simple trick: we re-interpret the *simultaneous* branching in the story graph as *alternative* branching. That is, every out-resource of a scene can be interpreted as a choice to "follow" that resource to wherever it is needed next.

To realize this idea, we use the *Quiescent Theater* project described in Chapter 3. To recapitulate the idea of the project: we wrote a compiler from linear logic proof terms to runnable Twine source code (also known as Twee). To make the output intelligible, we also introduced *scene files*, or mappings from linear logic program rules onto legible text, as follows:

```
Tamara
By John Krizanc, adapted by Chris Martens.

:: initial 2
Dante and Finzi welcome in the guests. "This is the home of Gabriele d'Annunzio,"
Dante informs you.

:: a1_dante_finzi 2
Dante and Finzi enter the atrium.

:: a1_emilia_enters 1
Emilia enters the atrium.

:: a1_dannunzio_enters 1
D'Annunzio enters the atrium.

:: a1_dannunzio_exits 1
D'Annunzio departs the atrium for his bedroom.

:: a1_emilia_exits 1
Emilia departs the atrium for the side hall.

:: a1_carlotta_enters 1
Carlotta enters the atrium.

:: a1_aelis_enters 1
Aelis enters the atrium.

:: a1_carlotta_and_aelis_exit 2
Carlotta exits toward the hall, and Aelis toward the dining room.

:: a1_mario_enters 1
Mario enters the atrium.

:: a1_mario_exits 1
Mario leaves the atrium.

:: a1_groups_split 3
Dante: "I'm going to go check on Signora Baccara."
Finzi: "I will visit the servants of the house, Aelis and Emilia."
Dante departs for Luisa's room, Finzi for the dining hall, and De Spiga
remains in the atrium.
```

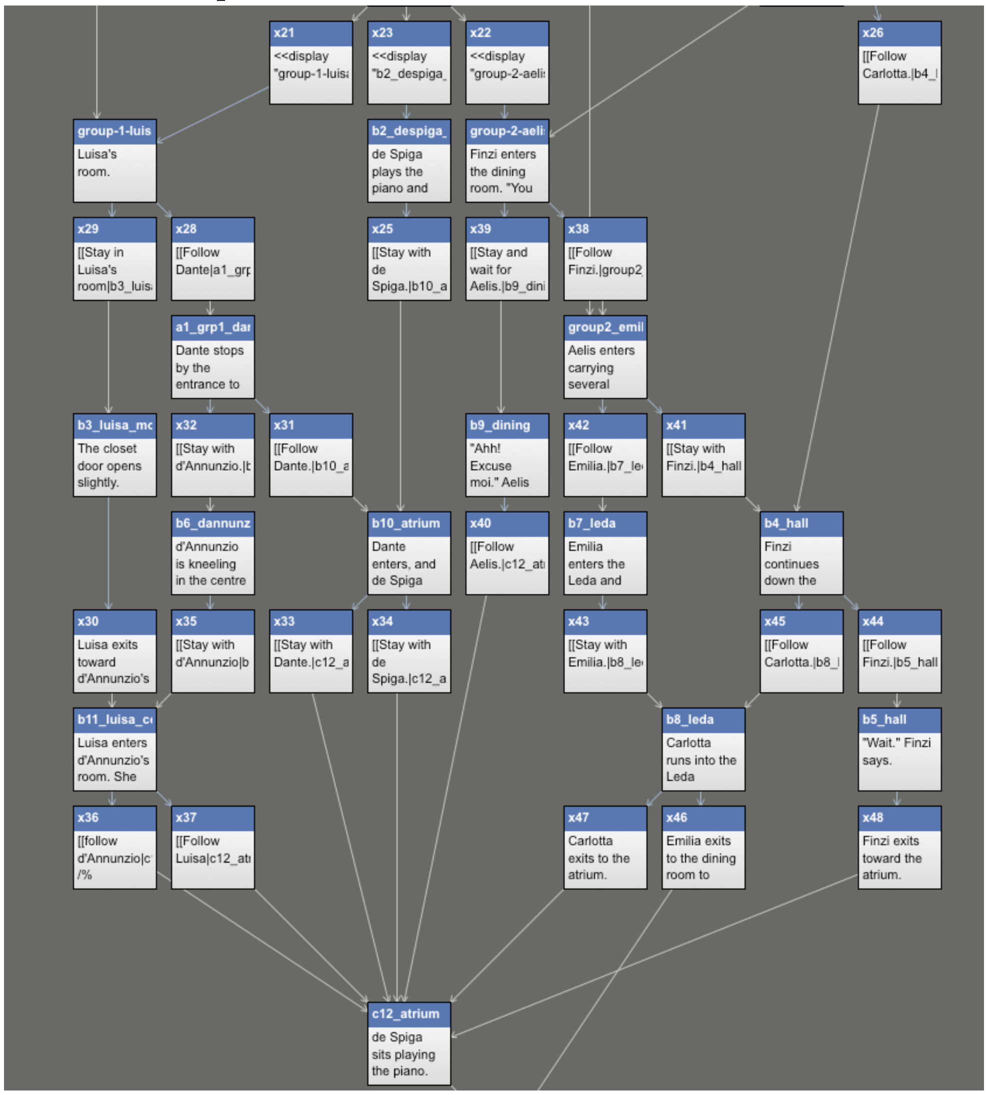
```
:: a1_grp1_luisa 2
Dante enters Luisa's room. "Signora Baccara?" She is nowhere to be seen.
Dante departs Luisa's room for D'Annunzio's room.
```

...and so on. The number following the rule name (e.g. 3 in `::a1_groups_split 3` signifies the number of "followable" resources (characters) in the scene.

The remaining scene rules can be found on the GitHub page linked above. Again, these formal entities are simply transliterations of Krizanc's script, encoding complementary information to the linear logic program rules.

The compiler then works as follows: taking the linear logic program and scene file as inputs, its runs the logic program to produce a structured proof, parses that proof into the Twee skeleton, and instantiates the passages with text and links as specified by the scene file.

The resulting Twine game for Tamara has the same essential structure as the CLF proof term. A sample of it is shown below:


x21 <<display "group-1-luis
x23 <<display "b2_despiga_
x22 <<display "group-2-aeli
x26 [[Follow Carlotta.|b4_|
group-1-luis Luisa's room.
b2_despiga_ de Spiga plays the piano and
group-2-aeli Finzi enters the dining room. "You
x29 [[Stay in Luisa's room|b3_luis
x28 [[Follow Dante|a1_grp
x25 [[Stay with de Spiga.|b10_a
x39 [[Stay and wait for Aelis.|b9_dini
x38 [[Follow Finzi.|group2
a1_grp1_dai Dante stops by the entrance to
group2_emil Aelis enters carrying several
b3_luisa_mc The closet door opens slightly.
x32 [[Stay with d'Annunzio.|t
x31 [[Follow Dante.|b10_a
b9_dining "Ahh! Excuse moi." Aelis
x42 [[Follow Emilia.|b7_le
x41 [[Stay with Finzi.|b4_hall
b6_dannunz d'Annunzio is kneeling in the centre
b10_atrium Dante enters, and de Spiga
x40 [[Follow Aelis.|c12_at
b7_leda Emilia enters the Leda and
b4_hall Finzi continues down the
x30 Luisa exits toward d'Annunzio's
x35 [[Stay with d'Annunzio|b
x33 [[Stay with Dante.|c12_a
x34 [[Stay with de Spiga.|c12_a
x43 [[Stay with Emilia.|b8_le
x45 [[Follow Carlotta.|b8_|
x44 [[Follow Finzi.|b5_hall
b11_luisa_c Luisa enters d'Annunzio's room. She
b8_leda Carlotta runs into the Leda
b5_hall "Wait." Finzi says.
x36 [[follow d'Annunzio|c /%
x37 [[Follow Luisa|c12_at
x47 Carlotta exits to the atrium.
x46 Emilia exits to the dining room to
x48 Finzi exits toward the atrium.
c12_atrium de Spiga sits playing the piano.

It can be played at `http://play.typesafety.net/world/tamara/`.

This project was joint work with Rob Simmons. We ran our compiler on a variety of linear logic programs and scene files, several of which were nondeterministic, unlike Tamara. The results can be found at `http://play.typesafety.net`.

5.5.4 Discussion

We have described a computational study of the theatrical play, *Tamara*, using linear logic to realize its simultaneous structure. We used linear logic as an intermediate language to transform the script of the play into a playable digital format, simulating the experience of an audience member.

Participatory theater has seen a surge in popularity and innovation in recent years, notably including the theatre company Punch Drunk and its acclaimed work *Sleep No More*, an immersive "installation art" performance [Pep11]. We imagine that authoring these works in such a way to guarantee *implementability* is rather difficult, especially in the presence of audience interaction that may chance the course of the story. We propose that computational tools might broaden participation in the authoring of such complex theatrical productions, perhaps making it more inviting to artists in game design as well. We also suggest that such tools could be an aid to producers, directors, and stage managers of such performances, allowing them to extract important dependency information between scenes without computing it by hand.

5.6 Discussion of Case Studies

We have presented five case studies, giving what we estimate to be a representative sample of the range of expressiveness for game and story mechanics in linear logic. We now discuss the strengths and limitations of linear logic programming for the domains we have studied.

First, there are a few common patterns in games that we have *not* illustrated in the examples above, but which would be feasible to build. The first of these is *endings* in games—so far, we have mainly discussed win-condition-free, open sandbox worlds (with the one exception of Tamara, which has a single deterministic ending). A common class of games with important "win state" detection is puzzle games, often stratified into levels where winning one level is a requirement for advancing to the next.

For an example, take Sokoban, the 2d tile-based crate-pushing game. Sokoban levels typically have *targets* on which all crates must be present to advance the level. Here is how we could codify that win condition check in Ceptre:

```
target tile : pred.
crate_at tile : pred.

uncounted_target tile : pred.
maybe_win : pred.
no_win : pred.
```

```
stage gen_uncounted = {
  target T -o uncounted_target T.
}
qui * stage gen_uncounted -o stage count.

stage count = {
  uncounted_target T * $crate_at T -o target T.
}
qui * stage count -o stage check * maybe_win.

stage check = {
  uncounted_target T * maybe_win -o target T * no_win.
}

win : pred.
qui * stage check * no_win -o stage play.
qui * stage check * maybe_win -o win * stage win.

stage play = {
 % ...
}

stage win = {
  win * level L * next_level L L' -o level L'.
}
% ...

t1 : tile.
t2 : tile.
t3 : tile.
t4 : tile.
context test = {
  target t1,
  target t3,
  crate_at t1,
  crate_at t2,
  crate_at t3}.

#trace _ gen_uncounted test.
```

While the idea of *level stratification* is itself not something we have covered with these examples, it should be clear by now how one could accomplish it with stages.

The other very common type of mechanic we have not discussed is the passage of time independent from player actions, or relatedly, asynchronous player input. Both of these concepts can be implemented as sensing predicates.

5.6.1 Limitations

There are a few common programming patterns that linear logic programming is *not* well-suited for, and we want to state these limitations to clarify what should be considered out of scope for the programming language in its current implementation (and that other systems do better).

- **Negation.** Any rule can check for the *presence* of an atom in the world state, but cannot check for its absence. As noted in Section 4.3.2, we can codify this behavior with stages, but it is often too cumbersome to do so in practice.

  The reason why negation is left out of the basic logical formalism is that it violates the critical *frame property* of the logic: whenever it is the case that $\Delta \to \Delta'$ is a valid transition, it should also be true that $\Delta, A \to \Delta', A$ is a valid transition. (One reason that this property is so important is that it lets us understand rules with respect to an "open world," i.e. any arbitrary context.) But if transitions can be contingent on the *absence* of a resource A, then this property no longer holds. Simmons and Toninho [ST12] suggest a logically sound basis for introducing negation under certain conditions that preserve this principle, but investigating the incorporation of those ideas into an implementation of Ceptre is left to future work.
- **Symmetric predicates.** Predicates ranging over two terms often want to be symmetric, i.e. whenever `p(A,B)`, `p(B,A)` also holds (especially in social story worlds). If we want this to be the case, we have to manage this invariant by hand.
- **Boolean or exclusive predicates.** Many predicates have a usage pattern in which there is never more than one copy, or in which they are mutually exclusive with another predicate to represent a Boolean state, or in which there is always exactly one term `X` for which `p(X)`. These patterns again must obey invariants that are maintained by the programmer by hand.
- **Complex geometric calculations.** Our framework would probably be ill-suited to modeling games involving rapid computation of convex hulls or three-dimensional collision boundaries, unless those computations were encapsulated as sensing and acting predicates (i.e. programmed outside the confines of Ceptre).
- **Time sensitivity.** We do not yet have any data collected about whether Ceptre programs would be able to meet "frames-per-second" benchmarks for time-sensitive games. In general, we do not imagine Ceptre as the engine driving a released, production-quality commercial game—we envision it as a prototyping language, one part of a designer's toolbox out of many.

5.6.2 Strengths

Linear logic programming has strengths as a prototyping tool that together position it as uniquely well-suited for game design prototyping. We believe these strengths lie in the modeling of game economies, multi-agent systems, generative methods, and interactive storytelling.

- **Game economies.** Linear logic programming shares with the Machinations [Dor11] approach to simulating game mechanics its affordances for describing *game economies*, or the resource exchanges that take place during player and automatic actions. This strength is exemplified by DungeonWorld, GardenWorld, and SiedlerWorld.
- **Multi-agent systems.** The use of first-order logic enables us additionally to specify complex multi-agent systems where each agent is privy to the same rules, as exemplified in BuffyWorld.
- **Interactive storytelling.** Both BuffyWorld and Tamara exemplify the expressiveness of linear logic for codifying interactive story mechanics that involve the coherent coordination of multiple actors, and the Quiescent Theater project shows how it can be used to generate structured, interactive stories.
- **Generative methods.** Finally, we argue that linear logic programming has great potential for modeling and experimenting with *generative methods* in games, based on the demonstrations in GardenWorld and BuffyWorld of generating plant life and television horror stories in collaboration with the player.

5.6.3 Potential

Some of the limitations of the system are, of course, only due to the bounded time and scope of this dissertation. Here we suggest high water-marks in future developments of and with linear logic-based programming methodologies. Two ideas in particular that we believe they can enable are:

Comparing Worlds

With the ability to express such a wide range of systems using a small number of programming constructs comes the ability to *compare* structure across domains. For instance, we notice that the program structure for the act/react architecture in BuffyWorld closely resembles that in GardenWorld; and the positive feedback loop pattern in GardenWorld itself resembles that in DungeonWorld. The desire to compare games structurally seems well-expressed by interest in *game design patterns* [BH04], but previous work on such patterns is typically restricted not only by operational logic but by specific formulations of game entities. With linear logical representations, we have a candidate for the formal expression of design patterns across distinct operational logics.

Combining Worlds

Along similar lines, being able to express distinct operational logics in the same logical framework could enable us to invent new game designs via a "mash-up" artistic approach, i.e. recombination. A possible future study could involve taking any pair of case studies in this chapter and exploring different ways of combining them, either by mapping the predicates from one domain into predicates of the other, or simply taking the union of the available player actions and state space, then adding new ways for those state spaces to interact.

Articulating Strategies

We have only very briefly, in Section 5.2, discussed possbilities for augmenting Ceptre programs with certain "AI" strategies. In general, such augmentations could be used for a variety of purposes, such as playtesting, implementing adversarial NPCs, or in-game guidance (see Treanor et al. [TZE$^+$15] for a recent survey of uses of AI techniques in game design). However, many such techniques depend on probabilistic sampling of a search space in order to adapt to the human player's techniques over time. Currently, Ceptre's implementation of randomness does not offer any guarantees about the distribution or offer any means of rule selection based on history. In order to maximize control over the programming of such agents, something like stochastic logic programming [Cus00] could potentially be integrated for modeling probability distributions and sampling.

5.6.4 Analyzing Game Dynamics

After articulating a set of mechanics (and optionally a particular player strategy), we may be interested in some emergent properties of the game rules. For instance, in a two-player, turn-based game, we can formalize a simple notion of *balance* as whether the first or second player in the game has a significant advantage, on average. Other game mechanic description languages such as Machinations [Dor11] include facilities for producing resource graphs and other summarization data over multiple (interactive or simulated) play-throughs of a game. Cursory investigations suggest that similar tools for Ceptre would be easy to build and helpful for our stated goals of aiding iteration and analysis of games.

Chapter 6

Reasoning About Linear Logic Programs

6.1 Introduction

Specifications in linear logic involve complex, interdependent state transformations that are not always easy to reason about, especially as programs grow. Many programs implicitly encode *invariants* that the programmer holds in their head while adding complexity. For instance, in the interactive fiction example described Section 4.6.3, the programmer must take great care not to write any rules that discard an atom of the form `at player Room`. The whole program's correctness hinges on the implict assumption that the player is always `at` some location in the world, and we can very easily create programs that break such assumptions. When the language permits writing such fragile code, we cannot make a strong claim about its usability, due to the potential for novices to make errors like these without the ability to know what they did wrong.

On the other end of a spectrum of expressive power for constraints, we can also consider full verification of high-level properties of a game such as solvability. Such constraints have been used in generative methods [SM11], including the automatic generation of playable games. Extending invariant checking to these domains would mean not only producing well-formed starting configurations (such as level configuration) but also proving that everything the player can *do* in a generated game or level does not break its well-formedness.

To these ends, we propose constructing automated reasoning tools for linear logic programs that enable the explicit statement and automatic checking of program invariants. In this chapter, we present partial progress toward this goal and discuss challenges to solving it in full generality. We present three candidate logics for stating invariants, several manual proofs illustrating each approach, and a decidability result for automatically verifying a fragment of the language.

6.1.1 Example: Blocks World

Consider the following specification of the "blocks world" domain from AI planning [Nil14]: the domain includes blocks stacked on a table and a robotic arm that can pick up blocks and set them down on top of one another or on the table.

```
block : type.
on block block : pred.    % one block on top of another.
on_table block : pred.    % the block is on the table.
clear block : pred.       % the block has nothing on it.
arm_holding block : pred. % robot arm holds the block.
arm_free : pred.          % robot arm holds nothing.

% pick up a block from the table.
pickup_from_table : on_table B * clear B * arm_free -o arm_holding B.

% put down a block on the table.
putdown_on_table  : arm_holding B -o on_table B * clear B * arm_free.

% pick up a block from atop another block.
pickup_from_block : on BHi BLow * clear BHi * arm_free
                      -o clear BLow * arm_holding BHi.

% put down a block on top of another block.
putdown_on_block  : clear BLow * arm_holding BHi
                      -o on BHi BLow * clear BHi * arm_free.
```

To fully specify the domain, we state some assumptions about well-formed states and interactions. For instance, we assume that blocks are arranged in linear stacks with the table at the bottom, we assume that the arm always holds at most one block, and we assume that there is a unique top block of every stack that is marked as "clear." In order for us to verify all of these assumptions, we need to know that they hold of any initial configuration of blocks, and we need to know that the rules in the program *preserve* the same assumptions. If this were a terminating program rather than a potentially-infinite simulation, we might also want to state and prove properties of its execution in terms of what *final* configurations look like, possibly as a function of initial configurations.

The hardest problem in the collection of problems just described seems to be *invariant preservation*, i.e. a property of a program that says its rules preserve some constraint on the shape of the context. The problem bears some similarity to that of specifying well-formedness constraints on LF contexts, which are expressed with *regular worlds* in the Twelf implementation [PS98], in that we want to specify general patterns or schema for predicates that must appear alongside one another pertaining to specific terms. Ideally, checking invariant preservation should not depend on a *domain instance*. An example of a domain instance for blocks world is a specification of the particular blocks and starting configuration in a given simulation, such as the following:

```
a : block.
b : block.
c : block.

context init =
{ on_table a, clear a,
  on_table b, on c b, clear c,
  arm_free}.
```

This part of the program can be thought of as *input* to the rule set, in the sense that the rules should make sense *for any* well-formed set of blocks and initial configuration. Thus a checking algorithm for invariant preservation ought not to depend on this part of the specification. [1]

To check initial states, final states, and invariants, we need a way of describing context schema (sets of configurations representing a property of interest) and a framework in which to express and prove the program's relationship to those schema. This chapter describes three candidate logics for stating and proving these properties: *meta-linear logic*, *generative signatures*, and *consumptive signatures*. We show examples of proofs carried out in each of these logics for domains and properties of interest to this thesis. We describe the limitations and potential for each approach's automatability, and for the case of generative signatures, we describe a decidability result for the propositional case of Ceptre. We conclude by suggesting avenues for further investigation of this still-open problem.

6.2 Meta-Linear Logic

The well-formedness conditions of the Blocks World domain can be decomposed into two constraints on blocks and one on the robot arm, which may be informally stated as follows:

1. The first constraint says a block must have a well-formed base: either it is on the table, it is on top of another block, or it is held by the robot arm.
2. The second constraint says a block must have a well-formed top: either it is clear, there is another block on top of it, or it is held by the robot arm.
3. The third constraint says that the robot arm must either be free or holding a block.

Note that each of these descriptions independently seem to describe some mutual exclusion between resources, which could be described through linear logic's additive disjunction operator $\oplus$. These properties do not *partition* the context, however. For instance, the robot arm holding a block can satisfy all three of them, but we do not want to suggest that there should be three robot arm-related resources, when in fact we intend there to be exactly one.

We can describe these possibly-overlapping properties independently in terms of *restrictions* of the context Δ to certain predicate sets related to certain entities (usually predicate indices) over which properties are described, written $\Delta|_{\{p_1,\ldots,p_n\}}$ where p_i are predicate patterns (predicates where each index either refers to a variable bound by a quantifier or a wildcard _ satisfied by any index). We can also quantify over these entities in the formula, universally at the outside and existentially on the inside. For instance, the three properties above map onto the following formal statements defining

[1] However, see Section 6.6.1 for an example of a specification that might break this assumption.

what it means for an argument context Δ to satisfy the invariant (abbreviating `on_table` to ot, `arm_holding` to ah, `arm_free` to af, and `clear` to cl):

- Invariant BW_1 (block bottom well-formedness):

$$\forall b : \mathsf{block}.\Delta\lfloor_{\{\mathsf{ot}\ b,\ \mathsf{on}\ b\ \_,\ \mathsf{ah}\ b\}} \vDash (\mathsf{ot}\ b) \oplus (\exists b'.\mathsf{on}\ b\ b') \oplus (\mathsf{ah}\ b)$$

- Invariant BW_2 (block top well-formedness):

$$\forall b : \mathsf{block}.\Delta\lfloor_{\{\mathsf{cl}\ b,\ \mathsf{on}\ \_\ v,\ \mathsf{ah}\ b\}} \vDash (\mathsf{cl}\ b) \oplus (\exists b'.\mathsf{on}\ b'\ b) \oplus (\mathsf{ah}\ b)$$

- Invariant BW_3 (arm well-formedness):

$$\Delta\lfloor_{\{\mathsf{ah}\ \_,\ \mathsf{af}\ \}} \vDash (\exists b.\mathsf{ah}\ b) \oplus \mathsf{af}$$

Once a specific ground term index is plugged into this formula and the restriction is calculated, satisfiability $\vDash$ is determined simply by producing a derivation in linear logic.

To check a *program rule*, we must parameterize over frame contexts in which the rule might apply. For example, consider the "pick up from table" rule, which has the form

```
ontable B * clear B * arm_free -o arm_holding B
```

In order for this rule to apply to a context Δ, that context must have the form

$$\Delta',\ \mathsf{ontable}\ b,\ \mathsf{clear}\ b,\ \mathsf{arm\_free}$$

(where b is a ground term), and the state after application of the rule will be

$$\Delta',\ \mathsf{arm\_holding}\ \ b$$

We quantify universally over the context Δ', which must be reasoned about parametrically. The invariant itself quantifies over block terms B, and to show that it holds of the resulting state, we must reason parametrically over those as well. The *locality* or frame property of linear logic transitions means that whichever ones might appear in Δ' but not in the rule will remain unchanged and easy to reason about. That leaves reasoning about the indices that do appear in the rule with respect to the invariant. [2]

An invariant I is preserved by a given rule taking a state Δ to a state Δ' exactly when $I(\Delta)$ implies $I(\Delta')$. We can show that each of the three parts of the blocks world invariant described above apply to the `pickup_from_table` rule as follows.

Proposition 6.2.1 (Rule preserves blocks world invariants)**.** *Given any* Δ',
let $\Delta = \Delta',\ \mathsf{ontable}\ b,\ \mathsf{clear}\ b, \mathsf{arm\_free}$.
For each $I \in \{BW_1, BW_2, BW_3\}$, *if* $I(\Delta)$ *then* $I(\Delta',\ \mathsf{arm\_holding}\ b)$.

[2] This convenience depends crucially on the "well-moded" nature of rules: any variables mentioned in the consequent of the rule must relate somehow to variables in the antecedent. This means that even with infinite term domains, we never need to reason inductively over all inhabitants.

Proof. **Invariant BW_1: `pickup_from_table` preserves block bottom well-formedness** Intuitively, this property holds because the only block affected is the one manipulated by the rule, and its bottom's initial $\mathsf{ontable}$ property is simply replaced by an equally satisfactory $\mathsf{arm\_holding}$ property. The proof follows.

By assumption (BW_1 holds of the input context),

$$\forall b'{:}\mathsf{block}.(\Delta',\ \mathsf{ontable}\ b,\ \mathsf{clear}\ b, \mathsf{arm\_free}\ )\lfloor_{\{\mathsf{ot}\ b', \mathsf{on}\ b'\ \_, \mathsf{ah}\ b'\}} \vDash (\mathsf{ot}\ b') \oplus (\exists b''.\mathsf{on}\ b'\ b'') \oplus (\mathsf{ah}\ b')$$

Let S be the restriction set given above, $\{\mathsf{ot}\ b', \mathsf{on}\ b'\ \_, \mathsf{ah}\ b'\}$.

What we need to show is that for all blocks c,

$$(\Delta',\ \mathsf{arm\_holding}\ b)\lfloor_{\{\mathsf{ot}\ c, \mathsf{on}\ c\ \_, \mathsf{ah}\ c\}} \vDash (\mathsf{ot}\ c) \oplus (\exists b''.\mathsf{on}\ c\ b'') \oplus (\mathsf{ah}\ c)$$

Assume an arbitrary block c for which to prove this fact.

There are two cases: $c = b$ (the block referred to in the particular transition) or $c \neq b$. In the case where b and c are distinct, $\Delta\lfloor_S = \Delta'\lfloor_S$, and also $(\Delta',\ \mathsf{arm\_holding}\ b)\lfloor_S = \Delta'\lfloor_S$. Then, what we know and need to show are the same, so the case is satisfied.

In the case where $b = c$, then we know $\Delta\lfloor_S = \Delta'\lfloor_S, \mathsf{on\_table}\ b$.

By inversion, the proof that $\Delta'\lfloor_S, \mathsf{on\_table}\ b \vDash \mathsf{ontable}\ b \vdash (\mathsf{ot}\ b) \oplus (\exists b'.\mathsf{on}\ b\ b') \oplus (\mathsf{ah}\ b)$ can only take the following form (according to the semantic interpretation of satisfiability given in 6.2.2, which essentially reduces to linear sequent provability):

$$\dfrac{\dfrac{}{\mathsf{ontable}\ b \vdash \mathsf{ontable}\ b}\ \mathsf{init}}{\mathsf{ontable}\ b \vdash (\mathsf{ot}\ b) \oplus (\exists b'.\mathsf{on}\ b\ b') \oplus (\mathsf{ah}\ b)}\ \oplus R_1$$

Thus, $\Delta'\lfloor_S$ must be empty.

Now we need to show

$$(\Delta',\ \mathsf{arm\_holding}\ b)\lfloor_S \vDash (\mathsf{ot}\ b) \oplus (\exists b'.\mathsf{on}\ b\ b') \oplus (\mathsf{ah}\ b)$$

Since $\Delta'\ \lfloor_S$ is empty,

$$\begin{aligned}(\Delta',\ \mathsf{arm\_holding}\ b)\lfloor_S &= \Delta'\lfloor_S, \mathsf{arm\_holding}\ b\\ &= \mathsf{arm\_holding}\ b\end{aligned}$$

So we equivalently need to show $\mathsf{arm\_holding}\ b \vDash (\mathsf{ot}\ b) \oplus (\exists b'.\mathsf{on}\ b\ b') \oplus (\mathsf{ah}\ b)$ which holds by derivation from $\oplus R$ rules. □

Invariant BW_2: The rule `pickup_from_table` preserves block top well-formedness

Intutitively, this property holds because the only block affected is the one manipulated by the rule (B), and its bottom's initial $\mathsf{ontable}$ property is simply replaced by an equally satisfactory $\mathsf{arm\_holding}$ property. The proof follows.

By assumption and expansion of BW_1, for all $b : block$, and in particular for the block b specified by the transition, $\Delta\lfloor_{\{\mathsf{ot}\ b,\ \mathsf{on}\ b\ \_,\ \mathsf{ah}\ b\}} \vDash (\mathsf{ot}\ b) \oplus (\exists b'.\mathsf{on}\ b\ b') \oplus (\mathsf{ah}\ b)$. Intuitively, this property holds because the only block affected is the one manipulated by the rule, and its top's initial clear property is simply replaced by an equally satisfactory $\mathsf{arm\_holding}$ property. The proof follows.

Let b be the block index chosen by the transition. Let the restriction set S be $\{\mathsf{cl}\ b, \mathsf{ah}\ b, \mathsf{on}\ \_\ b\}$.

We know by assumption that

$$\Delta\lfloor_S \vDash (\mathsf{cl}\ b) \oplus (\mathsf{ah}\ b) \oplus (\exists b'.\mathsf{on}\ b'\ b)$$

and since $\mathsf{clear}\ b$ satisfies the disjunction, $\Delta'\lfloor_S$ is empty by the same inversion reasoning as previously.

We need to show $(\Delta', \mathsf{ah}\ b)\lfloor_S \vDash (\mathsf{cl}\ b) \oplus (\mathsf{ah}\ b) \oplus (\exists b'.\mathsf{on}\ b'\ b)$.

Since $\Delta'\lfloor_S$ is empty, it suffices to show $\mathsf{ah}\ b \vdash (\mathsf{cl}\ b) \oplus (\mathsf{ah}\ b) \oplus (\exists b'.\mathsf{on}\ b'\ b)$, which is derivable by rules. □

Proof for BW_3 (`pickup_from_table` preserves arm well-formedness): Intuitively, this property holds because the arm's initial arm_free property is replaced by an equally satisfactory arm_holding property. The proof follows.

Let S be the restriction set $\{\mathsf{af}, \mathsf{ah}\ \_\}$.

We know by assumption that

$$\Delta\lfloor_S \vDash (\exists b.\mathsf{ah}\ b) \oplus \mathsf{af}$$

and since arm_free satisfies the disjunction, $\Delta'\lfloor_S$ is empty by the same inversion reasoning as previously.

We need to show $(\Delta', \mathsf{ah}\ )\lfloor_S \vDash (\exists b.\mathsf{ah}\ b) \oplus \mathsf{af}$

Since $\Delta'\lfloor_S$ is empty, it suffices to show $\mathsf{ah}\ b \vdash (\exists b.\mathsf{ah}\ b) \oplus \mathsf{af}$, which is derivable by rules. □

So far we have only proven the invariant for one rule, intentionally chosen as one of the simpler ones in the program. In order to give a representative sample of this method, we now show consider a rule manipulating the *interaction* between two blocks, which we also want to preserve the invariants. Below, we give the intuition for the proof of invariant preservation for the `putdown_on_block` rule. There is one case that differs in format from the cases in the previous proof: in particular, invariant BW_1 for block b', the preservation of the bottom block's bottom well-formedness.

Proposition 6.2.2. *For any Δ', let $\Delta = \Delta'$, $\mathsf{ah}\ b$, $\mathsf{cl}\ b'$. For each $I \in \{BW_1, BW_2, BW_3\}$, if $I(\Delta)$ then $I(\Delta'$, $\mathsf{on}\ b\ b'$, $\mathsf{cl}\ b$, af).*

Proof Sketch: We need to reason about each of the two blocks mentioned in this rule, b and b', and ensure that BW_1 and BW_2 holds for each of them.

Invariant BW_1 (bottom well-formedness) for block b:
This property holds because $\mathsf{ah}\ b$ is replaced by $\mathsf{on}\ b\ b'$.

Invariant BW_2 (top well-formedness) for block b:
This property holds because $\mathsf{ah}\ b$ is replaced by $\mathsf{clear}\ b$.

Invariant BW_1 (bottom well-formedness) for block b':
This property holds because the rule does not manipulate this property, i.e. predicates concerning it are not mentioned to the right nor the left of the rule.

Invariant BW_2 (top well-formedness) for block b':
This property holds because $\mathsf{clear}\ b'$ is replaced by $\mathsf{on}\ b\ b'$.

Invariant BW_3 (arm well-formedness)
This property holds because $\mathsf{ah}\ b$ is replaced by af .

6.2.1 Example: Tower of Hanoi

We continue to illustrate this meta-logical approach by example, this time with a variation on the blocks world domain that is closer to a games application: Tower of Hanoi. This example also involves the persistent, backward-chaining component of the language and allows us to illustrate our treatment of such constructs in program rules.

Define the Tower of Hanoi domain as follows: there is a finite number of posts on which rings can be placed. A robot arm may either be holding a ring or free. A ring R may only be placed on top of another ring R' if R is smaller than R'. To make our encoding more concise, we introduce a general `place` type to represent either an empty post or the top of a ring. The code follows:

```
ring : type.
smaller ring ring : bwd.

place : type.
post : type.
top_of ring : place.
bottom post : place.

on ring place : pred.
clear place : pred.
arm_free : pred.
arm_holding ring : pred.

pickup : clear (top_of R) * on R P * arm_free
      -o arm_holding R * clear P.

putdown_on_ring : arm_holding R * clear (top_of R’) * smaller R R’
      -o arm_free * on R (top_of R’) * clear (top_of R).

putdown_on_post : arm_holding R * clear (bottom P)
      -o arm_free * on R (bottom P) * clear (top_of R).
```

An example instantiation of this domain is:

```
p1 : post.  p2 : post.  p3 : post.
r1 : ring.  r2 : ring.  r3 : ring.

smaller r1 r2.
smaller r1 r3.
smaller r2 r3.

context init =
{clear (bottom p2), clear (bottom p3),
    on r3 (bottom p1), on r2 (top_of r3), on r1 (top_of r2),
    clear (top_of r1), arm_free}.
```

Runnable Ceptre code for this example can be found in Appendix C.3.

The invariant we want to show preserved is that whenever a ring is on another ring, the top ring is smaller. Here is a candidate statement of this invariant in the meta-logic:

$$I_{ToH^*}(\Delta) = \forall r, r'.\Delta\lfloor_{\{\mathsf{on}\ r\ (\mathsf{top\_of}\ r')\}} \vDash ((\mathsf{on}\ r\ (\mathsf{top\_of}\ r'))\otimes!\mathsf{smaller}\ r\ r') \oplus 1$$

This invariant says that any two rings in the program are either in the `on` relation, in which case they are accompanied by a provable `smaller` relation between them, or there is no such relationship between them.[3]

In order to make our proofs go through, however, we will need to strengthen the invariant to include one of the original properties of the blocks world program: in particular, the fact that rings have mutually exclusive predicates referring to what is under them and what is atop them. It suffices to supply just one of these properties, e.g.:

$$\begin{aligned} &I_{ToH}(\Delta) = \\ \forall r.\Delta\lfloor_{\{\mathsf{on}\ r\ \_,\ \mathsf{ah}\ r\}} &\vDash (\exists r'.(\mathsf{on}\ r\ (\mathsf{top\_of}\ r'))\otimes!\mathsf{smaller}\ r\ r') \\ &\oplus (\exists p.(\mathsf{on}\ r\ (\mathsf{bottom}\ p))) \\ &\oplus (\mathsf{arm\_holding}\ r) \end{aligned}$$

The proof that this invariant holds of the program is given rule-by-rule. For rule `pickup`:

Proposition 6.2.3 (Rule preserves Tower of Hanoi invariant)**.** *For all* Δ', r : ring, *and* p : place, *if* $I_{ToH}(\Delta',\ \mathsf{clear}\ (\mathsf{top\_of}\ r),\ \mathsf{on}\ r\ p,\ \mathsf{arm\_free}\ )$, *then* $I_{ToH}(\Delta',\ \mathsf{arm\_holding}\ r,\ \mathsf{clear}\ p)$.

Proof. Let $\Delta = \Delta',\ \mathsf{clear}\ (\mathsf{top\_of}\ r),\ \mathsf{on}\ r\ p,\ \mathsf{arm\_free}$. For most instantiations of the restriction set, the proof will be trivial, since the rule does not manipulate any instances of the restriction set unless the argument is r. In that case, let S be the restriction set $\{\mathsf{on}\ r\ \_, \mathsf{ah}\ r\}$, and let A be the formula

$$\begin{aligned} &(\exists r'.(\mathsf{on}\ r\ (\mathsf{top\_of}\ r'))\otimes!\mathsf{smaller}\ r\ r') \\ &\oplus (\exists p'.(\mathsf{on}\ r\ (\mathsf{bottom}\ p'))) \\ &\oplus (\mathsf{arm\_holding}\ r) \end{aligned}$$

By assumption, $\Delta\lfloor_S \vDash A$. The restriction $\Delta\lfloor_S$ computes to $\Delta'\lfloor_S, \mathsf{on}\ r\ p$, and since $\mathsf{on}\ r\ p$ suffices to prove the disjunction (either the first or second disjunct), $\Delta'\lfloor_S = \cdot$.

Need to show: $(\Delta', \mathsf{ah}\ r, \mathsf{cl}\ p)\lfloor_S \vDash A$. $(\Delta', \mathsf{ah}\ r, \mathsf{cl}\ p)\lfloor_S = \mathsf{ah}\ r$ by reduction. $\mathsf{ah}\ r \vdash A$ by rules (the single assumption satisfies the right disjunct).

□

[3] This invariant may seem awkward compared to a similar implication-flavored statement: $\mathsf{on}\ r\ (\mathsf{top\_of}\ r') \supset !\mathsf{smaller}\ r\ r'$. However, this implication sits at the meta-logical level, and we do not wish to include such a connective in our restricted metalogic for the sake of potential automation. On the other hand, in classical logic we have the equivalence $A \supset B \equiv (\neg A) \vee B$, which gives us a *positive* (in the focusing sense; see Chapter 2) characterization of implication more suitable to automatic search. This invariant statement is essentially a reflection of this equivalence, which is valid in this logic because we consider the presence or absence of an atom in the context to be decidable—i.e. in a context restricted to the set $\{a\}$, the proposition $a^k \oplus 1$ will always hold for some k, which is analogous to the law of excluded middle in classical logic.

For rule putdown_on_ring:[4]

Proposition 6.2.4. *For all* Δ' *and* r : ring, *if* $I_{ToH}(\Delta',\ \mathsf{arm\_holding}\ r, \mathsf{clear}\ (\mathsf{top\_of}\ r'))$ *and* $\Sigma \vdash \mathsf{smaller}\ r\ r'$, *then* $I_{ToH}(\Delta',\ \mathsf{arm\_free}\ ,\ \mathsf{on}\ r\ (\mathsf{top\_of}\ r'),\ \mathsf{clear}\ (\mathsf{top\_of}\ r))$.

Proof. Let $\Delta = \Delta',\ \mathsf{arm\_holding}\ r, \mathsf{clear}\ (\mathsf{top\_of}\ r')$. The proof is trivial for all instantiations of the invariant except in cases where the restriction set mentions r, so let us consider the instantiation of the invariant at r for which the proof is nontrivial. Let S be the restriction set $\{\mathsf{on}\ r\ \_,\ \mathsf{ah}\ r\}$. Let A be the formula

$$\begin{aligned}&(\exists r'.(\mathsf{on}\ r\ (\mathsf{top\_of}\ r')) \otimes !\mathsf{smaller}\ r\ r')\\ &\oplus (\exists p'.(\mathsf{on}\ r\ (\mathsf{bottom}\ p')))\\ &\oplus (\mathsf{arm\_holding}\ r)\end{aligned}$$

We know $\Delta\lfloor_S \vDash A$, and in particular the premise $\mathsf{ah}\ r$ satisfies the third disjunct, so $\Delta'\lfloor_S$ is constrained to be empty. We need to show

$$(\Delta', \mathsf{af}\ , \mathsf{on}\ r\ (\mathsf{top\_of}\ r'),\ \mathsf{clear}\ (\mathsf{top\_of}\ r))\ \lfloor_S \vDash A$$

The context restriction computes to $\mathsf{on}\ r\ (\mathsf{top\_of}\ r')$ which together with the assumption $!\mathsf{smaller}\ r\ r'$ satifies the first disjunct. □

For rule putdown_on_post:

Proposition 6.2.5. *For all* Δ', r : ring, *and* p : post, *if* $I_{ToH}(\Delta',\ \mathsf{clear}\ (\mathsf{bottom}\ p),\ \mathsf{arm\_holding}\ r)$, *then* $I_{ToH}(\Delta',\ \mathsf{arm\_free}\ ,\ \mathsf{on}\ r\ (\mathsf{bottom}\ p),\ \mathsf{clear}\ (\mathsf{top\_of}\ r))$.

Proof. Let $\Delta = \Delta',\ \mathsf{arm\_holding}\ r,\ \mathsf{clear}\ (\mathsf{bottom}\ p)$. The invariant holds of Δ instantiated at r because of the $\mathsf{arm\_holding}\ r$ premise, which is straightforwardly replaced by the $\mathsf{on}\ r\ (\mathsf{bottom}\ p)$ consequent. This proof follows the same pattern we have now seen several times. □

6.2.2 Potential for Automation

In order to discuss potential algorithms for deciding a fragment of this meta-logic, we need to define that fragment. To do so, we constrain the grammar of the meta-logic to formulas ϕ of the following form:

$$\begin{array}{rcl}\phi & ::= & \lambda\delta.\gamma\\ \gamma & ::= & \forall x{:}\tau.\gamma \mid \Delta\lfloor_S \vDash \psi\\ \psi & ::= & a \mid \psi \otimes \psi \mid \psi \oplus \psi \mid \exists x{:}\tau.\psi \mid 1\\ S & ::= & \cdot \mid \mathsf{pat}, S\\ \mathsf{pat} & ::= & a\ \mathsf{tpat}_1 \ldots \mathsf{tpat}_n\\ \mathsf{tpat} & ::= & \_ \mid t \mid c\ \mathsf{tpat}_1 \ldots \mathsf{tpat}_n\end{array}$$

[4]Note: since this rule involves a persistent premise, we must explicitly mention the signature Σ supplying the backward-chaining rules defining persistent propositions, which we had previously been leaving silent. This treatment emerges from what it means for a transition to apply; see Chapter 3 for details.

The grammar for terms t, term constants c, and atomic predicates a is borrowed from Ceptre's grammar, given in Chapter 4.

Then we give an interpretation of each component when a formula ϕ is applied to a context Δ (with respect to some type header Σ):

$$\begin{array}{rcl}
[\![\forall x{:}\tau.\gamma]\!] & = & \Sigma \vdash t : \tau \text{ implies } \Delta \vDash [t/x]\gamma \\
[\![\Delta \vDash a]\!] & = & \Delta = \{a\} \\
[\![\Delta \vDash \psi_1 \otimes \psi_2]\!] & = & \Delta = \Delta_1, \Delta_2 \text{ and } \Delta_1 \vDash \psi_1 \text{ and } \Delta_2 \vDash \psi_2 \\
[\![\Delta \vDash \psi_1 \oplus \psi_2]\!] & = & \Delta \vDash \psi_1 \text{ or } \Delta \vDash \psi_2 \\
[\![\Delta \vDash \exists x{:}\tau.\psi]\!] & = & \text{there exists } t \text{ s.t. } \Sigma \vdash t : \tau \text{ and } \Delta \vDash [t/x]\psi \\
[\![\Delta \vDash 1]\!] & = & \Delta = \cdot
\end{array}$$

Pattern restriction $\Delta{\downharpoonleft_S}$ computes another context Δ', defined approximately as the intersection of Δ and S where any wildcard patterns _ in S are considered equivalent to any term. We write that an atom p *matches* a pattern pat with the notation $p \triangleright \mathsf{pat}$, which is defined next.

Pattern restriction:

$$\begin{array}{rcl}
\cdot{\downharpoonleft_S} & = & \cdot \\
(\Delta, p){\downharpoonleft_S} & = & \Delta{\downharpoonleft_S}, p \quad \text{if } p \in^* S \\
 & = & \Delta{\downharpoonleft_S} \quad \text{otherwise} \\
p \in^* S & \text{iff} & \exists \mathsf{pat} \in S. p \triangleright \mathsf{pat}
\end{array}$$

Pattern matching:

$$\begin{array}{rcl}
t & \triangleright & t \\
t & \triangleright & \_ \\
c\, t_1 \ldots t_n & \triangleright & c\, \mathsf{tpat}_1 \ldots \mathsf{tpat}_n \text{ iff } t_1 \triangleright \mathsf{tpat}_1 \text{ and } \ldots \text{ and } t_n \triangleright \mathsf{tpat}_n \\
a\, t_1 \ldots t_n & \triangleright & a\, \mathsf{tpat}_1 \ldots \mathsf{tpat}_n \text{ iff } t_1 \triangleright \mathsf{tpat}_1 \text{ and } \ldots \text{ and } t_n \triangleright \mathsf{tpat}_n
\end{array}$$

Checking that a given context matches a formula in this fragment of the logic for a given context has a straightforward correspondence to the positive fragment of first-order linear logic, which is a subset of MALL and thus decidable. Including the exponential operator ! in the limited fashion demonstrated by the Tower of Hanoi example, i.e. to express constraints on indices whose provability can be fully extricated from the linear components, does not seem to affect this decidability in any obvious way, but we currently leave its decidability as conjecture.

For invariant checking, i.e. checking that a given transition $\Delta \to \Delta'$ enabled by the program under scrutiny preserves a property given in this language, we would need to codify the proof technique embodied by the manual proofs given above in such a way that were guaranteed to terminate, including inversion of assumed derivations.

Here is a sketch of the algorithm, generalized from the previous examples:

To check an invariant $\phi = \lambda\delta.\forall \vec{x{:}\tau}.\delta{\downharpoonleft_S} \vDash \psi$ of a transition $\Delta \to \Delta'$:

1. Expand $\phi(\Delta)$. Generate fresh symbols for each $\forall$-quantified variable x_i in $\phi(\Delta')$.
2. Enumerate cases for terms in Δ being equal or not equal to those symbols.
3. For each case, compute the restriction set S and determine possible inversions based on the rules for decomposing the inner formula ψ (analogous to linear sequent calculus right rules). These introduce equality constraints on the restriction set of the input context Δ and on the part it shares with Δ'.
4. Using those constraints and, again, decomposition rules for the invariant formula, determine satisfiability of $\Delta' \vDash \psi$.

This process does not straightforwardly correspond to a known-decidable procedure, but since the satisfiability semantics obey a subformula property in the same sense as their corresponding linear logic rules, we expect that inversion is a computable process. We also have not yet encountered examples in this fragment that require inductive lemmas. Thus, we conjecture that checking preservation of invariants described in this meta-logic is decidable. It remains to prove that this is the case, as well as to prove soundness and completeness with the logical derivability-based notion of invariant preservation. We leave further formalization and proof to future work.

6.2.3 Limitation: Recursive Predicates

Working in a restricted meta-logic to specify program invariants has some drawbacks. For instance, some specifications of well-formed states are simply inaccessible. Consider an encoding of linked lists in linear logic wherein memory areas are encoded as abstract *destinations* (terminology borrowed from the idea of "destination-passing style" [CPWW03]) used as indices to a predicate describing data at a location as well as a reference to the next location:

```
data : type.
location : type.
node location data location : pred.
```

The list `[a,b,c]` would be encoded as the domain instance:

```
a : data. b : data. c : data.
l1 : location. l2 : location. l3 : location.
end : location.
context list_abc = {node l1 a l2, node l2 b l3, node l3 c end}.
```

We can write programs over this data that, for instance, delete a node from the list (at location `L'`):

```
delete : at L V L' * at L' _ L'' -o at L V L''.
```

A common verification problem for reasoning about programs at a memory layout level is *shape analysis* [SRW02, DOY06]: can we reason that a given segment of memory has a particular shape (for instance binary tree or linked list) and that a program that manipulates it, possibly dynamically allocating memory, preserves that shape? Previously, linear logic has been investigated as a candidate for expressing these program constraints in terms more abstract than memory locations [JW06], and it seems that such analyses should also extend to linear logic programs themselves.

In particular, for this example, we might like to state that the linear, non-circular, linked list structure of our memory layout encoding is preserved by the deletion rule. But such a property cannot be given by local characterization of a single index; it must refer to the structure of how indices are shared between predicates in the context. We also cannot give it as a persistent property defined outside the state evolution rules, since it refers to facts (like the way nodes are linked) that fundamentally change during execution.

Instead, such a property is more naturally given recursively, by saying either the list is empty, or contains a node pointing to another well-formed linked list. One candidate approach for stating this kind of invariant is to extend the meta-logic with a notion of recursive predicate. Such extensions to first-order linear logic have been explored and proven sound [Bae08]. However, this extension may complicate decidability of the approach, and does not offer clear means for automation. We describe a related approach more fully as part of a different technique in Section 6.3.

6.3 Consumptive Invariants

Aside from automation concerns, we also hope to extend the expressiveness of the meta-logic by allowing the description of recursive invariants, such as those described in the shape analysis literature. We describe a technique that can *dynamically* check such invariants next.

This technique hinges on the observation that descriptors of context properties resemble grammars that themselves can be expressed in linear logic. And we can write recursive linear logic programs to express those properties via backward-chaining.

6.3.1 Backward-Chaining Linear Logic Programs

Linear logic programs may be given a backward-chaining interpretation so long as each clause only has a single, atomic consequent, i.e. takes the form $A \multimap p$. In concrete syntax, instead of `p1 * ... * pn -o p` for such a rule we will write the backwards, curried form `p o- p1 o- ... pn`, mimicking the syntax for persistent backward chaining programs.

We refer to p as the *head* of such a clause, and proof search may be carried out on a particular atom by matching it against the heads of all logic program clauses, then matching their premises as new goals, backtracking when search along a particular branch fails, just like persistent backward chaining. But linear rules still have a resource-based meaning, i.e. for a rule `p o- p1 o- p2` to succeed at establishing `p` in a context Δ means that Δ must be partitionable into Δ_1, Δ_2 such that `p1` consumes all of Δ_1 and `p2` consumes all of Δ_2.

6.3.2 Example: Linked List Shape Analysis

Consider the linked list example given in Section 6.2.3. We can write a simple recursive predicate defining a well-formed linked list *segment* with two arguments, s for the beginning location of the list and e for the ending location. The linked list segment is well formed if either $s \doteq e$ (the segment is empty) or there is some node at s with "next" field l such that l and e define a well-formed segment.

This definition can be represented by the following backward-chaining linear logic program:

```
ll location location : bwd.
ll X X.
ll S E o- at S V L
       o- ll L E.
```

Simmons [Sim12] refers to this kind of specification as a *consumptive* signature, because to use it as a verification tool, we would supply a context Δ and attempt to prove $\Delta \vdash \exists s, e.\mathsf{ll}\ s\ e$, which works operationally by using the rules defining `ll` to *consume* elements of Δ until none are left.

6.3.3 Potential for Automation

By combining the restriction mechanism from the meta-linear logic approach with predicates defined recursively in linear logic, we can straightforwardly extract a *dynamic* checking algorithm: after every rule application, check that the resultant context Δ (or whatever restriction of it) satisfies the formula by simply running linear logic proof search. We have not determined to what extent such invariants can be checked of a program statically, nor to what extent the dynamic process described can be optimized not to re-compute full provability between every transition in the program.

6.3.4 Limitation: Apartness Constraints

Apart from a lack of a known static decision procedure for preservation, consumptive invariants have the following limitation: they cannot enforce that a given predicate holds only for distinct term indices. For instance, consider this candidate representation of the blocks world well-formedness property.

```
wf_bw
  o- wf_arm
  o- wf_stacks.

wf_arm o- arm_free.
wf_arm o- arm_holding X.

wf_stacks o- 1.

wf_stacks o- on_table B
```

```
          o- wf_stack B
          o- wf_stacks.

wf_stack B o- clear B.

wf_stack B o- on B' B
           o- wf_stack B'.
```

This specification does not suffice to rule out ill-formed blocks world configurations as specified in Section 6.2, because it does not enforce that a given block B is not, say, simultaneously held by the arm and on the table (or appearing in multiple stacks). The term indices of the rules may be instantiated with any substitutions, including ones that unify them with otherwise-existing terms.

6.4 Generative Invariants

A *generative* signature, in contrast to a consumptive signature, is one that describes a context property through rules that *generate* all permissible contexts, again using the analogy between logic programs and grammars. Generative signatures and their use for specifying logic program invariants were also first described in [Sim12].

Generative signatures are collections of forward-chaining rules together with a *seed* context Δ_0, usually containing a distinguished `gen` atom which is expanded by the signature. By convention, we will assume that all seed contexts take this form such that the signature itself suffices to specify the property.

At first glance, generative signatures look like consumptive signatures "with the arrows turned around:" instead of distinct `wf` (well-formedness) predicates for each portion of the context, we have distinct *generators*, analogous to nonterminals in a grammar, for each portion of the context. A complete well-formed context Δ, then, is one for which there is a transition sequence $\Delta_0 \to \Delta$ along rules given in the generative signature Σ_{gen}, where Δ contains no nonterminals. An extremely simple example is given below:

```
gen -o {a * gen}.
gen -o {1}.
```

This signature, equipped with the seed context $\{\texttt{gen}\}$, describes contexts containing zero or more instances of `a`. Formally, the set of contexts that a signature Σ and a seed Δ_0 describes is the set of reachable contexts from Δ_0 following rules in Σ. We call such a signature and seed pair (Σ, Δ_0) a *generative invariant* of a program signature Σ' if all rules in Σ' maintain the program state's membership in the set generated by Σ seeded with Δ_0. We make this notion more precise later.

In addition to universally-quantified indices (standard logic variables), generative invariants may also include *existentially generated* variables via rules of the form `A -o exists x.B`. The existential quantifier has completely standard treatment from a forward-chaining proof search perspective; see the CLF paper [WCPW03] for a formal treatment.

6.4.1 Example: Generative Signature for Blocks World

Below we give a generative signature characterizing the blocks world domain, effectively by "turning the arrows around" in the consumptive signature. Let $\Sigma_{\mathsf{bwgen}} =$

```
gen/bw : gen -o gen_arm * gen_stacks.
gen_arm/af : gen_arm -o arm_free.
gen_arm/ah : gen_arm -o exists b. arm_holding b.
gen_stacks/done : gen_stacks -o 1.
gen_stacks/more :
  gen_stacks -o exists b. on_table b * gen_stack b * gen_stacks.
gen_stack/clear : gen_stack B -o clear B.
gen_stack/more  : gen_stack B -o exists b. on b B' * gen_stack b.
```

This signature says: in order to build a blocks world, build an arm and a set of block stacks. The arm can either be free or holding a block. A set of block stacks can be empty, or it can contain a block stack starting with a new block index, where that block is on the table, along with the rest of the set of block stacks. A block stack is indexed by its top block, and that block can either be clear, or have another block on top of it that becomes the new block stack index.

6.4.2 Generative Property Preservation

We now define what it means for program transitions to preserve generative properties.

Definition: A transition $\Delta, A \to \Delta, B$ preserves a generative property $\langle \Sigma_{gen}, \Delta_0 \rangle$ iff for all Δ, whenever $\Delta_0 \leadsto_{\Sigma_{gen}} \Delta, A^*$, it is also the case that $\Delta_0 \leadsto_{\Sigma_{gen}} \Delta, B^*$.

In particular, for the blocks world example, we can show how to reason that the `pickup_from_table` rule, enabling the transition $\Delta, \mathsf{ot}\ b,\ \mathsf{af},\ \mathsf{cl}\ b \to \Delta, \mathsf{ah}\ b$, preserves the invariant by showing that:

Proposition 6.4.1. *If*

$$\{\mathsf{gen}\} \to_{\Sigma_{\mathsf{bwgen}}} \Delta, \mathsf{on\_table}\ b,\ \mathsf{arm\_free},\ \mathsf{clear}\ b$$

then

$$\{\mathsf{gen}\} \to_{\Sigma_{\mathsf{bwgen}}} \Delta, \mathsf{arm\_holding}\ b$$

Proof Sketch: The signature Σ_{bwgen} can generate $\Delta, \mathsf{on\_table}\ b, \mathsf{arm\_free}, \mathsf{clear}\ b$ in only the following way, up to concurrent equality:

1. Apply rule `gen/bw` to get context `{gen_arm, gen_stacks}`.
2. Apply rule `gen_arm/af` to get context `{arm_free, gen_stacks}`.
3. Apply some unknown rule sequence to `gen_stacks` to yield an unknown context `D1` along with another copy of `gen_stacks`.
4. Apply `gen_stacks/more` to create b along with its `on_table` property. Context is now `{arm_free, D1, on_table b, gen_stack b, gen_stacks}`.
5. Apply `gen_stack/clear` to `gen_stack b` to get `clear B`.

6. Apply some unknown rule sequence to `gen_stacks` to get a side effect nonterminal context `D2`, and eventually eliminate all nonterminals.
   The context is now `{arm_free, D1, on_table b, clear b, D2}`.
   The original frame context Δ must equal `D1, D2`.

Need to show: $\{\mathsf{gen}\} \rightarrow_{\Sigma_{\mathsf{bwgen}}} \Delta, \mathsf{arm\_holding}\ b$.
Given the proof trace above, we can modify it selectively to produce the context we need. We modify step 2 to replace `gen_arm/af` with an application of `gen_arm/ah`, which generates a block b that is indistinguishable from the specific b in question up to alpha renaming, and the predicate `arm_holding b`. We modify steps 4 and 5 to be the null transition, simply linking together the trace that generated `D1` to the trace that uses `gen_stacks` to generate `D2`, eliminating the terminals that were produced as a side-effect. □

To make the above proof sketch formal, we need to give the input trace in terms of ϵ notation (a sequence of let-bindings **let** $\langle x_1 \ldots x_n \rangle = r\ R$ witnessing the transformation $\{\mathsf{gen}\} \rightarrow_{\Sigma_{\mathsf{bwgen}}} \Delta, \mathsf{on\_table}\ b,\ \mathsf{arm\_free},\ \mathsf{clear}\ b$) and prove that it really is the only possibility, up to concurrent equality. Simmons [Sim12] carries out such proofs by proving *permutability* lemmas, which say that in any trace, we can permute the generation of the terminals we care about to the end, then reason inductively on the remaining trace prefix. The tediousness and surfeit of cases in these proofs suggest that formal argument over traces is the wrong level of abstraction for these proofs—in the case of generative well-formedness, we do not care about the *order* in which things are generated by the signature but rather only about reachability between intermediate states of the generative program. An attempt at visualizing the structure of a generative trace is given in Figure 6.1.

The ability to draw the trace this way makes several assumptions that we would need to prove about a given generative signature, such as non-interference between `gen_arm` and `gen_stacks`, and for that matter the fact that terminals cannot interfere with the trace structure. All of these assumptions suggest constraining the grammar of generative signatures themselves in such a way that their traces always have this tree-like structure, which we will do when we formalize them in an attempt to automate proof search.

6.4.3 Potential for Automation

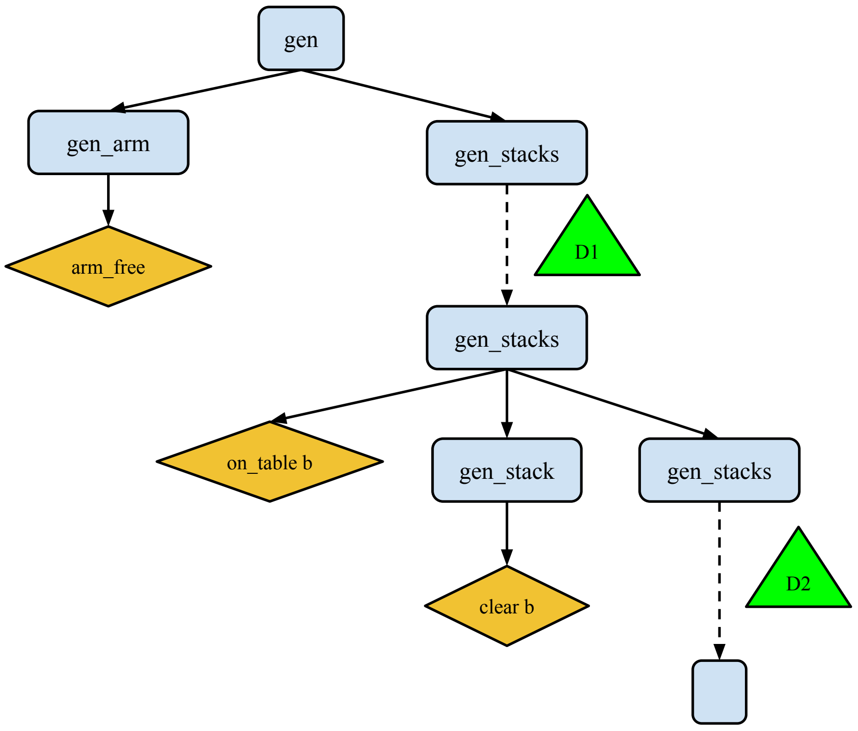
We have not yet devised a general algorithm for checking preservation of generative invariants, and we suggest that without a better layer of abstraction for reasoning about generative traces, such a task is intractable: the by-hand proofs frequently involve inductively-proven lemmas and careful scrutiny of the particular generative signature at hand. However, generative signatures so far yield the most definitive result of the three approaches: if we limit programs and generative signatures to *atomic propositions*, invariant preservation is decidable. We give a proof of this fact in Section 6.5.

Figure 6.1: Generative trace structure

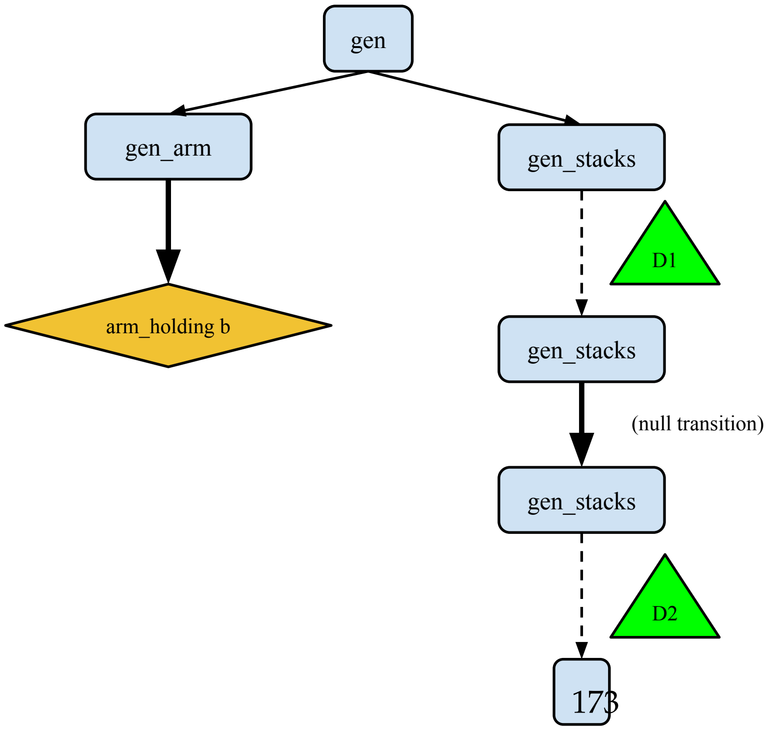
These diagrams depics the generative trace structure for blocks world well-formedness of precondition and postcondition of the rule

```
    pickup_from_table : on_table B * clear B * arm_free -o arm_holding B.
```

Precondition trace:



Postcondition trace:



6.4.4 Limitations

Generative signatures resolve the major limitation of consumptive signatures, the expression of apartness constraints, through the existential quantifier, which always generates a fresh term distinct from any others previously in existence. However, this approach is not without its limitations.

For one thing, we suspect that generative signatures are a substantially less intuitive way to write specifications than, for instance, the first-order formulas of meta-linear logic discussed in Section 6.2. While meta-linear logic is a fairly direct way of translating a constraint on a program into a logical formula, generative invariants require the programmer to *operationalize* their understanding of the constraint by specifying how to generate contexts that satisfy it.

Second, there is another large class of constraints we cannot describe with generative invariants: consider describing *well-formed graphs* where edges between nodes are predicates in the program, but we do not want to create self-loops or multi-edges. However, a new node created may refer back to an old node. A candidate signature is:

```
gen/nodes : gen -o exists n:node. gen_edges n * gen.
gen_edges/edge : gen_edges N * gen_edges N'
    -o edge N N' * gen_edges N * gen_edges N'.
gen_edges/done : gen_edges N -o 1.
```

This specification has two problems: first of all, it has a rule with two nonterminals on the left, which even in the propositional case falls outside the known-decidable fragment. Second, it cannot avoid generating self-edges or multi-edges, since nothing stops `N` and `N'` from having the same node substituted for them. The existential "fresh generation" property does not help us here, because it cannot promise to keep indices apart in a non-local way, i.e. in a rule other than the one where the existential appears.

In general, there does not appear to be a way to specify formation properties in which indices can be shared arbitrarily between predicates while also enforcing certain apartness constraints. Even with meta-linear logic, we would need to introduce new apartness primitives on terms in order to handle such a specification.

6.5 Decidability of Invariant Checking for the Propositional Fragment

In this section we prove that checking whether a (propositional) generative invariant holds of a (propositional) linear logic program is decidable. We do so by modeling propositional generative signatures and programs as *vector addition systems* (first introduced by that name by Karp and Miller [KM69], although the equivalent Petri nets were devised earlier [Pet66]) in a way that they can be treated using known techniques. Specifically, we show how invariant preservation can be modeled in Presburger Arithmetic, the first-order theory of natural numbers with inequality and addition, which (despite impractical computational complexity) happens to be decidable [Sta84].

6.5.1 The Propositional Fragment

Propositional Ceptre programs are those which do not have any logic variables (Π-bound variables), and thus the available transitions at any point in the program does not depend on the term language available. Formally, it is simply a restriction of the grammar of rules to

$$\begin{array}{rcl} A & ::= & S \multimap S \\ S & ::= & 1 \mid a \mid S \otimes S \end{array}$$

Instead of refering to these programs as propositional *Ceptre* programs, we will refer to them as propositional *Horn* programs by analogy with Horn clauses in standard logic programming (i.e. ones with no left-nested implication) [Kow74], which we believe is consistent with prior definitions of the Horn fragment of linear logic [Kan92, Kan95].

Below is an example of a propositional Ceptre program modeling coin exchange—`r1` exchances a dime for two nickels, and `r2` exchanges a quarter for two dimes and a nickel.

```
r1 : d -o n * n.
r2 : q -o d * d * n.
```

Even with such a limited specification, we can already ask all of the same questions we asked in the beginning:

- **What is the set of well-formed states?** In this case, it is simply any number of quarters, dimes, and nickels.
- **Which states do we take to be well-formed *initial* states?** For instance, we might specify that we start with entirely quarters, or we might allow any configuration of coins.
- **What is the set of possible states at *quiescence* (termination) of the program?** This depends on the set of initial states. If we take the initial states to be those with only quarters, then we expect to have a multiple of five nickels at the end and no other coins. If we allow any set of coins initially, then we will have an arbitrary number of nickels (and no other coins) at the end.

Indeed, the propositional fragment corresponds with Petri nets [Mur89] and has been used for quite sophisticated specifications in the games and interactive storytelling literature as well as for modeling concurrent and distributed programs. For instance, Dang et al. [DHCS11] specify a slapstick interactive storytelling scenario in which the player character is attending a party and may take various actions that have cascading effects, such as lighting a woman's cigarette:

```
light_cig : bored_woman * woman_has_cig * cig_not_lit
            -o woman_has_cig * cig_lit.
```

...which can lead to objects and animals catching fire. Because the scenes are all entirely hand-authored and specific to certain characters and objects, the whole interactive space can be described with atomic propositions.

Furthermore, first-order logic programs may be expanded to propositional ones through the process of *grounding*, if their term domains are finite. For instance, if we have the type header

```
char : type.
loc  : type.
at char loc : pred.

alice : char.
bob   : char.

den   : loc.
foyer : loc.
```

Then the rule `rule : at C L -o 1` may be expanded into the following four rules:

```
at alice den   -o 1.
at alice foyer -o 1.
at bob   den   -o 1.
at bob   foyer -o 1.
```

Such a process is typically not considered computationally practical, since the number of rules needed to represent a single rule grows combinatorially with the number of logic variables it has, and checking Presburger formulas is doubly-exponential in complexity [FR98]. However, since our only aim with this proof is to establish decidability (rather than a tractable algorithm), we note that our proof extends to these programs as well.

Thus we consider decidability of invariant checking for propositional programs to be a valuable and novel contribution to the space of validating interactive media.

6.5.2 A Tricky Example

To see why the problem of invariant preservation is a challenge, consider the following generative signature Σ_{abc}.

```
g1 : gen -o {a * cs}.
g2 : gen -o {b * c * c}.
g3 : cs -o {c * cs}.
g4 : cs -o {1}.
```

Note that the class of contexts (Σ_{abc}, `gen`) describes is one where an `a` can appear alongside arbitrarily many `c`s, but a `b` requires *exactly two* `c`s alongside it.

Here are two rules that should pass (preserve the invariant):

1. `a -o {a * c}.`
2. `a * c -o {a}.`

That is, if there is an `a` in the context, we should be able to add or subtract arbitrarily many `c`s.

On the other hand, here are two rules that should fail (do not preserve the invariant):

1. `1 -o {c}.`

2. `c -o {1}`.

That is, we may not arbitrarily remove or generate a `c`. The second generative rule `g2` permits a context with *exactly* one `b` and two `c`s. Rules are *context-sensitive* with respect to generative invariants; they cannot be reasoned about in isolation.

Thus, our algorithm needs to be able to account for preservation not on a purely local, or context-free, basis, but rather in a sense that takes into account *all the possible ways* an atom on the left-hand side of the rule could be generated—which may impose some constraints on the context—and conclude that, in all of those scenarios, including their constraints, the right-hand side of the rule could have been generated in its place.

6.5.3 Vector Addition Systems

In order to draw on prior work in algorithms for validating these systems, we need to understand them in terms of the formalisms chosen by that prior work. In particular, a number of useful properties have been shown of *vector addition systems*, which we employ to obtain decidability.

A *vector addition system* (VAS) V is an initial configuration $v_0 \in \mathbb{N}^n$ for a fixed dimension n, along with a set T of *transitions* describing how a configuration v may evolve. Transitions can be described as vectors $t \in \mathbb{Z}^n$ along with side conditions of the form $k \in \mathbb{N}^n$, which imposes the inequality constraint $v \geq k$. If the conditions hold of a configuration v (i.e. $v \geq k$) *and* $v + t \geq 0$, the transition t is said to be *applicable* or *fireable*, and the configuration may evolve to $v + t$.

For a transition $(t, k) \in T$ that is applicable in v, we notate a transition relation $\rightsquigarrow$, i.e. $v \rightsquigarrow v+t$, which we use in addition to its transitive closure $\rightsquigarrow^*$. The set of configurations $\{x \mid v \rightsquigarrow^* x\}$ is called the *reachability set* of v, and can be extended to sets of starting configurations as well. VASes are equivalent to Petri nets (see e.g. [Ler10, Kan95]), and for both systems, the central problem is computing reachability sets: given a set of starting configurations, what are all of the possible states the system might wind up in? This problem is known to be decidable [Pet77, ST77] (however, we do not make direct use of this fact for our proof).

The correspondence also extends to provability of a sequent in the propositional Horn fragment of linear logic. Simply put, a VAS transition rule of the form (t, k) can be interpreted as a forward-chaining linear logic rule

$$a_1^{c_1} \otimes \ldots \otimes a_n^{c_n} \multimap a_1^{c_1+t_1} \otimes \ldots \otimes a_n^{c_n+t_n}$$

Where $c_i = \mathsf{max}(k_i, -t_i)$ and a^c means $a \otimes \ldots \otimes a$ with c repetitions.

In the other direction, a linear logic program with rules of the form

$$a_1^{c_1} \otimes \ldots \otimes a_n^{c_n} \multimap b_1^{d_1} \otimes \ldots \otimes b_m^{d_m}$$

(with like atoms grouped together without loss of generality) can be interpreted as a VAS by the following process:

1. Give an arbitrary canonical ordering for all atoms in the program to form the space of possible vectors. Thus each atom a corresponds with an index i_a of the vector.

2. For each rule in the above form, create a transition (t, k) where $t_i = c_i - d_i$ and $k_i = c_i$.

By way of example, all of the rules in Σ_{abc} may be translated to the following vector addition rules over `<gen, cs, a, b, c>`:

```
g1 : <-1,  1, 1, 0, 0>   ; <1, 0, 0, 0, 0>
g2 : <-1,  0, 0, 1, 2>   ; <1, 0, 0, 0, 0>
g3 : < 0,  0, 0, 0, 1>   ; <0, 1, 0, 0, 0>
g4 : < 0, -1, 0, 0, 0>   ; <0, 1, 0, 0, 0>
```

The initial configuration $\{\mathsf{gen}\}$ is represented by the vector $\langle 1, 0, 0, 0, 0\rangle$.

As mentioned previously, the decidability of this problem does not help us directly with deciding if a given program rule preserves a generative invariant. We can, however, recast the preservation problem as a VAS problem: if we want to check whether a rule $r : A \multimap B$ satisfies a generative invariant I, reformulate I as a vector addition system $V_I = (i_0, T_I)$. The rule r will correspond to some transition (t, k) (in a different vector addition system, whose full definition is not needed).

Definition: A transition (t, k) *preserves the invariant* given by VAS V_I with starting configuration i_0 iff for all configurations $v \geq k$

$$i_0 \rightsquigarrow^*_{V_I} v$$

implies

$$i_0 \rightsquigarrow^*_{V_G} v + t$$

The vector v in this definition represents an arbitrary frame context Δ, A in which the rule fires. [5]

We can make a few more definitions to simplify this characterization:

Definition: The *reachability set* of a vector v under a particular VAS V is the set of all vectors v' such that the transitive closure of all transitions in V can take v to v':

$$\mathsf{reach}_V(v) = \{v' \mid v \rightsquigarrow^*_V v'\}$$

Definition: The *postset* of a given vector set S along a transition (t, k), denoted $\mathsf{post}_{(t,k)}(S)$, is the set of all vectors v' such that there exists a $v \in S$ where (t, k) is fireable on v and $v' = v + t$.

Now we can reword the above criterion for preservation as simply

$$\mathsf{post}_{(tk_r)}(S) \subseteq S$$

where $S = \mathsf{reach}_{V_G}(c_0)$.

[5] Note that, in general, the arity of vectors for a generative invariant will be higher than the arity for the VAS corresponding to the program we want to check—invariants contain nonterminals that do not appear in program configurations. So to make this definition more precise, the transition $i_0 \rightsquigarrow v$ should really be $i_0 \rightsquigarrow v'$ where $v'_i = v_i$ when i is within bounds for v and $v'_i = 0$ everywhere else.

In general, computing the reachability set does not help us answer this question. However, some (not all) VASes can be expressed as *Presburger Arithmetic formulas*, a subset of first-order logic propositions that may refer to multiplicaiton and addition of natural numbers (but not exponentiation). Presburger Arithmetic is known to be decidable, as shown by Presburger in 1929 (simultaneously with his presentation of the system). And if the reachability set S of a vector addition system V_I representing an invariant corresponds to a Presburger formula, then the question of whether a given rule r preserves it ($\mathsf{post}_{(r)}(S) \subseteq S$) can also be expressed as a Presburger formula. Thus, what remains is to show that the particular subset of VASes that correspond with generative invariants are *all* expressible as Presburger formulas.

6.5.4 Presburger Vector Addition Systems

In general, VASes are not expressible as Presburger Arithmetic formulas. An explanation of this fact and a characterization of the Presburger fragment is given in [Ler13]. Briefly speaking, one can encode exponentiation as a VAS, which is not within the bounds of Presburger Arithmetic. In this section, we recapitulate Leroux's characterization of Presburger Vector Addition Systems in terms of how we use them for generative invariants.

In order to show that generative invariant preservation is decidable, we need to isolate a fragment of the VAS language that is both suitable to use for our generative invariants of interest *and* expressible in Presburger Arithmetic.

Leroux shows that the Presburger-expressible VASes are exactly those which have an equivalent "flat" representation. A *flat* VAS is one that can be characterized as a finite sequence of arbitrarily-repeated finite transition sequences. More formally, consider a *word* w to be a sequence of transitions $t_1 \ldots t_k$ such that t_{i+1} may always follow t_i. The Kleene closure operator on words w^* means "w repeated 0 or more times, so long as it applies." For instance, the closure t^* of a transition t corresponding to the linear logic rule $a \multimap b$ means that t may be applied as many times as there are copies of a in the configuration it starts with.

A flat language, then is one that can be written as

$$(w_1)^* \ldots (w_n)^*$$

for some finite set of words $w_1 \ldots w_n$.

A *flatable* VAS is one with the same reachability set as a flat VAS. One known class of flatable VASs are so-called Basic Parallel Processes, or BPPs [Esp97], which are those corresponding to sets of Horn linear logic rules with a single premise (or Petri nets where every transition has a single input). Although this class may sound limiting, every generative invariant we have studied so far meets this criterion. [6]

[6]Generative invariants for specifying the operational semantics of programming languages [Sim12] occasionally include a second premise, but always a persistent one (premise of the form $!A$). This kind of rule may still be expressible as a BPP so long as the persistent premise can be represented as a side condition, but we currently exclude these examples from our proofs.

There exist published, proven terminating, and implemented methods to decompose VASs such as BPPs into flat languages [Fri00, FO97]. These techniques involve iteratively disentangling cyclic rule dependencies.

As an alternative to automatically finding an equivalent flat language, we can simply stipulate that a given generative invariant must be flat. We note that prior to this investigation, there was no rigorous characterization of what differentiates a *generative signature* from any other linear logic program, and a constraint like this one serves as a candidate for characterizing the space of suitable signatures.

6.5.5 Flat(able) Generative Invariants

Here is a generative invariant that is *not* flat, corresponding loosely to a propositional erasure of a well-formedness specification for blocks world.

```
g1 : gen -o {t * gen'}.  %% t ~= "on table"; construct a new stack
g2 : gen' -o {b * gen'}. %% b ~= "on"; add a block to a stack
g3 : gen' -o {c * gen}.  %% c ~= "clear"; finish a stack and return
g4 : gen -o {f}.         %% f ~= "arm free"
g5 : gen -o {h}.         %% h ~= "arm holding a block"
```

If we try to characterize the allowable traces generated by this program as a grammar, we would write something like $(g_1 g_2^* g_3)^* g_4^* g_5^*$, meaning basically: generate an arbitrary number of stacks of blocks by first generating a block on the table, then generating an arbitrary number of blocks atop it, then designating the last one as clear; finally, designate the arm as either free or holding a block. [7]

This specification is not flat because there is "nested" looping of the rule g2 within the wider loop formed by g1 and g3. This introduces the constraint that a t and c both *must* be generated in order for there to be more than zero bs. It also means that, if there are no nonterminals gen and gen', the number of ts and cs must be the same.

However, the set of contexts generated by this signature is equivalent to those creatable by a flat specification. In fact, the flat specification is closer to the one we have already seen:

```
g1 : gen -o {t * gen * gen'}  %% begin a new stack
g2 : gen' -o {b * gen'}.      %% add a block to a stack
g3 : gen' -o {c}.             %% finish a block stack
g4 : gen -o {f}.
g4 : gen -o {h}.
```

The flat language representing the reachability set is:

$$(g_1)^*(g_2)^*(g_3)^*(g_4)^*(g_5)^*$$

[7] We could give a more precise characterization of the signature with additional regular language constructs, such as $(g_1 g_2^* g_3)^*(g_4|g_5|1)$, since exactly one of g_4 or g_5 will fire at the end exactly once (or neither of them will). However, prior work makes use of only the regular language constructs of concatenation and repetition, which affects their mapping into Presburger formulas. The Kleene star can be used to express this same thing because after g_4 fires, neither g_4 nor g_5 can, and if g_4 fires 0 times, then g_5 may fire 0 or 1 times.

In other words, the generative signature can be read operationally as follows: First, create an arbitrary number of seeds for new block stacks. Then, if applicable, we add blocks to all of those stacks. Then finish all of those stacks by marking their tops as clear. Then optionally generate a free arm, and if applicable (i.e. if we opted not to generate a free arm) generate an arm holding a block.

6.5.6 Computing Presburger reachability sets

In general, the postset of a vector set under an iterated word w^* can be given by binding an existential variable for the number of iterations, then computing an inductive definition of the union of those sets.

More precisely, to recapitulate Leroux's result [Ler13], define the *displacement* of a word $w = t_1 \dots t_k$ to be the vector $\delta(w) = \sum_{j=1}^{l} t_j$. Define also for w a configuration c_w such that $c_w(i) = \mathsf{max}\{-(t_1 + \cdots + t_j)(i) \mid 0 \leq j \leq k\}$. This value c_w denotes the minimal configuration for which there exists a run (applicable transition sequence) labeled by w.

Then, the postset of a set S under a single repeated word w is given by the following Presburger formula:

$$\mathsf{post}(S, w^*) = \{x \mid \exists v \in S. \exists n \in \mathbb{N}. v \geq c_w \wedge v + n\delta(w) = x\}$$

Then, for a VAS with starting configuration c_0 and represented as the flat language $w_1^* \dots w_n^*$, a sequence of Presburger sets C_i denoting the reachability set for the entire system is computed inductively:

$$\begin{array}{rcl} C_0 & = & \{c_0\} \\ C_i & = & \mathsf{post}(C_{i-1}, (w_i)^*) \end{array}$$

The reachability set is given by $v \in \mathsf{post}^*(\{c_0\}) = C_n$ where n is the number of words in the flat sequence.

6.5.7 The Algorithm

Now that we have a Presburger characterization of the reachability sets of generative invariants, we can describe what it means for a rule $r : A \multimap \{B\}$ to preserve an invariant (Σ, Δ_0) as a Presburger formula itself. We denote by $|\Delta_0|$, $|\Sigma|$, and so on the compilation of each of these linear logic program components into their corresponding vector addition system components.

Let $S = \mathsf{post}^*_{|\Sigma|}(|\Delta_0|)$. The rule r preserves Σ iff

$$\mathsf{post}_{|r|}(S) \subseteq S$$

In general, a postset $\mathsf{post}_{(t,k)}(S)$ can be denoted logically as the Presburger set $\{v \mid \exists x \in S. x \geq k \wedge x + t = v\}$.

And we can write the above subset condition as an implication; in particular, if q is the number of constants in the specification (i.e. the width of the vector):

$$\forall x, v \in \mathbb{Z}^q.(x \in S \wedge x \geq k \wedge x + t = v) \supset v \in S$$

Whether a given vector is in S is decidable since the set itself is Presburger computable.

6.5.8 End-to-End Example

Let's revisit the tricky case from Section 6.5.2.

```
g1 : gen -o {a * cs}.
g2 : gen -o {b * c * c}.
g3 : cs -o {c * cs}.
g4 : cs -o {1}.
```

As a vector addition system over `<gen, cs, a, b, c>`, we write this as a sequence of vector pairs `t ; k`:

```
g1 : <-1,  1, 1, 0, 0>   ; <1, 0, 0, 0, 0>
g2 : <-1,  0, 0, 1, 2>   ; <1, 0, 0, 0, 0>
g3 : < 0,  0, 0, 0, 1>   ; <0, 1, 0, 0, 0>
g4 : < 0, -1, 0, 0, 0>   ; <0, 1, 0, 0, 0>
```

The corresponding flat language is:

```
g2*g1*g3*g4*
```

The language compiles to the Presburger formula C_4, where

$$\begin{aligned}
C_0 &= \{\texttt{<1,0,0,0,0>}\} \\
C_1 &= \mathsf{post}(C_0, \mathsf{g2}^*) \\
C_2 &= \mathsf{post}(C_1, \mathsf{g1}^*) \\
C_3 &= \mathsf{post}(C_2, \mathsf{g3}^*) \\
C_4 &= \mathsf{post}(C_3, \mathsf{g4}^*)
\end{aligned}$$

If we omit needless existential quantification over rules that only apply a fixed number of times, we can easily generate this set by hand:

$$\begin{aligned}
C_0 &= \{\texttt{<1,0,0,0,0>}\} \\
C_1 &= \mathsf{post}(C_0, \texttt{g2}^*) \\
&= \{\texttt{<1,0,0,0,0>}, \\
&\quad \texttt{<0,0,0,1,2>}\} \\
C_2 &= \mathsf{post}(C_1, \texttt{g1}^*) \\
&= \{\texttt{<1,0,0,0,0>}, \\
&\quad \texttt{<0,0,0,1,2>}, \\
&\quad \texttt{<0,1,1,0,0>}\} \\
C_3 &= \mathsf{post}(C_2, \texttt{g3}^*) \\
&= \{\texttt{<1,0,0,0,0>}, \\
&\quad \texttt{<0,0,0,1,2>}, \\
&\quad \texttt{<0,1,1,0,0>}\} \cup \\
&\quad \{v \mid \exists n \in \mathbb{N}.v = \texttt{<0,0,1,0,n>}\} \\
C_4 &= \mathsf{post}(C_3, \texttt{g4}^*) \\
&= \{\texttt{<1,0,0,0,0>}, \\
&\quad \texttt{<0,0,0,1,2>}, \\
&\quad \texttt{<0,1,1,0,0>}\} \cup \\
&\quad \{v \mid \exists n \in \mathbb{N}.v = \texttt{<0,1,1,0,n>}\} \cup \\
&\quad \{v \mid \exists n \in \mathbb{N}.v = \texttt{<0,0,1,0,n>}\}
\end{aligned}$$

If we further restrict this set to exclude states with nonterminals by intersecting with the set $\{v \mid v_0 = 0 \wedge v_1 = 0\}$, we get the set of well-formed states

$$\{\texttt{<0,0,0,1,2>}\} \cup \{v \mid \exists n \in \mathbb{N}.v = \texttt{<0,0,1,0,n>}\}$$

Or in other words, the contexts $\{b, c, c\}$ and $\{a, c^n\}$ for arbitrary n, as expected.

Checking Program Rules

Now we can work out the decision procedure for the example on a couple of different program rules. We revisit two examples from Section 6.5.2, one that preserves the invariant and one that does not:

1. `a -o {a * c}.` (should pass)
2. `1 -o {c}.` (should fail)

Consider the postset of C along each rule: for rule 1, the only part affected is the second disjunct. $C' = \mathsf{post}_1(C) =$

$$\{\texttt{<0,0,0,1,2>}\} \cup \{x \mid \exists n.x = \texttt{<0,0,1,0,n>}\} \cup \{x \mid \exists n.x\ \texttt{<0,0,1,0,n+1}\}$$

But this last unioned set is a subset of $\{x \mid \exists n.x = \texttt{<0,0,1,0,n>}\}$ (which can be determined in Presburger Arithmetic by modeling subsethood as implication).

For rule 2,

$$\begin{array}{rcl} C'' & = & \mathsf{post}_2(C) \\ & = & \{\texttt{<0,0,0,1,2>}\} \cup \\ & & \{x \mid \exists n.\texttt{<0,0,0,1,n+2>}\} \cup \\ & & \{x \mid \exists n.x = \texttt{<0,0,1,0,n>}\} \cup \\ & & \{x \mid \exists n.x\ \texttt{<0,0,1,0,n+1>}\} \end{array}$$

This set is not a subset of the original, so it fails, as expected.

6.5.9 Adequacy and Correctness

The following lemmas serve as basic sensibility checks on the encoding we have used to devise our decidability proof.

Proposition 6.5.1 (Adequacy). *Modeling generative invariants as vector addition systems is sound and complete with respect to derivability (in the logic) and reachability (in the VAS). That is, formally, if $|\Sigma_{gen}, \Delta_0| = \langle V, c_0\rangle$ then $\Delta_0 \to *_{\Sigma_{gen}}\theta$, if and only if $c_0 \rightsquigarrow^*_V |\theta|$.*

This property follows by the standard correspondence between linear logic derivability and vector addition reachability [Kan95].

Proposition 6.5.2 (Soundness). *If the Presburger formula representing a rule $r : A \multimap B$ preserving a flat generative invariant $\langle\Sigma, \Delta_0\rangle$ has a proof, then for all Δ, $\Delta_0 \rightsquigarrow^*_\Sigma \Delta, A$ implies $\Delta_0 \rightsquigarrow^*_\Sigma \Delta, B$.*

Proof Sketch: Let $S = \mathsf{post^*}_{|\Sigma|}(|\Delta_0|)$. Let (t, k) be the transition and constraint of $|r|$. By correspondence with vector addition systems, the Presburger formula has a proof if and only if $\mathsf{post}_{|r|}(S) \subseteq S$.

The postset along $|r|$ is $\{v | \exists x \in S.x \geq k \wedge x + t = v\}$.

If we consider the vector respresentation of each side of the rule $|A|$ and $|B|$, we can write t as $|B| - |A|$ and k as $|A|$. So this postset can be written alternatively: $\{v | \exists x \in S.x \geq |A| \wedge x + |B| - |A| = v\}$.

We know $|\Delta, A| \in S$, and we need to show $|\Delta, B| \in S$.

Δ is in the set of vectors $\{v\}$ specified by the postset, because $|\Delta, A|$ is in S, is greater than $|A|$ by translation of contexts into vectors, and $|\Delta, A| + |B| - |A| = |\Delta, B|$. Thus $|\Delta, B| \in S$, as required. □

Proposition 6.5.3 (Completeness). *Converse of soundness. If, for all Δ, ($\Delta_0 \rightsquigarrow^*_\Sigma \Delta, A$) implies ($\Delta_0 \rightsquigarrow^*_\Sigma \Delta, B$) and there is a flat representation of Σ, then the Presburger formula representing invariant preservation of $\langle\Sigma, \Delta_0\rangle$ along $r : A \multimap B$ has a proof.*

Proof Sketch: Let $S = \mathsf{post^*}_{|\Sigma|}(|\Delta_0|)$. The Presburger formula has a proof if and only if $\mathsf{post}_r(S) \subseteq S$, i.e. for all v,

$$(\exists x \in S.x \geq |A| \wedge x + |B| - |A| = v) \supset v \in S$$

which is what we need to show.

What we know is that $\forall\Delta.(\Delta_0 \to^*_\Sigma \Delta, A) \supset (\Delta_0 \to^*_\Sigma \Delta, B)$, or in VAS terms,

$$|\Delta, A| \in S \supset |\Delta, B| \in S$$

Assume a given vector v, and assume of it the premise of what we need to show: that there exists an x such that $x \in S \wedge x \geq |A| \wedge x + |B| - |A| = v)$.

Now instantiate the $|\Delta|$ quantified over in our assumption at $x - |A|$. Now we know $x - |A| + |A| \in S \supset x - |A| + |B| \in S$. We know $x \in S$, so now we know $v \in S$, as required. $\square$

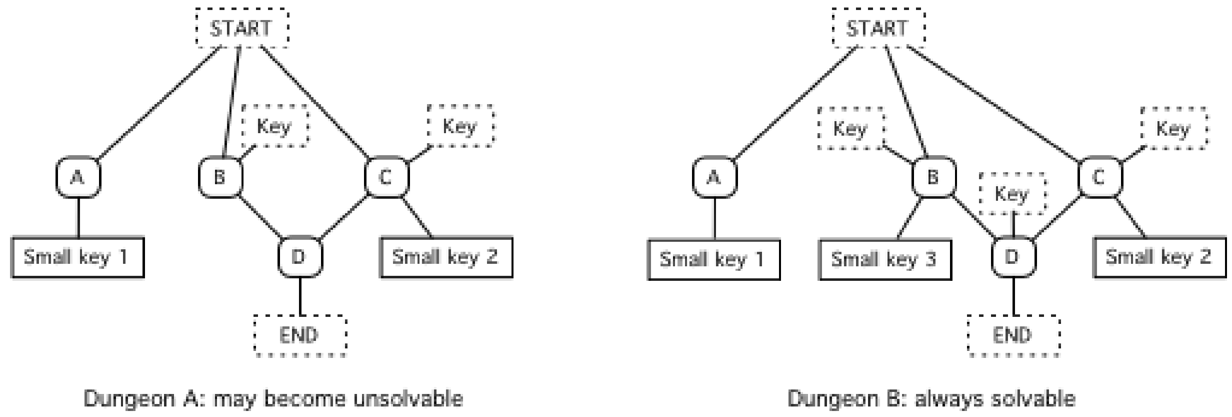
6.6 Conclusion

We have presented three methods for specifying program invariants, and for each discussed their potential for automation and their limitations. We have also presented a decidability proof for checking invariants of programs within a large fragment describing most of the programs we have considered in this thesis.

We close this chapter with a discussion of a limitation of all three of these systems, suggesting that there is still a great deal of investigation to do both in terms of automating specification checks and extending the expressiveness of specification languages.

6.6.1 Limitation: Instance-Dependent Invariants

One of our motivating examples for investigating program property proof is allowing the author to state and enforce *playability* constraints. For an example of a playability constraint, consider spatial navigation puzzles, such as the dungeon layouts in which locked doors separate some rooms and keys may be found in some rooms. Gareth Rees has studied this dungeon structure [8] in the context of The Legend of Zelda: Ocarina of Time [Nin98]; we replicate their diagram from Figure 3 here:



In this diagram, rooms (depicted with capidal letters) with "Key" labels require a key (any small key, indistinguishable) to open them, and rooms with "Small key" labels

[8]Article available at `http://garethrees.org/2004/12/01/ocarina-of-time/`.

contain such a key. For the dungeon to be *solvable* means that every room can eventually be reached by a player starting in the “Start” node.

We can easily give a program specifying the space of player actions for such a dungeon, which presumably include moving from room to room, picking up keys, and unlocking rooms that require a key when we have one:

```
door : type.
room : type.

entity : type.
player : entity.
small_key : entity.

adjacent room door room : bwd.
% defined by domain instance

locked door : pred.
open door : pred.

at entity room : pred.
holding entity entity : pred.

move : at player R * adjacent R D R’ * open D -o at player R’.
open_door : $at player R * adjacent R D R’ * locked D
            * holding player small_key
              -o open D.
pickup_key : $at player R * at small_key R
              -o holding player small_key.
```

Defining “accessibility” between locations requires some kind of recursive specification, which could perhaps be stated as a backward-chaining linear logic program or as a generative invariant of some kind. But note that any invariant that would describe solvability of a dungeon depends crucially on the specific initial configuration, since it is really a property of that configuration as well as of the program—and none of the techniques we have discussed so far encompass such a statement. Temporal logics, such as those used in model checking, present promising techniques that could account for properties like these, wherein it is feasible to exhaustively check the possible configuration space [CGP99]. In lieu of a concrete solution, however, we posit this example as a potential benchmark for game invariant specification techniques going forward.

Chapter 7

Conclusion

This dissertation has described the use of linear logic for representing game mechanics and narrative systems, unifying the ideas of play and narrative through the waypoint of logical proof. In particular:

- Chapter 2 described how to model narratives in linear logic, bringing out the dual notions of *alternative* and *simultaneous* structure in nonlinear stories, each corresponding to derivability principles in linear logic.
- Chapter 3 showed how to use story worlds modeled in linear logic as *narrative generators* by mapping proof search onto narrative generation. We showed how proof terms generated by a linear logic programming language contain simultaneous narrative structure, and we provide two tools, a visualization and query tool *CelfToGraph*, and a playable story generator *Quiescent Theater*, for investigating that structure.
- Chapter 4 presented a new programming language based on linear logic programming that can be used to author complex interactive simulations. We introduce the concept of *stages*, which add the ability to program with quiescence, allowing the program to pass control of the state between different components.
- Chapter 5 presented five case studies of linear logic programming, showcasing its use for modeling standard video game idioms like combat and resource management, board games idioms (Settlers of Catan), emergent behavior, social story worlds, and participatory theater.
- Chapter 6 presented three candidate conceptual frameworks for reasoning about linear logic programs, including several example proofs carried out by hand. We gave a proof of decidability for a substantial fragment of the framework, building the initial pathways toward automated reasoning tools.

Recall the thesis statement from Chapter 1:

Thesis statement: *Using linear logic to model interactive worlds enables rapid prototyping of experimental game designs and deeper understanding of narrative structure.*

We have supported this thesis in the following ways: Chapters 2 and 3 identify key aspects of narrative structure, alternatives and simultaneity, and show how linear logic proofs carry these structures intrinsically, providing *deeper understanding of narrative structure*. Chapters 4 and 5 shows how a concrete implementation of linear logic as a modeling language can be used to build interactive artifacts that support experimentation and iteration on the design of mechanics, supporting *rapid prototyping of experimental game designs*. Chapter 6 provides auxilliary support to the *rapid prototyping* part of the thesis statement, since simple invariants are often critical to program correctness. Having a framework of proof for properties of linear logic programs provides the basis for automatic tools that may help eliminate bugs.

Our findings constitute a foundation for unifying common formal aspects of digital games, analog games, participatory theater, and generative storytelling in a way that we believe furthers the prospects for creativity in all of these domains and their combinations.

7.1 Contributions

Because this thesis bridges two disciplines, logic/programming language theory and game/narrative design, we identify its core contributions in each of these two contexts.

7.1.1 Game and Narrative Design

For game and narrative design, our primary contribution has been that of establishing logical foundations for the field. We worked out the correspondence between proof and story in a way that enables programming with logical constructs to generate and analyze stories, and we showed how that same programming model could be used to describe game mechanics and interactive story worlds in a flexible and general way.

We believe that a logical foundation fills a gap in current game design tool theory. Current thought recognizes that people highly skilled in designing digital game mechanics need not also be highly skilled in programming in a general-purpose language. Thus, several tools have emerged as "end-user programming" languages for game designers—they tend to offer limited programming affordances (e.g. "attach a behavior to an object" or "link two passages of text") rather than general control constructs. But a common problem with these tools is that once the "end user" develops *mastery* with the limited affordances, they cannot grow their skill within the same framework: they need to break abstraction barriers and learn the more complex language underlying the tool.

A logical foundation for executable game descriptions addresses this problem: linear logic itself contains very few "features" to understand initially (effectively just $\otimes$ and $\multimap$), and Ceptre adds a few more (type declarations, stages, quiescence, and traces). But a *logic* as a formal structure is something that can be extended in a variety of ways to accommodate new domains and new representational concerns, while still being grounded in fundamental principles like soundness and completeness to guide the design of extensions. Having those fundamental principles as an anchor for the tool design means that we can grow the language with a sound basis for determining when new features work seamlessly with old ones (for instance, generalizing linear logic to the first order case) and how to soundly incorporate more complex feature interactions (for instance, introducing the $\{-\}$ (lax) modality in Celf to allow programmer control of forward- and backward-chaining). The question of whether appropriate logical connectives will always be available for the most salient author-facing needs has yet to be fully investigated, but we believe there are several promising avenues of future work in this vein.

Secondarily, the particular choice of linear logic for formalizing game rules, forcing a resource-oriented viewpoint, has enabled the following observation about game design authoring affordances: in order for rules to create harmonious interaction, they must effectively come in pairs. That is, if a rule *produces* some fact p (by containing p in its conclusion), some other rule should *consume* p, and vice versa. While this may seem like a simple observation for linear logic programs, tools such as Kodu and Twine do not always provide the means for an author to understand or create this duality. For instance, in Kodu, an entity can be programmed to `move towards` some other entity, and an entity can be programmed to do something whenever it `bumps` another entity. This action-sensor pair, `move towards` and `bump`, work crucially by their interaction with one another: an entity that moves toward another entity will eventually bump into it. But that relationship is not codified by the tool in the same way that those mechanics would be specified in linear logic. In languages like GameMaker, Stencyl, and Unity, behaviors may be attached to game entities, but there is no visible means of determining whether any of the game's progression depends meaningly on those behaviors; similarly, one may write code that case-analyzes certain behaviors that do not actually arise. Noticing the ubiquity of this pattern in linear logic leads us to make a recommendation for game creation tool designers: for every behavior that an author can add to their game, ensure that programming facilities exist both for introducing that behavior and depending on its outcome.

7.1.2 Programming Languages and Logic

In the area of logic-based programming language theory and design, we have contributed a large body of examples and studies on its use for the specific domains presented here, building an argument for the language design methodology in general.

By developing a new linear logic language designed around interactivity, we have also contributed to a better understanding of the logic, its use as a programming language, and forward chaining proof construction. In particular, we have clarified the notion that programming with *quiescence* enables a broad range of idioms that were unavailable before, and that it can ultimately be accounted for by the semantics already available in Celf (as shown in Chapter 4).

Finally, we also contribute results about the metatheory of linear logic that are broadly applicable for reasoning about concurrent state change, not just in games. We have established that the checking of *generative invariants* is decidable for propositional linear logic programs and first-order linear logic programs with finite term domains (Chapter 6), paving the way for development of practical automated checking tools.

7.2 Future Work

We propose several avenues for future work, divided into the subjects to which we hope to make further contributions: narrative representation and modeling; generative methods; accessible game design tools; and facilities for computer-aided reasoning about game and narrative specifications.

7.2.1 Narrative Representation

The primary notion in narratives that we feel bears further investigation from a logical standpoint is *knowledge*: our current representation of narrative states presents all information about the story world in one monolithic body of state (the linear context), failing to distinguish between information that is localized to particular agents and knowledge that is communicated between them. Further, in the *discourse* aspects of narrative [You07], i.e. the narration of the story to a reader or viewer (recipient), the recipient's knowledge becomes relevant. In many films or television shows, each scene serves primarily to reveal part of the information state to the recipient, sometimes truthfully but sometimes deceptively or ambiguously (the "unreliable narrator"). Other work on interactive narrative has explored the modeling of player and character knowledge to create more flexible authorial imposition, as well [RY13]. Some facets of knowledge can be represented in linear logic—e.g. it is easy enough to create a `knows C M` predicate, where `M` is some pre-defined message type—but modeling knowledge of *logical* information where `M` instead represents some linear logic proposition and knowledge is reflected by logical provability seems necessary in order to capture story discourses involving mystery, deception, and surprise. Thus, we propose the use of epistemic modal logics [Ver02] (and their integration with linear logic [GBB$^+$06]) as a candidate for representating character knowledge in narratives, and we would like to continue the programme of investigating proof theory for narrative representation via such a logic.

Related to knowledge, but perhaps more feasible to explore in linear logic, is the idea of *narrative focalization*:[1] a generalization of "point of view" (first, second, third person) that describes what the narrative recipient learns throughout the story in relationship to what the characters learn. Three kinds of narrative focalization are considered standard:[2]zero, internal, and external. Zero focalization is the *omnicient* point of view where the recipient knows more than the character or characters, i.e. the entire narrative state; internal focalization is the point of view equated with a particular character's knowledge; and external focalization is a point of view that does not reveal character interiority, instead depicting only what the characters' actions would reveal to an objective

[1]Not to be confused with logical focalization.

[2] See `http://wikis.sub.uni-hamburg.de/lhn/index.php/Focalization` for an overview.

witness. We propose using (perhaps annotated) causally-structured traces as artifacts for representing a single narrative model that can be told with arbitrary focalization—the choice of which events to reveal during narration could arise from the combined specification of a narrative trace and a focalization scheme.

A third path of investigation is that of comparing plot structures in multiple narratives. Game-o-Matic [TBMB12] is a template for this idea in the domain of games: an abstract specification of mechanics as 2D movement and collision in Game-o-Matic can be given multiple graphical realizations and shown to model different narratives when interpreted by humans. Similarly, we would like to investigate narratives that have the same trace structure—or Twine games that have the same branching choice structure—and compare their interpreted content, answering the research question of how much structure (in the senses we have identified) informs overall meaning. In a sort of dual direction, we wonder whether narratives that have "the same" plot, such as *Into the Wild* and *Death of a Salesman*, would actually be revealed to have similar structure in linear logical terms, or whether it is instead mainly the interpretive themes that connect them.

We propose the domain of fables and myths as a starting point for the above investigation, perhaps in combination with the PLOTTO book of abstracted plot formulas [Coo28], for narratives that are usually small and abstract enough to formalize on relatively objective terms. The hypothesis to test in this case would be that a formalization of the plot abstraction in linear logic could be given different rendering parameters to generate muliple existing examples of that abstraction in popular culture.

7.2.2 Generative Methods

In a talk at the 2015 Game Developers Conference surveying the field of procedural content generation (or generative methods),[3] the speakers describe classes of techniques distinguished by whether they work *top-down* or *bottom-up*: whether content is generated by way of selecting rules that match certain constraints (top down) or by way of randomly selecting instantiations for variable parameters in the content definition. Backward- and forward-chaining proof construction are sometimes also called top-down and bottom-up (respectively), with very similar meanings to this distinction. While we have investigated generative methods for *narrative* generation using linear logic, we wonder how well the technique could extend to other forms of content generation.

To make the idea a bit more explicit, one idea in bottom-up content generation is that of *production grammars*, such as Lindenmayer systems [PH13] for generating line-based images. The production grammar technique is classified as bottom-up. Grammars and rewriting systems have a well-established correspondence, and the propositional forward-chaining fragment of linear logic has been shown to correspond to multiset rewriting, so at the very least we can express multiset-like production grammars as simple linear logic programs—this would mean using programs like the *generative signatures* described in Chapter 6 to *actually generate* the context sets they characterize.

[3] `http://www.gdcvault.com/play/1022134/Making-Things-Up-The-Power`

(Further work would need to be done to determine how to operationalize those signatures, since they do not always have clear operational semantics.)

If we allow for first-order term variables restricted to binary predicates, then we seem to have a viable grammatical interpretation of these programs as *graph grammars*, or ways of rewriting directional graphs (nodes and potentially-labeled edges). Each binary predicate can be said to model an edge (where the label is the predicate name). Graph grammars have been used quite fruitfully for generative methods, using a graph structure to model game levels, for instance, and generating levels based on graph production rules [Dor10]. The research question we pose is whether proof search on linear logic programs representing graph grammars could be used to similar, or more fruitful, ends.

Finally, a recent concern of generative methods researchers is that of *expressive range analysis*, or determining how the output of a content generator can vary along certain axes [SW10]. Our work on generative invariants, if successful in the future, would be a strong candidate as a framework for carrying out this analysis: generative invariants can be seen as a way of specifying a vector space to which all executions of a linear logic program must stay confined, a similar notion to Smith and Whitehead's characterization of generative variety.

7.2.3 Accessible Game Design Tools

One of the major limitations of this work is that we have developed Ceptre as a core calculus rather than as a deliberately end-user-facing tool, due to our priorities defined by the scope of this thesis. In the future, we would like to develop a front-end development tool for Ceptre that would allow for visual specification of initial states and rules, and corresponding visualization of intermediate states as the program runs. This problem itself poses various research challenges, since the generality of the language means there will be no uniformly appropriate way of representing states visually. As a way forward, we propose using sensing and acting predicates hooked up to inter-process communication to provide an API for programmers to build their own visualization schemas. We would like to provide a few common schemas we imagine Ceptre being used for, such as a web-based interactive fiction parser and renderer, and a grid-based tile renderer and key press processing event loop, using visual rule-based languages such as Kodu, StageCast Creator, [4] and PuzzleScript as inspiration for the design of such a tool in these limited domains.

In the long run, we imagine that a language based on linear logic (not necessarily Ceptre) could be built into a *mixed initiative design* tool [SWM10] that can reason about game designs and provide feedback for a human novice designer on their own designs, using logical proofs as formal models of *design justification*. One proposal for a way forward with such a system would be to build a library of *patterns*, such as the mechanic patterns described at the following site: `http://www.jorisdormans.nl/machinations/wiki/index.php?title=Pattern_Library` Ideally, we could model these

[4] `http://www.stagecast.com/`

patterns in such a way that their recombination could be characterized via logical provability, providing a formal basis for compositional pattern-based design.

A recombination tool for game design patterns has been described in prior work [Orw00], which was used fruitfully in a board-game-like subspace of game designs—we would like to investigate its use for mechanics better suited to linear logical investigation, like emergent story worlds, quests, and resource management strategy.

7.2.4 Reasoning Tools

Finally, we posit that there is much work left to do in terms of building reasoning tools for game prototyping. Game designers frequently express the need for such tools, including hobbyist interactive fiction authors, as evidenced by Andrew Plotkin's development of the *PlotEx* tool for modeling and querying interactive fiction puzzle mechanics.[5] One of the major barriers to the success of simulationist interactive media is the lack of ability to predict and control for unexpected behavior from rules that are too general (or not general enough), and being able to state design intents that are automatically checked would be a major step forward.

Our most-developed avenue of future work to meet this goal is to continue developing the theory of equality for traces that arise from generative signatures such that the admissibility of their reconfiguration witnessing the preservation of an invariant across a rule is decidable. We conjecture that a suitable representation with fewer dependencies to reason about than concurrent traces will be needed to move this idea forward.

Further afield, we note that *answer set programming* (ASP) is a technique that has been investigated quite fruitfully to generate games via authors expressing constraints [SM11], where those constraints are not unlike the kinds of properties we are interested in verifying for hand-authored games. This approach lends itself to a kind of *correct-by-construction* mode of game prototyping. However, the process of *grounding* logic programs in ASP by instantiating each rule with every possible combination of terms makes the specifications less practical as executable artifacts, for which linear logic programs seem better suited. We speculate that a hybrid technique to game specification could be devised wherein linear logic programs could be checked *against* an ASP specification defining its correctness criteria. Along similar lines, we speculate that there is some duality between the constructive logic program *proof* representing a play trace and ASP's *model* representing the same, and we would like to explore those connections more explicitly.

7.3 Final Remarks

We have presented the use of linear logic to model interactive worlds across a broad spectrum of use cases. Linear logic itself has proved a useful tool for this domain, but we suggest that the broader take-away of this work is that the *methodology* of proof theory

[5] `http://eblong.com/zarf/plotex/`

can be quite useful in application to creative tasks. Logic, with all its flaws for capturing the reality of human reasoning, serves as an important common language for translating between human intuitions and executable artifacts. We present this thesis as a first step in establishing that proof-theoretic methodology, with its prescription for logic design and its application to programming languages and knowledge representation, can be a torchlight guiding the way through the twisty passages of understanding computer-assisted creativity.

Appendix A

CLF Semantics

In this appendix, we briefly discuss CLF/Celf's proof-theoretic basis for stratifying forward and backward chaining search.

Recall from Chapter 2: $\Gamma; \Delta \to \Gamma'; \Delta'$ is a permissible program step whenever it is the case that

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

is a permissible inference for an arbitrary conclusion γ.

We can rewrite the left sequent rules for connectives as transition rules in this fashion, as shown in prior work [DSC12]:

$\otimes L$:

$$(\Gamma; \Delta, A \otimes B) \to (\Gamma; \Delta, A, B)$$

$!L$:

$$(\Gamma; \Delta, !A) \to (\Gamma, A; \Delta)$$

copy:

$$(\Gamma, A; \Delta) \to (\Gamma, A; \Delta, A)$$

$\multimap L$:

$$\frac{\Gamma; \Delta' \vdash A}{(\Gamma; \Delta, \Delta', A \multimap B) \to (\Gamma; \Delta, B)}$$

Note that this last rule appeals to the standard judgment $\Gamma; \Delta \vdash A$ to solve for the antecedent of the rule, which might be immediately available in $\Gamma; \Delta$, or it might require backward-chaining search.

In a *purely forward chaining* system, we would require that all atoms in A appear already in the context at the time of selecting the rule; that is, we can rewrite the transition $\multimap L'$:

$$(\Gamma; \Delta, a_1, \ldots, a_n, a_1 \otimes \ldots \otimes a_n \multimap B) \to (\Gamma; \Delta, B)$$

If we further restrict the logic program so that *rules* $A \multimap B$ only appear in the program signature Σ, never the context, then we may understand rules of the form

$a_1 \ldots a_n \multimap b_1 \ldots b_m$ (where, again, atoms $a_1 \ldots a_n$ *only* arise as the consequents of forward-chaining rules) as inducing a corresponding transition

$$\Delta, a_1, \ldots, a_n \rightarrow \Delta, b_1, \ldots, b_m$$

for any Δ. [1]

If, further, no quantification over terms is permitted, then what we have produced is a fragment of the logic corresponding exactly to *multiset rewriting* [CS09]: a program is simply a set of rewrite rules and an initial multiset, and executions of the program correspond to ways that the initial multiset may be rewritten into stable (quiescent) multisets. Note that such a rewrite system is still *nondeterministic* in that multiple rules may apply; so too, then, is the semantics of the program: the same program and initial configuration may result in distinct states during distinct runs. For this reason, multiset rewriting systems have sometimes been considered a good basis for modeling concurrent and distributed computation [Mes90]. In the narrative generation setting, the nondeterminism of program semantics is what gives rise to variable story generation from a given story world.

In Celf, the semantics are slightly more complicated than this multiset rewriting depiction, because forward and backward chaining may be used in combination. CLF, Celf's theoretical basis, introduces a connective $\{A\}$ to specify forward chaining. We next give the concrete syntax of Celf and overview its semantics on an intuitive level.

An idealized version of Celf stratifies propositions into two categories, *positive* propositions S and *negative* propositions A, B, based on a theory called focusing [And92]. Using this stratification together with the connective $\{S\}$ for casting a positive proposition as a negative one, it effectively permits two kinds of rules, one each corresponding to forward and backward chaining: rules $A \multimap B$ are available in backward- *or* forward-chaining phases of the program, and rules $A \multimap \{S\}$ are available only in forward-chaining phases.

As an example, consider a program with the rules $r_1 : a \multimap b$, $r_2 : b \multimap \{c\}$. Given the goal sequent $\cdot; a \vdash c$, proof search will not succeed: since c is not of the form $\{A\}$, proof search will use backward chaining, and backward chaining on the goal c does not turn up any rules it can use (because r_2 is only available in forward-chaining).

If, however, we initialize the program with the sequent $\cdot; a \vdash \{c\}$, the proof will succeed: we will enter forward chaining, notice that a forward chaining rule (r_2) is available if only we can prove its antecedent b, which then initiates a *backward* chaining phase of search for the goal b. The rule r_1 is available in backward chaining, so we will try it (since its head matches the goal b), recursively solving for the goal a, which is available in the context, so search succeeds. Now we can use the rule r_2 to evolve the context to c, which is quiescent. At that point, we check whether the context matches the goal: in this case, the goal is c, which does indeed match.

[1] Note that parametricity over the context Δ, representing the ambient state that is irrelevant to the specific rule r, stays the same across the transition. This fact evades the *frame problem* present in other settings, such as event and situation calculi [Hay71].

Appendix B

Ceptre Typing and Kinding Rules

This appendix contains typing rules for terms and kinding rules for predicate constructors. They say what it means for a Ceptre program to be well formed at the syntactic level. Most of these rules implicitly carry a signature Σ, which is not mentioned unless the rule refers to it.

$$\Sigma \vdash K \ \mathsf{wf}$$

$$\frac{}{\Sigma \vdash \mathsf{type}\ \mathsf{wf}}\ \mathsf{wf/type} \qquad \frac{}{\Sigma \vdash \mathsf{pred}\ \mathsf{wf}}\ \mathsf{wf/pred} \qquad \frac{}{\Sigma \vdash \mathsf{bwd}\ \mathsf{wf}}\ \mathsf{wf/bwd}$$

$$\frac{\Sigma \vdash \tau : \mathsf{type} \quad \Sigma \vdash P\ \mathsf{wf}}{\Sigma \vdash \tau \to P\ \mathsf{wf}}\ \mathsf{wf/arr}$$

(Note: this last rule refers to P rather than K because type cannot be indexed by type arguments in Ceptre.)

$$\Gamma \vdash_\Sigma A : K$$

$$\frac{a{:}K \in \Sigma \quad \Sigma \vdash K\ \mathsf{wf}}{\Gamma \vdash_\Sigma a : K}\ \mathsf{ofk/const} \qquad \frac{\Gamma \vdash a : \tau \to K \quad \Gamma \vdash t : \tau}{\Gamma \vdash a\, t : K}\ \mathsf{ofk/app}$$

$$\frac{\Gamma \vdash \tau_1 : \mathsf{type} \quad \Gamma \vdash \tau_2 : \mathsf{type}}{\Gamma \vdash \tau_1 \to \tau_2 : \mathsf{type}}\ \mathsf{oft/arr}$$

$$\frac{\Gamma, x{:}\tau \vdash M : \mathsf{bwd}}{\Gamma \vdash \Pi x{:}\tau.M : \mathsf{bwd}}\ \mathsf{ofb/pi} \qquad \frac{\Gamma \vdash p : \mathsf{bwd} \quad \Gamma \vdash B : \mathsf{bwd}}{\Gamma \vdash p \to B : \mathsf{bwd}}\ \mathsf{ofb/arr}$$

$$\Gamma \vdash_\Sigma t : \tau$$

$$\frac{c{:}\tau \in \Sigma}{\Gamma \vdash_\Sigma c : \tau}\ \mathsf{oft/const} \qquad \frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\ \mathsf{oft/var}$$

$$\frac{\Gamma \vdash c : \tau' \to \tau \quad \Gamma \vdash t : \tau'}{\Gamma \vdash c\, t : \tau}\ \mathsf{oft/app}$$

Appendix C

Code Listings

C.1 First Past the Post Voting Protocol

```
nat : type.
z : nat.
s nat : nat.

candidate : type.
elected candidate : pred.
defeated candidate : pred.
ballot candidate : pred.
hopeful candidate nat : pred.

stage count = {
  count_ballot : ballot C * hopeful C N -o hopeful C (s N).
}

- : qui * stage count -o stage pick * max z.

lt nat nat : bwd.
lt/z : lt z (s N).
lt/s : lt (s N) (s M)
      <- lt N M.

max nat : pred.

stage pick = {
  increase : max N * hopeful C N' * lt N N'
               -o hopeful C N' * max N'.
  eliminate : max N * hopeful C N' * lt N' N
               -o max N * !defeated C.
}
```

```
- : qui * stage pick * hopeful C N -o stage done * !elected C.

stage done = {
  cleanup : max _ -o ().
}

% domain instantiation
alice : candidate.
bob : candidate.
charlie : candidate.

context init =
{ballot alice, ballot alice, ballot charlie, ballot bob,
ballot bob, ballot alice,
hopeful alice z,
hopeful bob z,
hopeful charlie z }.

#trace _ count init.
```

C.2 Sokoban

```
layer : type.
location : type.
entity : type.
direction : type.
intention : type.

% Instantiations that exist for every game
player : entity.

go direction : intention.
x : intention.
stationary : intention.

right : direction.
left : direction.
up : direction.
down : direction.

available intention : bwd.
available/left : available (go left).
available/right : available (go right).
available/down : available (go down).
available/up : available (go up).
```

```
available/x : available x.

% Persistent facts
adjacent location direction location : bwd.

% Game state that exists for every game
at layer location entity intention : pred.
empty layer location : pred.
turn : pred.

% Game-specific terms and state
crate : entity.

stage impartIntentions = {
  press_arrow :
    turn
      * available Intent
      * at Layer Loc player _
    -o at Layer Loc player Intent.
}
#interactive impartIntentions.
qui * stage impartIntentions -o stage processIntentions * turn.

% this is the part written by the game author in PuzzleScript
stage processIntentions = {
  % propagate player intention to crate
  push_crate :
    turn *
    $at Layer Loc player (go Dir) * adjacent Loc Dir Loc'
      * at Layer Loc' crate _
    -o at Layer Loc' crate (go Dir).
}
qui * stage processIntentions -o stage carryOutIntentions.

% resolve intentions in a nondeterministic order.
stage carryOutIntentions = {
  move :
    at Layer L Ent (go Dir) * adjacent L Dir L'
       * empty Layer L' -o at Layer L' Ent stationary * empty Layer L.
}
qui * stage carryOutIntentions -o stage cleanup.

% friction! otherwise stuff w/intent to move would keep moving...
stage cleanup = {
  spare_turns : turn -o ().
  friction : at Layer L Ent (go Dir) -o at Layer L Ent stationary.
}
qui * stage cleanup -o stage impartIntentions * turn.

% layer/level definitions
bg : layer.
fg : layer.

cell00 : location.
cell01 : location.
cell02 : location.
cell10 : location.
cell11 : location.
cell12 : location.
cell20 : location.
cell21 : location.
cell22 : location.

% 00 01 02
adjacent cell00 right cell01.
adjacent cell01 left cell00.
adjacent cell01 right cell02.
adjacent cell02 left cell01.
adjacent cell00 down cell10.

% 10 11 12
adjacent cell10 up cell00.
adjacent cell10 right cell11.
adjacent cell11 left cell10.
adjacent cell11 up cell01.
adjacent cell01 down cell11.
adjacent cell11 right cell12.
adjacent cell12 left cell11.
adjacent cell12 up cell02.
adjacent cell02 down cell12.

% 20 21 22
adjacent cell10 down cell20.
adjacent cell20 up cell10.
adjacent cell20 right cell21.
adjacent cell21 left cell20.
adjacent cell21 up cell11.
adjacent cell11 down cell21.
adjacent cell21 right cell22.
adjacent cell22 left cell21.
adjacent cell22 up cell12.
adjacent cell12 down cell22.
```

```
context init =
{ turn,
  at fg cell00 player stationary,
  at fg cell01 crate stationary,
  empty fg cell02,
  empty fg cell10,
  empty fg cell11,
  empty fg cell12,
  empty fg cell20,
  empty fg cell21,
  empty fg cell22}.

#trace _ impartIntentions init.
```

C.3 Tower of Hanoi

```
ring : type.
smaller ring ring : bwd.

place : type.
post : type.
top_of ring : place.
bottom post : place.
post : place.

on ring place : pred.
clear place : pred.
arm_free : pred.
arm_holding ring : pred.

stage do = {
pickup
  : clear (top_of R) * on R P * arm_free -o arm_holding R * clear P.

putdown_on_ring
  : arm_holding R * clear (top_of R') * smaller R R'
      -o arm_free * on R (top_of R') * clear (top_of R).

putdown_on_post
  : arm_holding R * clear (bottom P)
      -o arm_free * on R (bottom P) * clear (top_of R).
}
#interactive do.
```

```
% domain instantiation

p1 : post.
p2 : post.
p3 : post.

r1 : ring.
r2 : ring.
r3 : ring.

smaller r1 r2.
smaller r1 r3.
smaller r2 r3.

context init =
{clear (bottom p2), clear (bottom p3),
    on r3 (bottom p1),
    on r2 (top_of r3),
    on r1 (top_of r2),
    clear (top_of r1),
    arm_free}.

#trace _ do init.
```

Bibliography

[Ada89] Jon-K Adams. Causality and narrative. *Journal of Literary Semantics*, 18(3):149–162, 1989. 2.1

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992. 2.3, A

[AR09] Manuel Araújo and Licínio Roque. Modeling games with Petri nets. *Breaking New Ground: Innovation in Games, Play, Practice and Theory. DIGRA2009. Londres, Royaume Uni*, 2009. 2.6.2

[ARLG$^+$09] 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. 4.3.3, 4.4

[Art08] Electronic Arts. Spore. *Electronic Arts*, 2008. 1.2

[Ayl99] Ruth Aylett. Narrative in virtual environments-towards emergent narrative. In *Working notes of the Narrative Intelligence Symposium*, 1999. 1.2

[Bac78] John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978. 1

[Bae08] David Baelde. *A linear approach to the proof-theory of least and greatest fixed points*. PhD thesis, PhD thesis, Ecole Polytechnique, 2008. 6.2.3

[BCC10] Anne-Gwenn Bosser, Marc Cavazza, and Ronan Champagnat. Linear logic for non-linear storytelling. *Frontiers in Artificial Intelligence and Applications*, 215: ECAI 2010:713–718, 2010. 1.5, 2.1, 2.7, 4.4

[BCFC11] Anne-Gwenn Bosser, Pierre Courtieu, Julien Forest, and Marc Cavazza. Structural analysis of narratives with the Coq proof assistant. In *ITP*, 2011. 2.6

[BH04] Staffan Bjork and Jussi Holopainen. Patterns in game design (game development series). 2004. 5.6.3

[Bro74] Edward J Brown. The formalist contribution. *Russian Review*, pages 243–258, 1974. 2.2

[Bro08] Cameron Browne. *Automatic generation and evaluation of recombination* *games*. PhD thesis, Queensland University of Technology, 2008. 5.3.6

[Bru90] Jerome S Bruner. *Acts of meaning*. Harvard University Press, 1990. 2.1

[CCM02] Marc Cavazza, Fred Charles, and Steven J. Mead. Character-based interactive storytelling. *IEEE Intelligent Systems*, 17(4):17–24, 2002. `doi:http://doi.ieeecomputersociety.org/10.1109/MIS.2002.1024747`. 2.1, 5.1.4

[CCP03] Bor-Yuh Evan Chang, Kaustuv Chaudhuri, and Frank Pfenning. A judgmental analysis of linear logic. 2003. 2.3

[CGP99] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999. 6.6.1

[Chu32] Alonzo Church. A set of postulates for the foundation of logic. *Annals of mathematics*, pages 346–366, 1932. 1

[CJ08] Nathanael Chambers and Daniel Jurafsky. Unsupervised learning of narrative event chains. In *ACL*, volume 94305, pages 789–797. Citeseer, 2008. 2.1

[CMC01] Fred Charles, Steven J Mead, and Marc Cavazza. Character-driven story generation in interactive storytelling. In *vsmm*, page 609. IEEE, 2001. 2.6.1

[COM13] Kate Compton, Joseph C Osborn, and Michael Mateas. Generative methods. In *The Fourth Procedural Content Generation in Games workshop, PCG*, 2013. 1.2

[Coo28] William Cook. *PLOTTO: the master book of all plots*. Ellis Publishing Company, 1928. 7.2.1

[CPP08] Kaustuv Chaudhuri, Frank Pfenning, and Greg Price. A logical characterization of forward and backward chaining in the inverse method. *Journal of Automated Reasoning*, 40(2–3):133–177, 2008. `doi:10.1007/s10817-007-9091-0`. 3.1, 3.6

[CPWW03] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework ii: Examples and applications. Technical report, DTIC Document, 2003. 6.2.3

[CRGP14] Flavio Cruz, Ricardo Rocha, Seth Copen Goldstein, and Frank Pfenning. A linear logic programming language for concurrent programming over graph structures. *arXiv preprint arXiv:1405.3556*, 2014. 4.3.3

[CS09] Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic (full-version). *Information and Computation*, 207(10):1044–1077, 2009. A

[Cus00] James Cussens. Stochastic logic programs: Sampling, inference and applications. In *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, pages 115–122. Morgan Kaufmann Publishers Inc., 2000. 5.6.3

[CW77] William Crowther and Don Woods. Adventure. 1977. 1

[Dah12] Michael F Dahlstrom. The persuasive influence of narrative causality:

Psychological mechanism, strength in overcoming resistance, and persistence over time. *Media Psychology*, 15(3):303–326, 2012. 2.1

[DCA13] Kim Dung Dang, Ronan Champagnat, and Michel Augeraud. A methodology to validate interactive storytelling scenarios in linear logic. *T. Edutainment*, 10:53–82, 2013. 2.6.2, 3.6

[DHCS11] Kim Dung Dang, Steve Hoffmann, Ronan Champagnat, and Ulrike Spierling. How authors benefit from linear logic in the authoring process of interactive storyworlds. In *ICIDS*, pages 249–260, 2011. 2.6.2, 6.5.1

[DHL^{+}98] Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomír Měch, Matt Pharr, and Przemyslaw Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 275–286. ACM, 1998. 5.4.5

[Dor10] Joris Dormans. Adventures in level design: generating missions and spaces for action adventure games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, page 1. ACM, 2010. 7.2.2

[Dor11] Joris Dormans. Simulating mechanics to study emergence in games. In *Artificial Intelligence in the Game Design Process*, 2011. 2.6.2, 5.2.4, 5.6.2, 5.6.4

[DOY06] Dino Distefano, Peter W Ohearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 287–302. Springer, 2006. 6.2.3

[DS12] Henry DeYoung and Carsten Schürmann. Linear logic voting protocols. In *VoteID 2011*, pages 53–70. Springer LNCS 7187, 2012. 4.3.4

[DSB09] Lucas Dixon, Alan Smaill, and Alan Bundy. Verified planning by deductive synthesis in intuitionistic linear logic. In *ICAPS Workshop on Verification and Validation of Planning and Scheduling Systems*, 2009. 2.6.1

[DSC12] Yuxin Deng, Robert J Simmons, and Iliano Cervesato. Relating reasoning methodologies in linear logic and process algebra. *Mathematical Structures in Computer Science*, pages 1–39, 2012. A

[DST09] Lucas Dixon, Alan Smaill, and Tracy Tsang. Plans, actions and dialogues using Linear Logic. *Journal of Logic, Language and Information*, 18(2):251–289, 2009. `doi:http://dx.doi.org/10.1007/s10849-008-9079-0`. 2.6.1

[ELL^{+}13] Marc Ebner, John Levine, Simon M Lucas, Tom Schaul, Tommy Thompson, and Julian Togelius. Towards a video game description language. *Artificial and Computational Intelligence in Games*, 6:85–100, 2013. 4.8

[ES] R Evans and E Short. Versu: A simulationist storytelling system. 1, 1.1, 4.8, 5.1.4

[Esp97] Javier Esparza. Petri nets, commutative context-free grammars, and basic parallel processes. *Fundamenta Informaticae*, 31(1):13–25, 1997. 6.5.4

[FDM^{+}11] Teale Fristoe, Jill Denner, Matt MacLaurin, Michael Mateas, and Noah Wardrip-Fruin. Say it with systems: Expanding Kodu's expressive power through gender-inclusive mechanics. In *Proceedings of the 6th International Conference on Foundations of Digital Games*, FDG 2011, pages 227–234, New York, NY, USA, 2011. ACM. URL: `http://doi.acm.org/10.1145/2159365.2159396`, `doi:10.1145/2159365.2159396`. 4.6.1

[Fla01] Gustave Flaubert. *Madame Bovary*. Folio, 2001. In French. Also available in English, OUP Oxford, reissue edition, 2008. 2.6.2

[FN71] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. In *Proceedings of the 2Nd International Joint Conference on Artificial Intelligence*, IJCAI'71, pages 608–620, San Francisco, CA, USA, 1971. Morgan Kaufmann Publishers Inc. URL: `http://dl.acm.org/citation.cfm?id=1622876.1622939`. 2.6.1

[FO97] Laurent Fribourg and Hans Olsén. *Proving safety properties of infinite state systems by compilation into Presburger arithmetic*. Springer, 1997. 6.5.4

[FR98] Michael J Fischer and Michael O Rabin. *Super-exponential complexity of Presburger arithmetic*. Springer, 1998. 6.5.1

[Fri00] Laurent Fribourg. Petri nets, flat languages and linear arithmetic. In *WFLP*, pages 344–365. Citeseer, 2000. 6.5.4

[GAH74] Gary Gygax, Dave Arneson, and TSR Hobbies. *Dungeons and dragons*, volume 19. Tactical Studies Rules Lake Geneva, WI, 1974. 5.2

[Gay13] Steve Gaynor. *Gone Home*. Fullbright, 2013. 1.1

[GBB$^+$06] Deepak Garg, Lujo Bauer, Kevin D Bowers, Frank Pfenning, and Michael K Reiter. A linear logic of authorization and knowledge. In *Computer Security–ESORICS 2006*, pages 297–312. Springer, 2006. 2.6.1, 3.6, 7.2.1

[Gen35] Gerhard Gentzen. Untersuchungen über das logische schließen. i. *Mathematische zeitschrift*, 39(1):176–210, 1935. 2.3

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987. `doi:10.1016/0304-3975(87)90045-4`. 1.4.2, 2.1

[Gir95] Jean-Yves Girard. Linear logic: its syntax and semantics. *London Mathematical Society Lecture Note Series*, pages 1–42, 1995. 2.3

[GL87] Jean-Yves Girard and Yves Lafont. Linear logic and lazy computation. In *TAPSOFT'87*, pages 52–66. Springer, 1987. 2.3

[GPSL03] Stefan Greuter, Jeremy Parker, Nigel Stewart, and Geoff Leach. Undiscovered worlds–towards a framework for real-time procedural world generation. In *Fifth International Digital Arts and Culture Conference, Melbourne, Australia*, 2003. 5.4.5

[Hay71] Patrick J Hayes. *The Frame Problem and Related Problems on Artificial Intelligence*. Stanford University, 1971. 1

[Her03] David Herman. *Narrative theory and the cognitive sciences*. Number 158.

Center for the Study of Language and Inf, 2003. 2.1

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993. URL: `http://doi.acm.org/10.1145/138027.138060`, `doi:10.1145/138027.138060`. 1.4.4

[Hig10] T Higgins. Unity-3d game engine. *Accessed December*, 1:2014, 2010. 4.1

[HLZ04] Robin Hunicke, Marc LeBlanc, and Robert Zubek. Mda: A formal approach to game design and game research. In *Proceedings of the AAAI Workshop on Challenges in Game AI*, volume 4, 2004. 1

[HP96] Joshua S Hodas and Jeffrey Polakow. Forum as a logic programming language: preliminary report. *Electronic Notes in Theoretical Computer Science*, 3:196–207, 1996. 3.6

[HPW96] James Harland, David Pym, and Michael Winikoff. Programming in lygon: An overview. In *Algebraic Methodology and Software Technology*, pages 391–405. Springer, 1996. 3.6

[HPW00] James A Harland, David J Pym, and Michael Winikoff. Forward and backward chaining in linear logic. *Electronic Notes in Theoretical Computer Science*, 37:1–16, 2000. 3.1

[Hui14] Johan Huizinga. *Homo Ludens Ils 86*. Routledge, 2014. 1

[HW04] James Harland and Michael Winikoff. Agents via mixed-mode computation in linear logic. *Annals of Mathematics and Artificial Intelligence*, 42:167–196, 2004. 3.6

[HWF10] Pat Harrigan and Noah Wardrip-Fruin. *Second person: Role-playing and story in games and playable media*. The MIT Press, 2010. 1.1

[HZDR11] Ken Hartsook, Alexander Zook, Sauvik Das, and Mark O Riedl. Toward supporting stories with procedurally generated game worlds. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 297–304. IEEE, 2011. 4.1

[JW06] Limin Jia and David Walker. ILC: A foundation for automated reasoning about pointer programs. In *Programming Languages and Systems*, pages 131–145. Springer, 2006. 6.2.3

[KA08] Chris Klimas and Leon Arnott. Twine: a tool for creating interactive stories. `http://www.twinery.org`, 2008. Accessed: 2015-05-31. 1, 1.1, 4.1

[Kan92] Max Kanovich. Horn programming in linear logic is np-complete. In *Logic in Computer Science, 1992. LICS'92., Proceedings of the Seventh Annual IEEE Symposium on*, pages 200–210. IEEE, 1992. 6.5.1

[Kan95] Max I Kanovich. Petri nets, horn programs, linear logic and vector games. *Annals of Pure and Applied Logic*, 75(1):107–135, 1995. 2.6.2, 6.5.1, 6.5.3, 6.5.9

[KM69] Richard M Karp and Raymond E Miller. Parallel program schemata. *Journal of Computer and system Sciences*, 3(2):147–195, 1969. 6.5

[Kow74] Robert Kowalski. Predicate logic as programming language. In *IFIP congress*, volume 74, pages 569–544, 1974. 6.5.1

[Kri89] John Krizanc. *Tamara: a play*. London: Methuen Drama, 1989. 5, 5.5

[KS89] Robert Kowalski and Marek Sergot. A logic-based calculus of events. In *Foundations of knowledge base management*, pages 23–55. Springer, 1989. 1.4.2, 2.3

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994. 1.4.2, 2.3

[Lav13] Stephen Lavelle. PuzzleScript game engine. `http://www.puzzlescript.net`, 2013. Accessed: 2015-05-31. 4.1, 4.1

[LD99] Bernd Lintermann and Oliver Deussen. Interactive modeling of plants. *Computer Graphics and Applications, IEEE*, 19(1):56–65, 1999. 5.4.5

[Ler10] Jérôme Leroux. The general vector addition system reachability problem by presburger inductive invariants. *Logical Methods in Computer Science*, 6(3), 2010. URL: `http://dx.doi.org/10.2168/LMCS-6(3:22)2010`, `doi:10.2168/LMCS-6(3:22)2010`. 6.5.3

[Ler13] Jérôme Leroux. Presburger vector addition systems. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 23–32, 2013. URL: `http://doi.ieeecomputersociety.org/10.1109/LICS.2013.7`, `doi:10.1109/LICS.2013.7`. 6.5.4, 6.5.6

[LPPW05] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 35–46. ACM, 2005. 3.6

[Mac11] Matthew B. MacLaurin. The design of kodu: A tiny visual programming language for children on the xbox 360. *SIGPLAN Not.*, 46(1):241–246, January 2011. URL: `http://doi.acm.org/10.1145/1925844.1926413`, `doi:10.1145/1925844.1926413`. 4.1

[Mas93] M. Masseron. Generating plans in Linear Logic: II. A geometry of conjunctive actions. *Theoretical Computer Science*, 113(2):371–375, 1993. 2.6.1

[Mat99] Michael Mateas. *An Oz-centric review of interactive drama and believable agents*. Springer, 1999. 3.6

[May84] Ernst W Mayr. An algorithm for the general petri net reachability problem. *SIAM Journal on computing*, 13(3):441–460, 1984. 2.6.2

[MBFC13] 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. 2.6.2, 3.7

[Mee76] James Richard Meehan. The metanovel: writing stories by computer. Technical report, DTIC Document, 1976. 2.1, 3.6

[Mes90] José Meseguer. *Rewriting as a unified model of concurrency*. Springer, 1990. A

[MFB14] Chris Martens, Joao F Ferreira, and Anne-Gwenn Bosser. Generative story worlds as linear logic programs. In *Intelligent Narrative Technologies (INT) 7*, 2014. 3.7

[MH69] John McCarthy and Patrick J Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Readings in artificial intelligence*, pages 431–450, 1969. 1.4.2, 2.3

[Mil96] Dale Miller. Forum: A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, 1996. 3.6

[Mil97] Robin Milner. *The definition of standard ML: revised*. MIT press, 1997. 4.7

[ML96] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic journal of philosophical logic*, 1(1):11–60, 1996. 2.3

[Mon05] Nick Montfort. *Twisty Little Passages: an approach to interactive fiction*. Mit Press, 2005. 1.1, 4.6.3

[Mot86] Warren F Motte. *Oulipo: a primer of potential literature*. University of Nebraska Press Lincoln & London, 1986. 1.1

[MS99] Michael Mateas and Phoebe Sengers. Narrative intelligence. In *Proceedings AAAI Fall Symposium on Narrative Intelligence*, pages 1–10, 1999. 2.1

[MS03] Michael Mateas and Andrew Stern. Façade: An experiment in building a fully-realized interactive drama. In *Game Developers Conference*, pages 4–8, 2003. 1.2, 3.6, 4.1

[MTS$^+$10] Josh McCoy, Mike Treanor, Ben Samuel, Brandon Tearse, Michael Mateas, and Noah Wardrip-Fruin. Comme il faut 2: A fully realized model for socially-oriented gameplay. In *Proceedings of the Intelligent Narrative Technologies III Workshop*, page 10. ACM, 2010. 1.2, 2.5, 4.1

[MTS$^+$11] Josh McCoy, Mike Treanor, Ben Samuel, Michael Mateas, and Noah Wardrip-Fruin. Prom week: social physics as gameplay. In *Proceedings of the 6th International Conference on Foundations of Digital Games*, pages 319–321. ACM, 2011. 2.5

[Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989. 2.6.2, 6.5.1

[MW00] Bride Mallon and Brian Webb. Structure, causality, visibility and interaction: propositions for evaluating engagement in narrative multimedia. *International Journal of Human-Computer Studies*, 53(2):269–287, 2000. 2.1

[MWF09] Michael Mateas and Noah Wardrip-Fruin. Defining operational logics. *Digital Games Research Association (DiGRA)*, 2009. 4.1

[Nel01] Graham Nelson. *The Inform Designer's Manual*. Placet Solutions, 2001. 1, 4.1, 4.1

[Nil14] Nils J Nilsson. *Principles of artificial intelligence*. Morgan Kaufmann, 2014. 6.1.1

[Nin98] EAD Nintendo. The legend of zelda: Ocarina of time. *Nintendo. Video game published for the Nintendo*, 64, 1998. 6.6.1

[Nol00] Christopher Nolan. Memento [motion picture]. *United States: Newmarket Films*, 2000. 2.2

[Orw00] Jonathan Orwant. Eggg: Automated programming for game generation. *IBM Systems Journal*, 39(3.4):782–794, 2000. 7.2.3

[PC09] Julie Porteous and Marc Cavazza. Controlling narrative generation with planning trajectories: the role of constraints. In *Interactive Storytelling*, pages 234–245. Springer, 2009. 3.6

[Pep11] Erik Pepenburg. Stage is set. ready for your part? March 2011. URL: `http://www.nytimes.com/2011/03/20/theater/sleep-no-more-from-punchdrunk-transforms-chelsea-warehouses.html`. 5.5.4

[Pet66] Carl Adam Petri. *Communication with automata*. PhD thesis, Darmstadt Technical University, July 1966. 6.5

[Pet77] James L Peterson. Petri nets. *ACM Computing Surveys (CSUR)*, 9(3):223–252, 1977. 6.5.3

[PH13] Przemyslaw Prusinkiewicz and James Hanan. *Lindenmayer systems, fractals, and plants*, volume 79. Springer Science & Business Media, 2013. 7.2.2

[Pot05] Neill Potts. Character interiority: Space, point of view and performance in hitchcock's vertigo [1958]. *Style and Meaning: Studies in the Detailed Analysis of Film*, pages 85–97, 2005. 1.2

[Pro10] Vladimir Propp. *Morphology of the Folktale*, volume 9. University of Texas Press, 2010. 2.2

[Pru86] Przemyslaw Prusinkiewicz. Graphical applications of L-systems. In *Proceedings of graphics interface*, volume 86, pages 247–253, 1986. 5.4.5

[PS98] Frank Pfenning and Carsten Schuermann. Twelf users guide. Technical report, version 1.2. Technical Report CMU-CS-98-173, Carnegie Mellon University, 1998. 1.4.4, 6.1.1

[RFW09] Whitman Richards, Mark Alan Finlayson, and Patrick Henry Winston. Advancing computational models of narrative. *MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, Tech. Rep*, 63:2009, 2009. 2.1

[RGWFM14] Aaron A Reed, Jacob Garbe, Noah Wardrip-Fruin, and Michael Mateas. Ice-bound: Combining richly-realized story with expressive gameplay. *Foundations of Digital Games. FDG*, 2014. 2.7

[RY04] Mark Owen Riedl and R Michael Young. An intent-driven planner for multi-agent story generation. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 186–193. IEEE Computer Society, 2004. 2.4.4, 3.6

[RY10] Mark O Riedl and Robert Michael Young. Narrative planning: Balancing plot and character. *Journal of Artificial Intelligence Research*, 39(1):217–268, 2010. 1.4.2, 2.6.1, 3.6, 5.1.4

[RY13] Justus Robertson and Robert Michael Young. Modelling character knowledge in plan-based interactive narrative to extend accomodative mediation. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013. 7.2.1

[SCS10] István Szita, Guillaume Chaslot, and Pieter Spronck. Monte-carlo tree search in settlers of catan. In *Advances in Computer Games*, pages 21–32. Springer, 2010. 5.3

[Sho00] Emily Short. Galatea. *Electronic Literature Collection: Volume One*, 2000. 1.2

[Sim12] Robert J. Simmons. *Substructural Logical Specifications*. PhD thesis, Carnegie Mellon University, 2012. 6.3.2, 6.4, 6.4.2, 6

[SM11] Adam M Smith and Michael Mateas. Answer set programming for procedural content generation: A design space approach. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):187–200, 2011. 6.1, 7.2.4

[Smi12] Adam M. Smith. *Mechanizing Exploratory Game Design*. PhD thesis, University of California, Santa Cruz, December 2012. 4.8

[SN09] Anders Schack-Nielsen. Redesigning the CLF type theory. 2009. 4

[SNM09] Adam M Smith, Mark J Nelson, and Michael Mateas. Computational support for play testing game sketches. In *AIIDE*, 2009. 4.8

[SNS08] 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. 3.1

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002. 6.2.3

[ST77] George S. Sacerdote and Richard L. Tenney. The decidability of the reachability problem for vector addition systems (preliminary version). In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, pages 61–76, New York, NY, USA, 1977. ACM. URL: `http://doi.acm.org/10.1145/800105.803396`, `doi:10.1145/800105.803396`. 6.5.3

[ST12] Robert J Simmons and Bernardo Toninho. Constructive provability logic. *arXiv preprint arXiv:1205.6402*, 2012. 5.6.1

[Sta84] Ryan Stansifer. Presburgers article on integer airthmetic: Remarks and translation. Technical Report TR84-639, Cornell University, Computer Science Department, September 1984. URL: `http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cs/TR84-639`. 6.5

[Sto10] Kathryn T Stolee. Kodu language and grammar specification. *Microsoft Research whitepaper, Retrieved September*, 1, 2010. 4.6.1

[Stu11] Bethesda Game Studios. *The Elder Scrolls V: Skyrim*. Bethesda Softworks, 2011. 1.1

[SW10] Gillian Smith and Jim Whitehead. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, page 4. ACM, 2010. 4.9, 7.2.2

[SWM10] Gillian Smith, Jim Whitehead, and Michael Mateas. Tanagra: A mixed-initiative level design tool. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, pages 209–216. ACM, 2010. 7.2.3

[TBMB12] Mike Treanor, Bryan Blackford, Michael Mateas, and Ian Bogost. Game-o-matic: Generating videogames that represent ideas. In *Proceedings of the The third workshop on Procedural Content Generation in Games*, page 11. ACM, 2012. 7.2.1

[Teu07] Klaus Teuber. The Settlers of Catan game rules and almanac, 2007. URL: `http://www.catan.com/en/download/?SoC_rv_Rules_091907.pdf`. 5.3, 5.3.1, 5.3.1

[THT12] Luke Trodd, James Harland, and John Thangarajah. Agent deliberation via forward and backward chaining in Linear Logic. In *AAMAS*, pages 1443–1444, 2012. 3.1

[TS85] Tom Trabasso and Linda L Sperry. Causal relatedness and importance of story events. *Journal of Memory and language*, 24(5):595–611, 1985. 2.1

[TZE$^+$15] Mike Treanor, Alexander Zook, Mirjam P Eladhari, Julian Togelius, Gillian Smith, Michael Cook, Tommy Thompson, Brian Magerko, John Levine, and Adam Smith. Ai-based game design patterns. In *Proceedings of the 10 International Conference on Foundations of Digital Games*, FDG 2015. Society for the Advancement of the Science of Digital Games, 2015. 1.4.3, 5.6.3

[Ver02] W van der Hoek Rineke Verbrugge. Epistemic logic: A survey. *Game theory and applications*, 8:53, 2002. 7.2.1

[VGN04] Liliana Vega, Stefan Grünvogel, and Stéphane Natkin. A new methodology for spatiotemporal game design. In *Mehdi, Q. and Gough, N.,(Eds.). Proceedings of CGAIDE 2004, 5th Game-On International Conference on Computer Games: Artificial Intelligence, Design and Education*, pages 109–113. Citeseer, 2004. 2.6.2

[VRD14] Riemer Van Rozen and Joris Dormans. Adapting game mechanics with micro-machinations. In *Foundations of Digital Games*. Society for the Advancement of the Science of Digital Games, 2014. 2.6.2

[WCPW03] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The propositional fragment. In *TYPES*, pages 355–377, 2003. 3.1, 3.4, 6.4

[Wri96] Will Wright. *SimCity*. Erbe, 1996. 1.2

[Wri00] Will Wright. The sims. *MAXIS/Electronic Arts*, 2000. 1.2

[You99] R Michael Young. Notes on the use of plan structures in the creation of interactive plot. In *AAAI Fall Symposium on Narrative Intelligence*, pages 164–167, 1999. 2.2, 2.6.1

[You07] R Michael Young. Story and discourse: A bipartite model of narrative generation in virtual worlds. *Interaction Studies*, 8(2):177–208, 2007. 7.2.1

[YPM94] R Michael Young, Martha E Pollack, and Johanna D Moore. Decomposition and causality in partial-order planning. In *AIPS*, pages 188–194, 1994. 2.6.1