Of Ludics & Ludology: Systems of Play as Linear Logic Programs

Chris Martens
Carnegie Mellon University

Presented at NJPLS, May 16, 2014
takeaway:

linear logic* as a tool for game design enables rapid experimentation and structural analysis of a wide range of core mechanics.

*… with various extra-logical additives.
THREE ACTS

I. Game Design Vocabulary
   Example
   Linear logic programming

II. Payoff: Proofs as interaction traces
   Generation & Analysis

III. Promise: Interactivity
     Invariant Checking
ACT I

Setup: destroy assumptions
You are in the living room. There is a doorway to the
strange goblin lettering to the west, which appears to
trophy case, and a large oriental rug in the center of
above the trophy case hangs an elvish sword of great a
A battery-powered brass lantern is on the trophy case.

- Light lamp
- Taken.
- Remove rug
- With a great effort, the rug is moved
- Dusty cover of a closed trap door.
- Opens trap door
- The door reluctantly open to reveal
- Darkness.

You have moved into a dark place.
The trap door crashes shut, and you
It is pitch black. You are likely to

Minecraft
what are (digital) games made of?

graphics + sound + mechanics
movement, enemies, levels, bosses, items, characters, endings, ...
most frameworks’ starting point:

**graphics** + sound + mechanics

**movement**, enemies, levels, bosses, items, characters, endings, ...
languages for programming games:

Unity
Twine
Inform 7
GameMaker, Scratch
StageCast, PuzzleScript
C++?
FRP?
LANGUAGE AFFECTS DESIGN DECISIONS
a better starting point?

graphics + sound + mechanics
movement, enemies, levels, bosses, items, characters, endings, ...
my starting point:

rules

resources
RULES

Rules of play

~

Rules of logic

linear logic
RULES

Rules of play

~

state change through manipulation of resources
RULES

Rules of play

~

state change through manipulation of resources

~

linear logic
Linear Logic

core judgment:
\[ \Gamma; \Delta \vdash A \]

\( A \text{ persistent} \in \Gamma \)
\( A \text{ linear} \in \Delta \)
Linear Logic

core judgment:
\[ \Gamma; \Delta \vdash A \]

\( A \in \Gamma \): subject to wk, contr, exchg
\( A \in \Delta \): “use exactly once”
Linear Logic

A \to B
A \times B
!A
1
Linear Logic Programming

fill a signature with predicate declarations
pred <arg_types>

and constant declarations
c : A
Linear Logic Programming

fill a signature with predicate declarations
pred <arg_types>

and constant declarations
r : B -o C
EXAMPLE
2d, turn-based puzzle games

http://www.puzzlescript.net
Hardcoded assumptions:

- turn-based
- 2d grid of adjacent locations
- player controls one entity
- controls are up, down, left, right, x
- single player
e.g. Sokoban
in PuzzleScript:

[ > Player | Crate ] -> [ > Player | > Crate ]
My assumptions

turn-based

2d grid of adjacent locations
player controls one entity
controls are up, down, left, right, x
single player
sokoban in linear logic

@  •  □  ➔  □  •  @

@  □  ➔  □  @
sokoban in linear logic

@ 🔴 □ ➔ □ 🔴 @

@ □ ➔ @ □
sokoban in linear logic

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

sokoban in linear logic

\textbf{push} :
\begin{align*}
\text{loc pusher } L & * \ \text{in\_dir } L \ \text{Dir } L' * \ \text{loc block } L' \\
* \ \text{in\_dir } L' \ \text{Dir } L'' & * \ \text{empty } L'' \\
- & o \ \{\text{empty } L & * \ \text{loc pusher } L' * \ \text{loc block } L''\}.
\end{align*}

\textbf{move} :
\begin{align*}
\text{loc pusher } L & * \ \text{in\_dir } L \ \text{Dir } L' * \ \text{empty } L' \\
- & o \ \{\text{empty } L & * \ \text{loc pusher } L'\}.
\end{align*}
EXECUTABLE SPECS AS LINEAR LOGIC PROGRAMS

1. specify the predicates needed to track state, e.g.
   \texttt{loc <entity> <location>}

2. codify state transitions as linear implications
   \texttt{A \rightarrow \{B\}}

3. specify a query: initial state and expected final state
A -o \{B\} induces a transition:

$$\forall \Delta, \Delta, A \rightarrow \Delta, B$$
SEMANTICS OF LINEAR LOGIC PROGRAMS

the \{curly braces\} mean “forward chaining” proof search (lax modality/monad)
SEMANTICS OF LINEAR LOGIC PROGRAMS

A -o \{B\} induces a transition:

\[
\forall \Delta, \ \Delta, \ A \rightarrow \Delta, \ B
\]

meaning this is admissible:

\[
\Delta, \ B \rightarrow * \\
\Delta, \ A \rightarrow *
\]
COMMITTED CHOICE

when there are multiple choices available,
pick one and commit
(NO BACKTRACKING)
these ideas are implemented in frameworks like Celf, LolliMon, Lygon

(I use Celf)
ACT II

Payoff: More interesting examples, Proofs & Analysis
GOAL: interactive fiction with complex character interaction.
shakespearean tragedy world

state components:
character location, possession,
**sentiment** toward other characters,
goals
shakespearean tragedy world

at <character> <location>
has <character> <object>
anger <character> <character>
philia <character> <character>
depressed <character>
shakespearean tragedy world

!dead <character>
!killed <character> <character>
do/insult :
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/compliment :
at C L * at C’ L * philia C C’
-o {at C L * at C’ L *
    philia C C’ * philia C’ C}.
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/mourn :
at C L * philia C C' * dead C'
-o {philia C C' * at C L *
    depressed C * depressed C}. 
do/becomeSuicidal :
 at C L * depressed C * depressed C * depressed C * depressed C
 -o {at C L * suicidal C * 
    wants C weapon}. 
do/loot

: at C L * dead C' * has C' O * wants C O
-o {at C L * has C O}. 
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}. 
initial state
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 *
...

... *
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}. 
final state
ending_happy : nonfinal *
actor C * actor C' *
at C L * at C' L * married C C' -o {final}.

ending_vengeance : nonfinal *
actor C1 * actor C2 * actor C3 *
killed C1 C2 * philia C3 C2 * killed C3 C1 -o {final}. 
proofs as stories
proof of
init -o {final}

\[ \lambda x: \text{init}. \]

let \([xs] = r [ys]\) in ... end
X78 : at mercutio L
X85 : at romeo L
X86 : suicidal romeo
X81 : philia mercutio romeo
X83 : philia romeo mercutio

X88 : at mercutio L
X89 : at romeo L
X90 : philia mercutio romeo
X91 : philia romeo mercutio
X92 : philia romeo mercutio
concurrent equality

\[
\text{let } x_1 = \text{M1 in let } x_2 = \text{M2 in M} \\
\sim \\
\text{let } x_2 = \text{M2 in let } x_1 = \text{M2 in M}
\]

iff the inputs of M2 are separate from the outputs of M1.
... 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, X72]]]]} = 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 
...
graphical representation of traces
queries on sets of traces
> exists ending_1

> exists do/thinkVengefully &&
~link do/thinkVengefully do/murder
Martens, Ferreira, Bosser
“Generative Story Worlds as Linear Logic Programs”
accepted to INT 2014
ACT III

Promise: Interactivity;
Invariant Checking
sokoban, reprise

interactivity, version 1:
at choice points (multiple rules apply), present all available options to player

move
Dir = up, down;
L = ..., L' = ...

push
Dir = right
L, L', L''..
PROBLEM: not all parts of the program should be manipulable by the player
interactivity, version 2: give a language of interaction
dir : type.
  u, d, l, r : dir.

act : type.
move <dir> : act.

-- new piece of state:
  action <act>
augment rules w/extra premise:

push :

\[ \text{action (move } \text{ Dir} \text{)} * \]
\[ \text{loc pusher } L * \text{ in\_dir } L \text{ Dir } L' \]
\[ * \text{ loc block } L'* \text{ in\_dir } L' \text{ Dir } L'' \]
\[ * \text{ empty } L'' \]
\[ -o \{ \text{empty } L * \text{ loc pusher } L' * \text{ loc block } L'' \} \].

move :

\[ \text{action (move } \text{ Dir} \text{)} * \]
\[ \text{loc pusher } L * \text{ in\_dir } L \text{ Dir } L' * \text{ empty } L' \]
\[ -o \{ \text{empty } L * \text{ loc pusher } L' \} \].
but when to introduce “action A”?
PHASES
Phases

Block-delimited subsignatures

phase **world** = {
    rule1 : current Action * ... -o {...}.
    rule2 : current Action * ... -o {...}.
}

phase **player** = {
    rule : player_turn -o {...}
}

Phases

Connected by specification of quiescence behavior

phase world = {...}

phase player = {...}

quiesced world -o
   {player_turn * phase player}.

quiesced player -o {phase world}.
Phases

...are block-delimited subsignatures connected by specifications of quiescence behavior.

\texttt{quiesced P \ast State \to \{phase P' \ast State\}.}

arbitrarily many phases
looping + branching
rock * paper -o {paper}.
paper * scissors -o {scissors}.
scissors * rock -o {rock}.

rock * rock_count N -o {rock_count N+1}.
paper * paper_count N -o {paper_count N+1}.
scissors * scissors_count N -o {scissors_count N+1}.

init -o {rock_count 0 * paper_count 0 * scissors_count 0
  * rock * rock * rock * paper * paper * scissors}.
rock * paper -o {paper}.
paper * scissors -o {scissors}.
scissors * rock -o {rock}.

rock * rock_count N -o {rock_count N+1}.
paper * paper_count N -o {paper_count N+1}.
scissors * scissors_count N -o {scissors_count N+1}.

init -o {rock_count 0 * paper_count 0 * scissors_count 0
* rock * rock * rock * paper * paper * scissors}.
rock * paper -o {paper}.
paper * scissors -o {scissors}.
scissors * rock -o {rock}.

rock * rock_count N -o {rock_count N+1}.
paper * paper_count N -o {paper_count N+1}.
scissors * scissors_count N -o {scissors_count N+1}.

init -o {rock_count 0 * paper_count 0 * scissors_count 0
* rock * rock * rock * paper * paper * scissors}. 
phase \textbf{rps} = 
\{ 
    \text{rock} * \text{paper} \rightarrow \{\text{paper}\}.
    \text{paper} * \text{scissors} \rightarrow \{\text{scissors}\}.
    \text{scissors} * \text{rock} \rightarrow \{\text{rock}\}.

    \text{init} \rightarrow \{\text{rock} * \text{rock} * \text{rock} * \text{paper} * \text{paper} * \text{scissors}\}.
\}

phase \textbf{count} = 
\{ 
    \text{init} \rightarrow \{\text{rock\_count} 0 * \text{paper\_count} 0 * \text{scissors\_count} 0\}.

    \text{rock} * \text{rock\_count} N \rightarrow \{\text{rock\_count} N+1\}.
    \text{paper} * \text{paper\_count} N \rightarrow \{\text{paper\_count} N+1\}.
    \text{scissors} * \text{scissors\_count} N \rightarrow \{\text{scissors\_count} N+1\}.
\}
phase \texttt{rps} = {
    \texttt{rock} * \texttt{paper} \rightarrow \texttt{paper}.
    \texttt{paper} * \texttt{scissors} \rightarrow \texttt{scissors}.
    \texttt{scissors} * \texttt{rock} \rightarrow \texttt{rock}.
}

\texttt{init} \rightarrow \texttt{rock} * \texttt{rock} * \texttt{rock} * \texttt{paper} * \texttt{paper} * \texttt{scissors}.
}

phase \texttt{count} = {
    \texttt{init} \rightarrow \texttt{rock\_count} 0 * \texttt{paper\_count} 0 * \texttt{scissors\_count} 0.
}

\texttt{rock} * \texttt{rock\_count} N \rightarrow \texttt{rock\_count} N+1.
\texttt{paper} * \texttt{paper\_count} N \rightarrow \texttt{paper\_count} N+1.
\texttt{scissors} * \texttt{scissors\_count} N \rightarrow \texttt{scissors\_count} N+1.
}
\texttt{%% expects: all rock, all paper, or all scissors.}
We can interpret phase-structured programs as programs with higher-order, mixed-chaining rules in Celf.
Compiling Phases

We can interpret phase-structured programs as programs with higher-order, mixed-chaining rules in Celf.
FINALE
Minecraft Production Web
how designs fail
FINALE

takeaway:

*linear logic with phases*
as a DSL for game design enables
*rapid experimentation and structural analysis*
of a wide range of core ludical mechanics.