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.
```

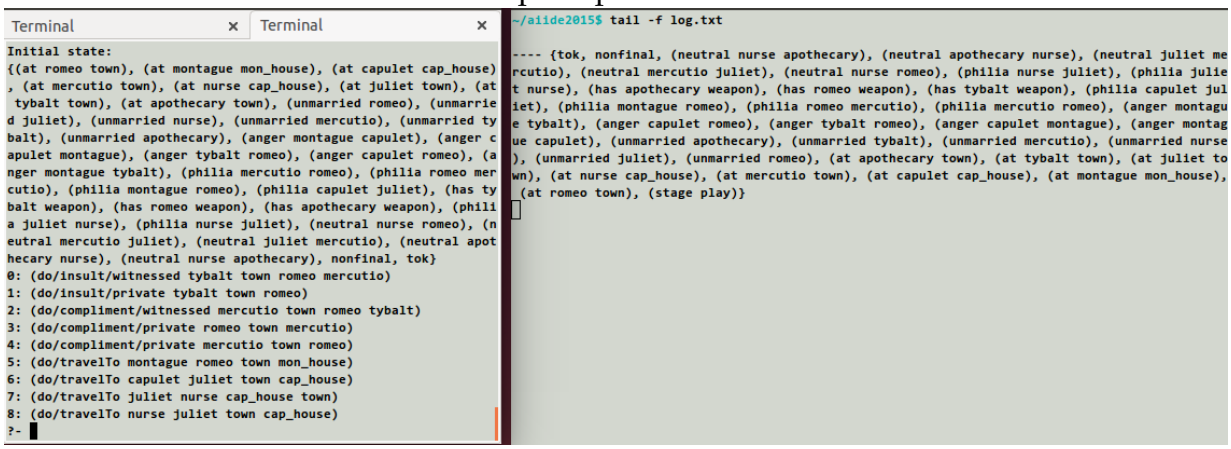
In this formulation of the game, the player may act arbitrarily many times before reactions are calculated (and in fact must do so until quiescence of the act stage). We can slightly modify the program to enjoy a turn-taking idiom instead, offering one turn of input between potentially many steps of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

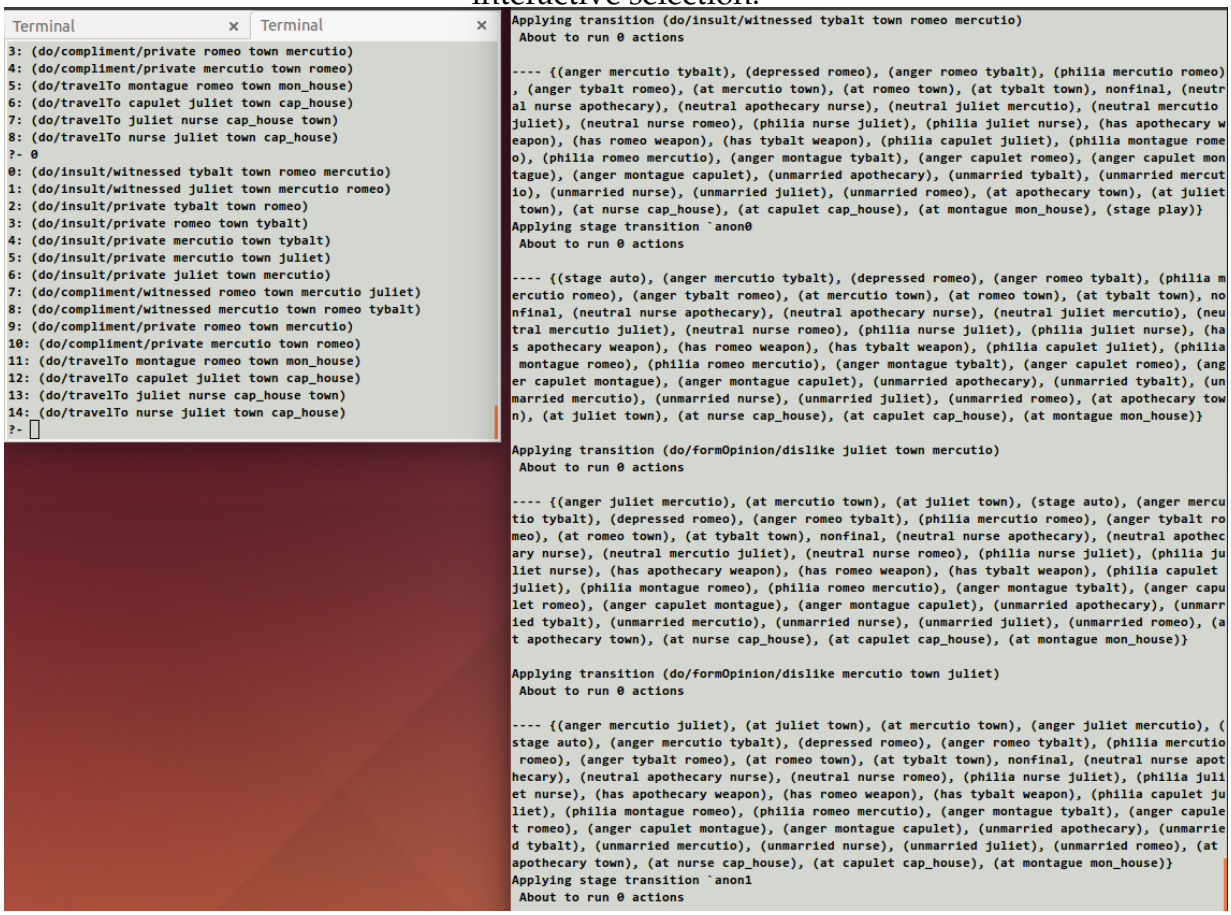
Each rule in the stage is edited to accept one additional premise `tok` (for "token"), which is generated whenever control is transfered to the stage. Turn tokens may be introduced to the reactive side of the program as well, if we wish to carry out a bounded number of reactive steps before returning 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:



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



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 $\Pi c{:}\mathrm{cand}.\Pi n{:}\mathrm{nat}.\mathrm{ballot}\ c \otimes \mathrm{hopeful}\ c\ n \multimap \mathrm{hopeful}\ c\ (s\ n)$.

Stage contents Σ and stage declarations are defined as:

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

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:

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

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

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

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` ϕ'}, 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.

% [...]
```

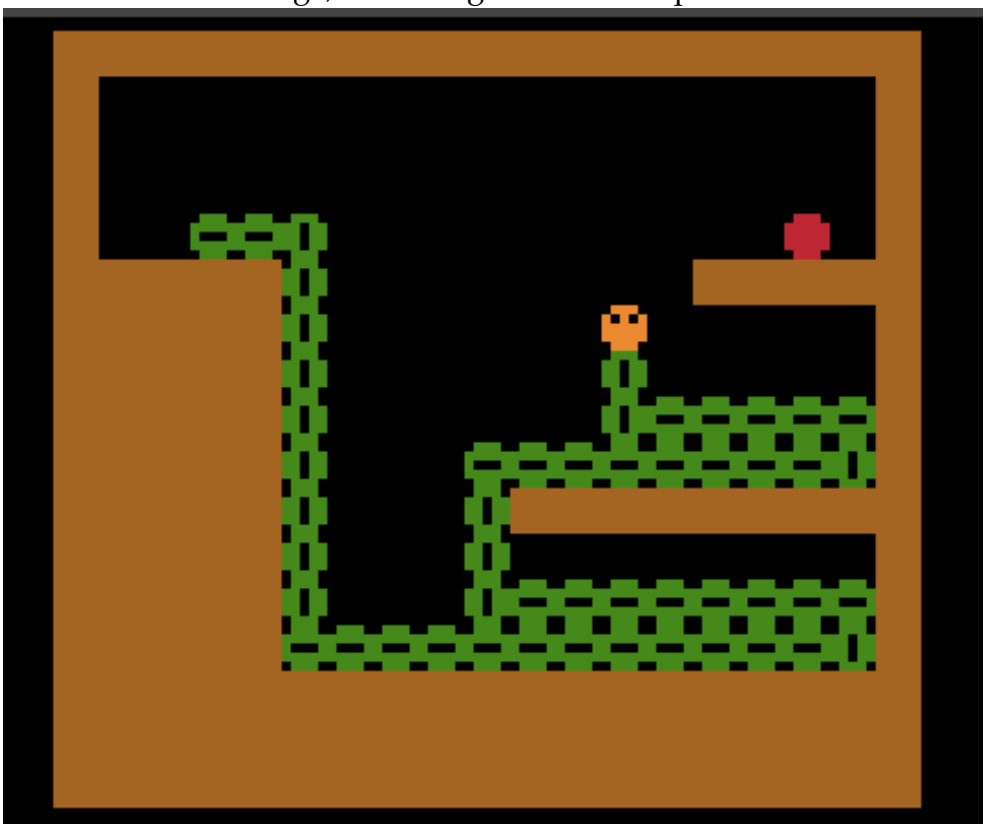
The full code listing for a runnable Sokoban implementation in Ceptre is found in Appendix C.2.

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.

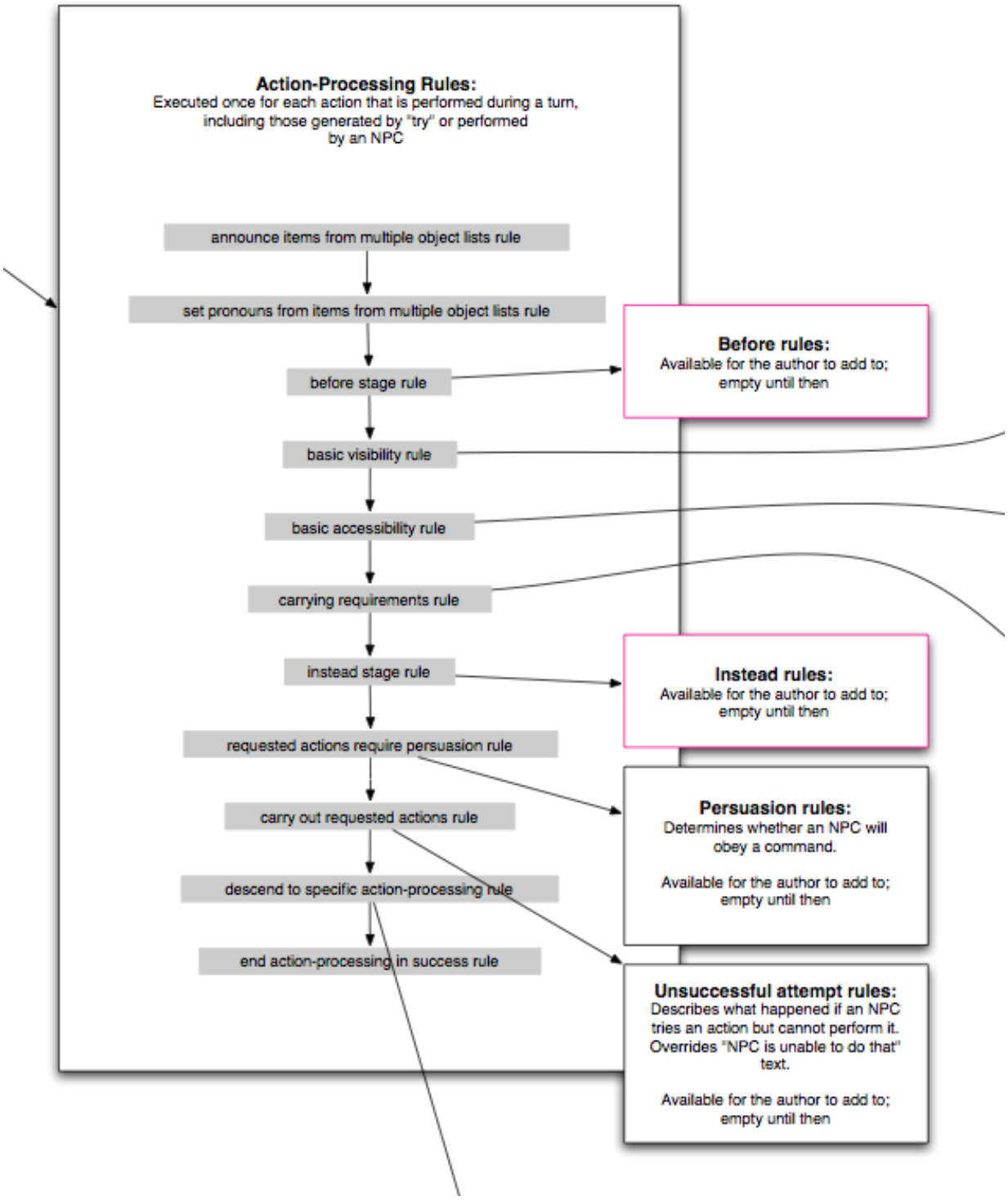
First we will give a type header for the world entities. Aside from verbs, another form of sensing predicate is *visibility*, which could be implemented in terms of the player's location relative to an object, the open or closed states of containers, darkness, and so on. We choose to leave it abstract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



```
            -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.