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: $\Gamma; \Delta \rightarrow \Gamma'; \Delta'$ 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]:

$\otimes L$:

$$(\Gamma; \Delta, A \otimes B) \rightarrow (\Gamma; \Delta, A, B)$$

$!L$:

$$(\Gamma; \Delta, !A) \rightarrow (\Gamma, A; \Delta)$$

copy:

$$(\Gamma, A; \Delta) \rightarrow (\Gamma, A; \Delta, A)$$

$\multimap L$:

$$\frac{\Gamma; \Delta' \vdash A}{(\Gamma; \Delta, \Delta', A \multimap B) \rightarrow (\Gamma; \Delta, B)}$$

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 $\multimap L'$:

$$(\Gamma; \Delta, a_1, \ldots, a_n, a_1 \otimes \ldots \otimes a_n \multimap B) \rightarrow (\Gamma; \Delta, B)$$

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

$$\Delta, a_1, \ldots, a_n \to \Delta, b_1, \ldots, b_m$$

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.

$$\Sigma \vdash K \ \mathsf{wf}$$

$$\frac{}{\Sigma \vdash \mathsf{type\ wf}}\ \mathsf{wf/type} \qquad \frac{}{\Sigma \vdash \mathsf{pred\ wf}}\ \mathsf{wf/pred} \qquad \frac{}{\Sigma \vdash \mathsf{bwd\ wf}}\ \mathsf{wf/bwd}$$

$$\frac{\Sigma \vdash \tau : \mathsf{type} \quad \Sigma \vdash P\ \mathsf{wf}}{\Sigma \vdash \tau \to P\ \mathsf{wf}}\ \mathsf{wf/arr}$$

(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.
```