Lecture 3: Inversion and Chaining

Disclaimer: In this lecture, we drop the names of the judgments eph *and* pers*, as it is clear form the context, which formula is index how.* Full linear logic contains a rich set of connectives for building formulas. We have discussed the essentials in the previous lectures. For the few examples that we have seen, it seems relatively straightforward to formulate the rules, but searching for proofs is difficult. In general, there can be a lot of non-determinism, for example which rule to apply next and how to instantiate quantifiers. In this lecture we discuss how to remove some of the non-determinism, by defining to technique called inversion and chaining, which play a defining role for focusing [**?**]. In this presentation we follow Frank Pfenning in his lecture notes for a graduate class on linear logic that he taught at Carnegie Mellon University in 2012. Let's look at a first example. Let a and b be propositions and let us try to prove that $\mathsf{a} \otimes \mathsf{b} \multimap \mathsf{b} \otimes \mathsf{a}$.

To do this proof we start bottom up.

$$\dfrac{\cdot; \mathsf{a} \otimes \mathsf{b} \Longrightarrow \mathsf{b} \otimes \mathsf{a}}{\cdot; \cdot \Longrightarrow \mathsf{a} \otimes \mathsf{b} \multimap \mathsf{b} \otimes \mathsf{a}} \multimap \mathsf{R}$$

Now we have a choice. If we pick $\otimes\mathsf{R}$, we make a mistake, because we have to decide if $\mathsf{a} \otimes \mathsf{b}$ should go to the left or right. Therefore, we have to exercise rule $\otimes\mathsf{L}$ first and arrive at a new situation.

$$\dfrac{\dfrac{\cdot; \mathsf{a}, \mathsf{b} \Longrightarrow \mathsf{b} \otimes \mathsf{a}}{\cdot; \mathsf{a} \otimes \mathsf{b} \Longrightarrow \mathsf{b} \otimes \mathsf{a}} \otimes\mathsf{L}}{\cdot; \cdot \Longrightarrow \mathsf{a} \otimes \mathsf{b} \multimap \mathsf{b} \otimes \mathsf{a}} \multimap \mathsf{R}$$

Since both a and b are atomic, there is only way forward.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot; \mathsf{b} \Longrightarrow \mathsf{b}}\,\mathsf{pax} \quad \dfrac{}{\cdot; \mathsf{a} \Longrightarrow \mathsf{a}}\,\mathsf{pax}}{\cdot; \mathsf{a}, \mathsf{b} \Longrightarrow \mathsf{b} \otimes \mathsf{a}} \otimes\mathsf{R}}{\cdot; \mathsf{a} \otimes \mathsf{b} \Longrightarrow \mathsf{b} \otimes \mathsf{a}} \otimes\mathsf{L}}{\cdot; \cdot \Longrightarrow \mathsf{a} \otimes \mathsf{b} \multimap \mathsf{b} \otimes \mathsf{a}} \multimap \mathsf{R}$$

The trick towards reducing redundancy lies with the fact that some rules are invertible. We say that a rule is *invertible*, if we can derive the the premiss from a conclusion. Intuitively, if a rule is invertible, we can never enter a dead end, because we can always apply the inverted rule to get back. The $\otimes\mathsf{L}$ rule is invertible. Assume

$$\begin{array}{c} \mathcal{D} \\ \Gamma; \Delta, A \otimes B \Longrightarrow C \end{array}$$

Then with a clever application of cut, we just finish the derivation.

$$\dfrac{\dfrac{\dfrac{}{\Gamma;\mathsf{A}\Longrightarrow\mathsf{A}}\,\mathsf{ax}\quad\dfrac{}{\Gamma;\mathsf{B}\Longrightarrow\mathsf{B}}\,\mathsf{ax}}{\Gamma;A,B\Longrightarrow A\otimes B}\qquad\begin{array}{c}\mathcal{D}\\ \Gamma;\Delta,A\otimes B\Longrightarrow C\end{array}}{\Gamma;\Delta,A,B\Longrightarrow C}\,\mathsf{cut}_e$$

This means that the left rule for tensor is invertible. in our example, we should always apply it if possible this rule. This raises the question, if the right rule for tensor is also applicable. Let's try. This time, we assume that

$$\begin{array}{c}\mathcal{D}\\ \Gamma;\Delta_1,\Delta_2\Longrightarrow A\otimes B\end{array}$$

from this, we should be able to derive either premiss. Without loss of generality we aim for the left.

$$\dfrac{\begin{array}{c}?\\ \Gamma;\Delta_1\Longrightarrow A\end{array}}{?}$$

Since we do not know anything about the form of A, we cannot apply a right rule. Since we don't know anything about Γ and Δ_1, we cannot apply a left rule. So the only leftover candidate is the cut_e rule. This we can try, but then $\mathcal{D}$ would have to be the left premiss, and this is not possible in the general case, as $\Delta_1,\Delta_2\subset\Delta_1$. We conclude that the right rule is invertible.

Here we have a connective, where the left rule is invertible, but the right rule is not. We summarize the result in form of a lemma.

Lemma 8 $\otimes\mathsf{L}$ *is invertible and* $\otimes\mathsf{R}$ *is not.*

I wonder if this special to the tensor, or perhaps is it a pattern that we can find with the other connectives?

Lemma 9 $\multimap\mathsf{R}$ *is invertible and* $\multimap\mathsf{L}$ *is not.*

Proof: First claim: Let

$$\begin{array}{c}\mathcal{D}\\ \Gamma;\Delta\Longrightarrow A\multimap B\end{array}$$

Now we can show the premiss of the $\multimap\mathsf{R}$ rule as follows:

$$\dfrac{\begin{array}{c}\mathcal{D}\\ \Gamma;\Delta\Longrightarrow A\multimap B\end{array}\qquad\dfrac{\dfrac{}{\Gamma;A\Longrightarrow A}\,ax\quad\dfrac{}{\Gamma;B\Longrightarrow B}\,ax}{\Gamma;A,A\multimap B\Longrightarrow B}}{\Gamma;\Delta,A\Longrightarrow B}\,\mathsf{cut}_e$$

Let's tend to the second claim. Assume that

$$\begin{array}{c}\mathcal{D}\\ \Gamma;\Delta_1,\Delta_2,A\multimap B\Longrightarrow C\end{array}$$

We need to show, for example, that

$$\frac{?}{\Gamma; \Delta_1 \Longrightarrow A}\,?$$

which is as impossible as justifying that $\otimes\mathsf{R}$ is invertible. □

Lemma 10 $1\mathsf{L}$ *is invertible and* $1\mathsf{R}$ *is not.*

Proof: Assume

$$\begin{array}{c}\mathcal{D}\\ \Gamma; \Delta, 1 \vdash C\end{array}$$

It is easy to convince ourselves of the invertibility of this rule:

$$\frac{\dfrac{}{\Gamma; \cdot \Longrightarrow \mathbf{1}}\,ax \qquad \begin{array}{c}\mathcal{D}\\ \Gamma; \Delta, \mathbf{1} \vdash C\end{array}}{\Gamma; \Delta \vdash C}\,\mathsf{cut}_e$$

□

For completeness, we'll look the remaining two connectives, ! and ∀.

Lemma 11 $!\mathsf{L}$ *is invertible and* $!\mathsf{R}$ *is not.*

Proof: Assume

$$\begin{array}{c}\mathcal{D}\\ \Gamma; (\Delta, !A) \vdash C\end{array}$$

It is easy to convince ourselves of the invertibility of this rule:

$$\frac{\dfrac{\dfrac{\dfrac{}{(\Gamma, A); A \Longrightarrow A}\,ax}{(\Gamma, A); \cdot \Longrightarrow A}\,copy}{(\Gamma, A); \cdot \Longrightarrow\, !A}\,!\mathsf{R} \qquad \begin{array}{c}\mathcal{D}\\ \Gamma; (\Delta, !A) \vdash C\end{array}}{\Gamma, A; \Delta \vdash C}\,\mathsf{cut}_p$$

$!\mathsf{L}$ is not invertible, although it might look at first glance that it is. □

Lemma 12 $\forall\mathsf{R}$ *is invertible and* $\forall\mathsf{L}$ *is not.*

Proof: We only show the first claim. Assume

$$\begin{array}{c}\mathcal{D}\\ \Gamma; \Delta \vdash \forall x : \tau.\, A\end{array}$$

We can easily show.

$$\frac{\begin{array}{c}\mathcal{D}\\ \Gamma; \Delta \vdash \forall x : \tau.\, A\end{array} \qquad \dfrac{\dfrac{}{\Gamma; A[a/x] \Longrightarrow A[a/x]}\,ax}{\Gamma; \forall x : \tau.\, A \Longrightarrow A[a/x]}\,\forall\mathsf{L}}{\Gamma, A; \Delta \vdash A[a/x]}\,\mathsf{cut}_e$$

$\forall\mathsf{R}$ is not invertible. □

Now, after cycling through all five connectives, we notice that for some the left rules are invertible, for others the right rules. This observation allows us to classify formulas into two classes, $\multimap$ and $\forall$ are called negative (or asynchronous) connectives, $!, \mathbf{1}, and \otimes$ are called positive (or synchronous connectives) [**?**]. Nothing has been said yet about the atoms, which for now may be negative P^- or positive P^+.

$$\begin{array}{llrcl} \text{Negative Formulas} & A^-, B^- & ::= & P^- \mid \forall x{:}\tau.\, A \mid A \multimap B \\ \text{Positive Formulas} & A^+, B^+ & ::= & P^+ \mid A \otimes B \mid \mathbf{1} \mid !A \\ \text{Formulas} & & A & ::= & A^- \mid A^+ \end{array}$$

The fragment includes atomic formulas P^-, universal quantification $\forall x{:}\tau.\, A^-$, linear implication $A^+ \multimap \{B^+\}$, simultaneous conjunction $A^+ \otimes B^+$ and its unit $\mathbf{1}$, the unrestricted modality $!A^-$, and an inclusion, A^-, of negative formulas into positive formulas.

This means, that we can apply invertible rules eagerly. For a particular theorem proving goal, this might mean, applying several of those invertible rules in sequence, called an *inversion phase* It is interesting to note that such as phase terminates (because with every application of an invertible rule, we loose one connective).

What shall we do when we run out ouf possibilities when we do inversion? The interesting observation is that we may pick one assumption to work on and apply non-invertible rules as eagerly as possible. into so called chains. This might be surprising. It turns out, that you never have to backtrack within one of those chains. Either you need the entire chain to complete, or you don't need to work on the chosen assumption at all. This is one of the insights that is due to Andreoli [**?**]. To make this idea precise, we introduce a focus $[A]$. In our hypothetical judgment $\Gamma; \Delta \Longrightarrow A$ we may have at most one focus. No focus means that we are still inverting, one focus simple singles our that we are in teh middle of chain apply non-invertible rules. Not to confuse things, we write $\Gamma; \delta \longrightarrow \gamma$ for this judgment, where we define

$$\begin{array}{rcl} \delta & ::= & \cdot \mid \delta, A \mid \delta, [A] \\ \gamma & ::= & A \mid [A] \end{array}$$

First, we only consider the fragment without persistant resources. We keep

the Γ;, but we will consider it later.

$$\frac{}{\Gamma; [P^-] \longrightarrow P^-}\, pax^- \qquad \frac{}{\Gamma; P^+ \longrightarrow [P^+]}\, pax^+$$

$$\frac{\Gamma; (\delta, A) \longrightarrow B}{\Gamma; \delta \longrightarrow A \multimap B} \multimap \mathsf{R} \qquad \frac{\Gamma; \Delta_1 \longrightarrow [A] \quad \Gamma; \Delta_2, [B] \longrightarrow C}{\Gamma; (\Delta_1, \Delta_2, [A \multimap B]) \longrightarrow C} \multimap \mathsf{L}$$

$$\frac{\Gamma; \Delta_1 \longrightarrow [A] \quad \Gamma; \Delta_2 \longrightarrow [B]}{\Gamma; \Delta \longrightarrow [A \otimes B]} \otimes \mathsf{R} \qquad \frac{\Gamma; (\delta, A, B) \longrightarrow \gamma}{\Gamma; (\delta, A \otimes B) \longrightarrow \gamma} \otimes \mathsf{L}$$

$$\frac{}{\Gamma; \cdot \longrightarrow \mathbf{1}}\, \mathbf{1}\mathsf{R} \qquad \frac{\Gamma; \delta \longrightarrow C}{\Gamma; (\delta, \mathbf{1}) \longrightarrow C}\, \mathbf{1}\mathsf{L}$$

$$\frac{\Gamma; \delta \longrightarrow A[a/x]}{\Gamma; \delta \longrightarrow \forall x{:}\tau.A} \forall \mathsf{R}^{(}a : \tau)$$

$$\frac{\Gamma; (\Delta, [A[t/x]]) \longrightarrow C}{\Gamma; (\Delta, [\forall x{:}\tau.A]) \longrightarrow C} \forall \mathsf{L}, \text{where } t \text{ has sort } \tau$$

Next, we consider how to enter a chain an how to leave it. There are two rules for entering,

$$\frac{\Gamma; \Delta \longrightarrow [A^+]}{\Gamma; \Delta \longrightarrow A^+} \mathsf{focusR} \qquad \frac{\Gamma; \Delta, [A^-] \longrightarrow C}{\Gamma; \Delta, A^- \longrightarrow C} \mathsf{focusL}$$

and two rules exiting (called blurring), if you read the rules bottom up.

$$\frac{\Gamma; \Delta \longrightarrow A^-}{\Gamma; \Delta \longrightarrow [A^-]} \mathsf{blurR} \qquad \frac{\Gamma; \Delta, A^+ \longrightarrow C}{\Gamma; \Delta, [A^+] \longrightarrow C} \mathsf{focusL}$$

In the last lecture we proved the initiality extension and the admissibility of the cut rule for our logic. By permitting to focus on assumptions, we need to generalize both induction hypothesis of the admissibility theorem, by the following admissible rules. Here the admissible rules initiality expansion

$$\frac{}{\Gamma; A \longrightarrow A}\, id \qquad \frac{}{\Gamma; A \longrightarrow [A]}\, id\mathsf{R} \qquad \frac{}{\Gamma; [A] \longrightarrow A}\, id\mathsf{L}$$

Next, the addmissible cut rules.

$$\frac{\Gamma; \Delta \longrightarrow [A] \quad \Gamma; (\delta, A) \longrightarrow C}{\Gamma; (\Delta, \delta) \longrightarrow C}\, cut\mathsf{L}_e \qquad \frac{\Gamma; \delta \longrightarrow A \quad \Gamma; (\delta, [A]) \longrightarrow C}{\Gamma; (\Delta, \delta) \longrightarrow C}\, cut\mathsf{R}_e$$

$$\frac{\Gamma; \Delta \longrightarrow A^- \quad \Gamma; (\delta, A^-) \longrightarrow C}{\Gamma; (\Delta, \delta) \longrightarrow C}\, cut\mathsf{L}_e \qquad \frac{\Gamma; \delta \longrightarrow A^+ \quad \Gamma; (\delta, A^+) \longrightarrow C}{\Gamma; (\Delta, \delta) \longrightarrow C}\, cut\mathsf{R}_e$$

The main theorem is due to Andreoli, shows that focusing provability. We need to show two directions.

Theorem 13 (Soundness) *Let* $\Gamma; \delta \longrightarrow \gamma$. *If we erase all focusing brackets from* δ *and* γ, *we obtain* Δ *and a formula* C. *Then* $\Gamma; \Delta \Longrightarrow C$.

Proof: Easy induction. Remove focusing and bluring rules. □

Theorem 14 (Completeness) *If* $\Gamma; \Delta \overset{\mathcal{D}}{\Longrightarrow} C$ *then* $\Gamma; \Delta \longrightarrow C$.

Proof: This proof is much more complicated. We show only one case that $\mathcal{D}$ ends in the $\multimap \mathsf{L}$ rule.

$$\dfrac{\Gamma; \Delta_1 \overset{\mathcal{D}_1}{\Longrightarrow} A \qquad \Gamma; (\Delta_2, B) \overset{\mathcal{D}_2}{\Longrightarrow} C}{\Gamma; (\Delta_1, \Delta_2, A \multimap B) \Longrightarrow C} \multimap \mathsf{L}$$

By applying the induction hypothesis on $\mathcal{D}_1$ and $\mathcal{D}_2$, we obtain that $\Gamma; \Delta_1 \longrightarrow A$ and $\Gamma; (\Delta_2, B) \longrightarrow C$. We need to show that $\Gamma; (\Delta_1, \Delta_2, A \multimap B) \longrightarrow C$. Let's start bottom up, and apply the focusing rule focusL.

$$\Gamma; (\Delta_1, \Delta_2, [A \multimap B]) \longrightarrow C$$

This can only be achieved by applying $\multimap \mathsf{L}$, now we only need to show $\Gamma; \Delta_1 \longrightarrow [A]$ and $\Gamma; (\Delta_2, [B]) \longrightarrow C$, which looks pretty much like the result of induction hypothesis from above but not quite, because we cannot just randomly add a focus. We just don't know if A is positive, and B negative. In fact, we would make a mistake as the following counte example shows. Let $\Delta = a^+, a^+ \multimap b^+$ and $A = b^+$. Clearly, $\cdot; (a^+, a^+ \multimap b^+) \Longrightarrow b^+$ is provable, in the unfocused system. But $\cdot; (a^+, a^+ \multimap b^+) \longrightarrow b^+$ is not: Since b^+ is not in Δ, focusing on b^+ in

$$\cdot; (a^+, a^+ \multimap b^+) \longrightarrow [b^+]$$

will fail!

Instead, we will need to do some cutting.

$$\dfrac{\Gamma; \Delta_1 \overset{\mathcal{D}_1}{\longrightarrow} A \qquad \dfrac{\dfrac{\dfrac{}{\Gamma; A \longrightarrow [A]}\, id\mathsf{R} \qquad \dfrac{}{\Gamma; [B] \longrightarrow B}\, id\mathsf{L}}{\Gamma; (A, [A \multimap B]) \longrightarrow B} \multimap \mathsf{L}}{\Gamma; (A, A \multimap B) \longrightarrow B}\, \mathsf{focusL}}{\Gamma; (\Delta_1, A \multimap B) \longrightarrow B}\, \mathsf{cut} \qquad \Gamma; (\Delta_2, B) \overset{\mathcal{D}_2}{\longrightarrow} C}{\Gamma; (\Delta_1, \Delta_2, A \multimap B) \longrightarrow C}\, \mathsf{cut}$$

□

Finally, we address the question of persistant resources. Since last lecture, the derivability judgment refers to Γ the context of presistant resources. We will show below, that there is no need to define a focus for Γ, the only change is that when copying a resources from Γ into Δ, we will put a rule immediately into focus. As a consequence, we will introduce one new axiom rule, because if we are in right focus we must be able to look up for a positive atom in Γ without loosing right focus. This is also why the copy rule focuses only on non-positive atoms.

$$\frac{\Gamma, A; \Delta, [A] \vdash C}{\Gamma, A; \Delta \vdash C}\, copy, A \neq P^+ \qquad \frac{}{\Gamma, P^+; \cdot \longrightarrow [P^+]}\, pax!$$

Two things until we are done. First, we need to give focused versions of the ! left and right rules. This is straightforward since we know that ! is a positive connective.

$$\frac{\Gamma; \cdot \longrightarrow [A]}{\Gamma; \cdot \longrightarrow [!A]}\,!\mathsf{R} \qquad \frac{(\Gamma, A); \delta \longrightarrow \gamma}{\Gamma; (\delta, !A) \longrightarrow \gamma}\,!\mathsf{L}$$

Finally, we just state that the focused version of the persistant fragment of linear logic also permits and admissible cut rule:

$$\frac{\Gamma; \cdot \longrightarrow A \quad (\Gamma, A); \delta \longrightarrow \gamma}{\Gamma; \delta \longrightarrow \gamma}\,\mathsf{cut}!$$

Lecture 4: Celf

In the last lecture, we will map the logic into the CLF type theory. We just motivated the following syntax for postive and negative formulas.

$$\begin{array}{lrcl}
\text{Negative Formulas} & A^-, B^- & ::= & P^- \mid \forall x{:}\tau.\, A \mid A \multimap B \\
\text{Positive Formulas} & A^+, B^+ & ::= & P^+ \mid A \otimes B \mid \mathbf{1} \mid\, !A \\
\text{Formulas} & A & ::= & A^- \mid A^+
\end{array}$$

The first simiplicfication that we do is to commit our choices of polarities. We will not be considering positive atoms.

$$\begin{array}{lrcl}
\text{Negative Formulas} & A^-, B^- & ::= & P^- \mid \forall x{:}\tau.\, A^+ \mid A^+ \multimap B^- \mid\uparrow A^+ \\
\text{Positive Formulas} & A^+, B^+ & ::= & A^+ \otimes B^+ \mid \mathbf{1} \mid\, !A^- \mid\downarrow A^-
\end{array}$$

Now we turn formulas into types, hereby collapsing sorts and types. Negative atoms turn into indexed type families. We observe a missing existential quantifier – we just add a dependent pair. It is easy to check the rules, please check this on your own.

$$\begin{array}{lrcl}
\text{Negative Types} & A^-, B^- & ::= & P^- \mid \Pi x{:}A^+.\, B^- \mid A^+ \multimap B^- \mid\uparrow A^+ \\
\text{Positive Types} & A^+, B^+ & ::= & \exists x{:}A^+.\, B^+ \mid A^+ \otimes B^+ \mid \mathbf{1} \mid\, !A^- \mid\downarrow A^-
\end{array}$$

Finally, we simplify the rules: it is suffient to restrict ourselves to either the uparrow or the downarrow. For CLF, we remove the downarrow (and use curely braces instead of the uparrow). Interestingly, the $\{A^+\}$ forms a monad, with which we can capture concurrency.

$$\begin{array}{lrcl} \text{Negative Types} & A^-, B^- & ::= & P^- \mid \Pi x{:}A^+.\, B^- \mid A^+ \multimap B^- \mid \{A^+\} \\ \text{Positive Types} & A^+, B^+ & ::= & \exists x{:}A^+.\, B^+ \mid A^+ \otimes B^+ \mid \mathbf{1} \mid\, !A^- \mid A^- \end{array}$$

After all of these cosmetic changes, we need to think about objects that inhabit those types. We distinguish between three categories, normal objects N, monadic objects M, expresions E, and patterns ρ. In addition, we can make negative atoms more precise.

$$\begin{array}{lrcl} \text{Negative Atoms} & P^- & ::= & a^-\ N_1 \ldots N_n \\ \text{Negative Types} & A^-, B^- & ::= & P^- \mid \Pi x{:}A^+.\, B^- \mid A^+ \multimap B^- \mid \{A^+\} \\ \text{Positive Types} & A^+, B^+ & ::= & \exists x{:}A^+.\, B^+ \mid A^+ \otimes B^+ \mid \mathbf{1} \mid\, !A^- \mid A^- \end{array}$$

where

$$\begin{array}{lrcl} \text{Normal Objects} & N & ::= & h\ N_1 \ldots N_n \mid \lambda\rho{:}A^+.N \mid \{E\} \\ \text{Expressions} & E & ::= & \mathsf{let}\ \{\rho\} = h\ N_1 \ldots N_n\ \mathsf{in}\ E \mid M \\ \text{Monadic Objects} & M & ::= & \langle M_1, M_2\rangle \mid \mathbf{1} \mid\, !N \mid\!\downarrow N \\ \text{patterns} & \rho & ::= & \langle \rho_1, \rho_2\rangle \mid \mathbf{1} \mid\, !x \mid\!\downarrow x \end{array}$$

One last remark. The way to program in Celf is to define a file of type family declarations and constant declartions. To help the programmer, Celf will always try to infer the types of uppercase variables and implicity Π quantifies them. That's all for the theory folks, let's look at the example of single transferable vote in Celf.

For our specification of STV, we must introduce several predicates, which are summarized in Table 1. The *uncounted-ballot*, *counted-ballot*, *hopeful*, *defeated*, *elected*, *quota*, and *winners* predicates characterize the ballot box, candidates' statuses, and the election's state. The *elect-all*, *defeat-min*, *defeat-min′*, *transfer*, and *begin* predicates are used to indicate progress through the STV algorithm's phases. Finally, *minimum* is an auxiliary predicate used in determining a candidate with the fewest votes. (We again assume the usual ordering predicates on natural numbers, such as $!(N \geq N')$.)

The linear logical axioms that specify STV are given in Fig. 2-4. Several of these axioms pattern-match on the shape of a list of candidates. Following standard convention, we use [ ] to stand for the empty list and $[C \mid L]$ to stand for the non-empty list with head C and tail L. (We again follow the convention that universal quantification is implicit for variables written in upper case.)

These axioms faithfully encode STV in a concise and elegant fashion—rather than requiring hundreds or thousands of lines of imperative source code, our full STV specification fits on a single page! To make plain the close correspondence of the axioms with the natural language description of STV used in current practice, we will now walk through their meanings.

Table 1: Descriptions of predicates used in the STV specification.

Predicate	Meaning
uncounted-ballot(C, L)	An uncounted ballot with highest preference for candidate C and list L of lower preferences.
counted-ballot(C, L)	A ballot counted for candidate C, with list L of lower preferences.
hopeful(C, N)	Candidate C is not yet defeated nor elected, and N ballots have been counted for C thus far.
!*defeated*(C)	Candidate C has been (and will remain) defeated.
!*elected*(C)	Candidate C has been (and will remain) elected.
!*quota*(Q)	Q votes are needed to be elected.
winners(W)	The candidates in list W have been elected thus far.
begin(S, H, U)	Token to signal that the STV algorithm should begin running. There are S seats up for election, H hopeful candidates, and U ballots cast.
count-ballots(S, H, U)	Token to indicate that the algorithm is counting ballots, and that there are S open seats, H hopeful candidates, and U uncounted ballots remaining.
!*elect-all*	Token to indicate that there are more open seats than hopefuls remaining; all remaining hopefuls should become elected.
defeat-min(S, H, M)	Token to indicate that the algorithm is in the first step of determining a candidate who has the fewest votes. There are S open seats, H hopeful candidates, and M potential minimums remaining.
defeat-min$'(S, H, M)$	Token to indicate that the algorithm is in the second step of determining a candidate who has the fewest votes. There are S open seats, H hopeful candidates, and M potential minimums remaining.
minimum(C, N)	Candidate C's vote count of N is a potential minimum.
transfer(C, N, S, H, U)	Token to indicate that newly defeated candidate C's remaining N votes are being transferred. There are S open seats, H hopeful candidates, and U uncounted ballots.

Beginning the STV Algorithm. The begin/1 axiom describes the initial step of the STV algorithm: the Droop quota is computed and recorded. Ballot counting is initiated, with no candidates having been declared winners.

Counting the Ballots.

- count/1 describes counting a ballot that does not cause its candidate, C, to reach the quota: C's vote total increases and ballot counting continues.

- count/2 describes counting a ballot that causes its candidate, C, to finally reach the quota. C becomes elected, being a hopeful no longer, and is added to the list of winners. Any ballots remaining uncounted for C constitute C's vote surplus; the surplus is randomly selected because ballots are counted in a random order. After C is elected, ballots continue to be counted.

- count/3.1, count/3.2, count/4.1, and count/4.2 express that no more ballots are counted for candidates that are already either elected or defeated. The ballots transfer to the next highest preference; if none exists, the ballot is consumed—that is, the vote is wasted.

- Finally, count/5 and count/6 describe what happens when there are no more ballots to count. If there are fewer open seats than hopefuls remaining (count/5), then a candidate with the fewest votes is defeated; the generated *defeat-min* token begins this process. Otherwise, if there are at least as many open seats as hopefuls (count/6), then all remaining hopefuls are elected.

Defeating a Candidate with the Fewest Votes.

- defeat-min/1 labels all hopeful candidates as potential minimums. When there are no more hopefuls to label (i.e., when the H counter reaches 0), defeat-min/2 transitions to the second phase of defeating a candidate.

- defeat-min′/1 and defeat-min′/2 describe a random tournament for finding, among the potential minimums, a candidate with the fewest votes. Candidates not selected as the minimum are restored to their hopeful status (defeat-min′/1). When only one candidate is a potential minimum (i.e., when the M counter reaches 1), that candidate must have the fewest votes; she is defeated and the process of transferring her votes begins (defeat-min′/2).

Transferring a Defeated Candidate's Votes.

- transfer/1 expresses that ballots counted for a newly defeated candidate, C, are returned to the ballot box as uncounted ballots. As transfer/2 shows, when the N counter reaches 0, these ballots will be re-counted. Because

```
count/1 : count-ballots S H (s U) *
          uncounted-ballot C L *
          hopeful C N * !quota Q * !nat-less (s N) Q
            -o {counted-ballot C L *
                hopeful C (s N) *
                count-ballots S H U}.

count/2 : count-ballots (s (s S)) (s H) (s U) *
          uncounted-ballot C L *
          hopeful C N *
          !quota Q * !nat-lesseq Q (s N) * winners W
            -o {counted-ballot C L *
                !elected C *
                winners (cons C W) *
                count-ballots (s S) H U}.

count/3 : count-ballots (s z) H U *
          uncounted-ballot C L *
          hopeful C N * !quota Q *
          !nat-lesseq Q (s N) * winners W
            -o {counted-ballot C L *
                !elected C *
                winners (cons C W) *
                !defeat-all}.

count/4_1 : count-ballots S H U *
            uncounted-ballot C (cons C' L) *
            !elected C
              -o {uncounted-ballot C' L *
                  count-ballots S H U}.

count/4_2 : count-ballots S H U *
            uncounted-ballot C (cons C' L) *
            !defeated C
              -o {uncounted-ballot C' L *
                  count-ballots S H U}.

count/5_1 : count-ballots S H (s U) *
            uncounted-ballot C nil *
            !elected C
              -o {count-ballots S H U}.

count/5_2 : count-ballots S H (s U) *
            uncounted-ballot C nil *
            !defeated C
              -o {count-ballots S H U}.

count/6 : count-ballots S H z
            -o {defeat-min S H z}.
```


Figure 2: Part 1: A linear logical specification of single transferable vote.

```
defeat-min/1 : defeat-min S (s H) M *
               hopeful C N
                 -o {minimum C N *
                     defeat-min S H (s M)}.

defeat-min/2 : defeat-min S z M
                 -o {defeat-min' S z M}.

defeat-min'/1 : defeat-min' S H (s M) *
                minimum C N *
                minimum C' N' *
                !nat-less N N'
                  -o {minimum C N *
                      hopeful C' N' *
                      defeat-min' S (s H) M}.

defeat-min'/2 : defeat-min' S H (s z) *
                minimum C N
                  -o {!defeated C *
                      transfer C N S H z}.

transfer/1 : transfer C (s N) S H U *
             counted-ballot C (cons C' L)
               -o {uncounted-ballot C' L *
                   transfer C N S H (s U)}.

transfer/2 : transfer C (s N) S H U *
             counted-ballot C nil
               -o {transfer C N S H U}.

transfer/3 : transfer C z S H U *
             !nat-less S H
               -o {count-ballots S H U}.

transfer/4 : transfer C z S H U *
             !nat-lesseq H S
               -o {!elect-all}.

defeat-all/1 : !defeat-all *
               hopeful C N
                 -o {!defeated C}.

elect-all/1 : !elect-all *
              hopeful C N *
              winners W
                -o {!elected C *
                    winners (cons C W)}.
```


Figure 3: Part 2: A linear logical specification of single transferable vote.

```
cleanup/1 : !defeat-all *
            uncounted-ballot C L
              -o {1}.

cleanup/2 : !defeat-all *
            counted-ballot C L
              -o {1}.

cleanup/3 : !elect-all *
            uncounted-ballot C L
              -o {1}.

cleanup/4 : !elect-all *
            counted-ballot C L
              -o {1}.

run/1 : run S H U *
        !nat-divmod U (s S) Q _
          -o {!quota (s Q) *
              winners nil *
              count-ballots S H U}.
```

Figure 4: Part 3: A linear logical specification of single transferable vote.

C is now defeated, re-counting these ballots will effectively transfer them to the next highest preference, if one exists (count/3.2 and count/4.2).

Finishing the STV Election.

- elect-all/1 expresses that the STV algorithm finishes by electing all remaining hopefuls. (Note that there may possibly be no remaining hopefuls at this point.) Because this is the last step of the STV algorithm, we may think of this step as continuing forever, idling once all remaining hopefuls have been elected. This justifies the use of the ! modality here and also in count/6.

- When the STV algorithm finishes, the counted ballots will remain as linear resources. The resource discipline of linear logic demands that these be used once. Therefore, the cleanup/1 axiom consumes any remaining ballots. This is safe because the STV algorithm has already filled all seats.