# RECITATION 13: SUBSINGLETON LOGIC

RYAN KAVANAGH

Yesterday, we saw how we could view ordered proofs as concurrent programs. We focused on the subsingleton fragment of ordered logic, namely, the fragment where each judgment has at most one antecedent. We introduced the judgment $\omega \vdash P : A$, which under the proofs-as-programs interpretation, can equivalently be viewed as saying that $P$ is a proof of $A$ using $\omega$, and that $P$ is a process providing $A$ and using $\omega$. Finally, we saw that the computational interpretation comes from cut reduction.

## 1. Rules of Subsingleton Logic

The subsingleton fragment is given by the following rules, where $\omega$ is an ordered context consisting of zero or one antecedents:

$$\frac{A \vdash C \quad B \vdash C}{A \oplus B \vdash C}\ \oplus L \qquad \frac{\omega \vdash A}{\omega \vdash A \oplus B}\ \oplus R_1 \qquad \frac{\omega \vdash B}{\omega \vdash A \oplus B}\ \oplus R_2$$

$$\frac{A \vdash C}{A \,\&\, B \vdash C}\ \& L_1 \qquad \frac{B \vdash C}{A \,\&\, B \vdash C}\ \& L_2 \qquad \frac{\omega \vdash A \quad \omega \vdash B}{\omega \vdash A \,\&\, B}\ \& R$$

$$\frac{}{A \vdash A}\ \mathrm{id}_A \qquad \frac{\omega \vdash A \quad A \vdash C}{\omega \vdash C}\ \mathrm{cut}_A \qquad \frac{\cdot \vdash C}{\mathbf{1} \vdash C}\ \mathbf{1}L \qquad \frac{}{\cdot \vdash \mathbf{1}}\ \mathbf{1}R$$

where we had the following term assignments:

$$\frac{A_i \vdash Q_i : C \quad (\forall i \in I)}{\oplus\{l_i : A_i\}_{i \in I} \vdash \mathsf{caseL}\ (l_i \Rightarrow Q_i)_{i \in I} : C}\ \oplus L \qquad \frac{\omega \vdash P : A_k \quad (k \in I)}{\omega \vdash \mathsf{R}.l_k; P : \oplus\{l_i : A_i\}_{i \in I}}\ \oplus R_k$$

$$\frac{\cdot \vdash C}{\mathbf{1} \vdash \mathsf{waitL} : C}\ \mathbf{1}L \qquad \frac{}{\cdot \vdash \mathsf{closeR} : \mathbf{1}}\ \mathbf{1}R$$

$$\frac{}{A \vdash \leftrightarrow : A}\ \mathrm{id}_A \qquad \frac{\omega \vdash P : A \quad A \vdash Q : C}{\omega \vdash (P \mid Q) : C}\ \mathrm{cut}_A$$

(we didn't assign terms to external choice and they won't be used below, but they are symmetric relative to the $\vdash$). These were united by the following reduction rules:

$$\frac{(\mathsf{R}.l_k; P) \mid \mathsf{caseL}\ (l_i \Rightarrow Q_i)_{i \in I}}{P \mid Q_k} \qquad \frac{\mathsf{closeR} \mid (\mathsf{waitL}; Q)}{Q} \qquad \frac{\leftrightarrow \mid Q}{Q} \qquad \frac{Q \mid \leftrightarrow}{Q}$$

We remark that we treat $\mid$ associatively, that is to say, that $P \mid (Q \mid R)$ and $(P \mid Q) \mid R$ have the same reductions.

*Date*: 29 November 2017.

## 2. Programming with Subsingleton Logic

We now consider three examples. We adopt the convention that "$;$" binds more tightly than "$|$", that's to say, we interpret $P; Q \mid R; S$ as $(P; Q) \mid (R; S)$ instead of $P; (Q \mid R); S$.

### 2.1. Producing strings.

We first want to write a program that produces a given (constant) string. We assume the type string to be given by $\mathit{str} = \oplus\{\mathtt{a} : \mathit{str}, \mathtt{b} : \mathit{str}, \$ : \mathbf{1}\}$. Then $\mathbf{1} \vdash \ulcorner \mathtt{ababbb} \urcorner : \mathit{str}$ is given by

$$\ulcorner \mathtt{ababbb} \urcorner = \mathsf{R}.a; \mathsf{R}.b; \mathsf{R}.a; \mathsf{R}.b; \mathsf{R}.b; \mathsf{R}.b; \mathsf{R}.\$; \leftrightarrow.$$

What would have happened had we not included $\leftrightarrow$ at the end? We would end up with something ill-typed (cf. the $\oplus\mathsf{R}_k$ rule above). For the sake of illustration, we show the typing derivation of $\mathbf{1} \vdash \ulcorner \mathtt{a} \urcorner : \mathit{str}$:

$$\dfrac{\dfrac{\dfrac{}{\mathbf{1} \vdash \leftrightarrow : \mathbf{1}}\ \mathsf{id}_{\mathbf{1}}}{\mathbf{1} \vdash \mathsf{R}.\$; \leftrightarrow : \mathit{str}}\ \oplus\mathsf{R}_{\$}}{\mathbf{1} \vdash \mathsf{R}.a; \mathsf{R}.\$; \leftrightarrow : \mathit{str}}\ \oplus\mathsf{R}_{\mathtt{a}}$$

### 2.2. Incrementing Binary Integers.

We define the type $\mathit{bin} = \oplus\{\mathtt{b1} : \mathit{bin}, \mathtt{b0} : \mathit{bin}, \$ : \mathbf{1}\}$, analogously to $\mathit{str}$, and encode binary numbers $\mathtt{b}_0 \cdots \mathtt{b}_n$ in exactly the same way as we encoded strings, except that we reverse the bits (we do so due to the cut rule, as will be made clear below). For example, 110 is encoded as $\ulcorner 110 \urcorner = \mathsf{R}.\mathtt{b0}; \mathsf{R}.\mathtt{b1}; \mathsf{R}.\mathtt{b1}; \mathsf{R}.\$; \leftrightarrow$.

Given a binary number string $\mathtt{b}_0 \cdots \mathtt{b}_n$, give a program $\mathtt{inc}$ of type $\mathit{bin} \vdash \mathtt{inc} : \mathit{bin}$ that increments it. In particular, $\ulcorner \mathtt{b}_0 \cdots \mathtt{b}_n \urcorner \mid \mathtt{inc}$ should produce something of type $\mathit{bin}$ to the right that corresponds to the increment of $\mathtt{b}_0 \cdots \mathtt{b}_n$. It is useful to consider how we would implement this in ordered logic:

$$\dfrac{\mathtt{b0} \quad \mathtt{inc}}{\mathtt{b1}} \qquad \dfrac{\mathtt{b1} \quad \mathtt{inc}}{\mathtt{inc} \quad \mathtt{b0}} \qquad \dfrac{\$ \quad \mathtt{inc}}{\$ \quad \mathtt{b1}}$$

The corresponding program is

$$\mathtt{inc} = \mathsf{caseL}\ (\mathtt{b0} \Rightarrow \mathsf{R}.\mathtt{b1}; \leftrightarrow \mid \mathtt{b1} \Rightarrow \mathsf{R}.\mathtt{b0}; \mathtt{inc} \mid \$ \Rightarrow \mathsf{R}.\mathtt{b1}; \mathsf{R}.\$; \leftrightarrow)$$

We consider an example evaluation. For illustration purposes, we consider the helper program *turkey* that gobbles up everything passed to it from the left:

$$\mathit{turkey} = \mathsf{caseL}\ (\mathtt{b0} \Rightarrow \mathit{turkey} \mid \mathtt{b1} \Rightarrow \mathit{turkey} \mid \$ \Rightarrow \leftrightarrow).$$

Then we see that *turkey* does indeed gobble the increment $\ulcorner 10 \urcorner$ of 1:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathsf{R}.\mathtt{b1}; \mathsf{R}.\$; \leftrightarrow \mid \mathtt{inc} \mid \mathit{turkey}}{\mathsf{R}.\$; \leftrightarrow \mid \mathsf{R}.\mathtt{b0}; \mathtt{inc} \mid \mathit{turkey}}}{\mathsf{R}.\$; \leftrightarrow \mid \mathtt{inc} \mid \mathit{turkey}}}{\mathsf{R}.\mathtt{b1}; \mathsf{R}.\$; \leftrightarrow \mid \mathit{turkey}}}{\mathsf{R}.\$; \leftrightarrow \mid \mathit{turkey}}}{\leftrightarrow \mid \leftrightarrow}}{\leftrightarrow}$$

(For illustrative purposes, the input consumed by *turkey* is in red.)

Instead of *turkey*, we could instead have defined an id process:

$$\mathsf{id} = \mathsf{caseL}\ (\mathtt{b0} \Rightarrow (\mathsf{id} \mid \mathsf{R.b0}; \leftrightarrow) \mid \mathtt{b1} \Rightarrow (\mathsf{id} \mid \mathsf{R.b1}) \mid \$ \Rightarrow \mathsf{R.}\$; \leftrightarrow).$$

This process acts as the identity process, and when computing examples, has the advantage of leaving its input in sight:

$$\begin{array}{c}
\mathsf{R.b1}; \mathsf{R.}\$; \leftrightarrow \mid \mathtt{inc} \mid \mathsf{id} \\ \hline
\mathsf{R.}\$; \leftrightarrow \mid \mathsf{R.b0}; \mathtt{inc} \mid \mathsf{id} \\ \hline
\mathsf{R.}\$; \leftrightarrow \mid \mathtt{inc} \mid \mathsf{id} \mid \mathsf{R.b0}; \leftrightarrow \\ \hline
\mathsf{R.b1}; \mathsf{R.}\$; \leftrightarrow \mid \mathsf{id} \mid \mathsf{R.b0}; \leftrightarrow \\ \hline
\mathsf{R.}\$; \leftrightarrow \mid \mathsf{id} \mid \mathsf{R.b1}; \leftrightarrow \mid \mathsf{R.b0}; \leftrightarrow \\ \hline
\leftrightarrow \mid \mathsf{R.}\$; \leftrightarrow \mid \mathsf{R.b1}; \leftrightarrow \mid \mathsf{R.b0}; \leftrightarrow \\ \hline
\mathsf{R.}\$; \leftrightarrow \mid \mathsf{R.b1}; \leftrightarrow \mid \mathsf{R.b0}; \leftrightarrow
\end{array}$$

Again, we see that we end up with a process that will produce the intended output $\ulcorner 10 \urcorner$.

2.3. **String Reversal.** Given some string $c_0 \cdots c_n$, give a program `rev` of type *str* $\vdash$ `rev` : *str* that provides its reversal to the right. Assume the encoding given above.

We introduce the following thunks as helper functions, where $k \in \{a, b\}$:

$$T_k = \mathsf{caseL}\ (\$ \Rightarrow \mathsf{R.}k; \mathsf{R.}\$; \leftrightarrow \mid a \Rightarrow \mathsf{R.}a; T_k \mid b \Rightarrow \mathsf{R.}b; T_k).$$

Then $str \vdash T_k : str$ acts as the identity on all inputs from the left, except for R.\$, on which it outputs the thunked value R.$k$ followed by R.\$, and then terminates. We can now capture $str \vdash \mathtt{rev} : str$ as follows:

$$\mathtt{rev} = \mathsf{caseL}\ (a \Rightarrow (\mathtt{rev} \mid_{str} T_a) \mid b \Rightarrow (\mathtt{rev} \mid_{str} T_b) \mid \$ \Rightarrow \mathsf{R.}\$; \leftrightarrow).$$

We illustrate this with $\ulcorner ab \urcorner$, again using (an analogous) id to gobble up the output and marking the value passed to the right in red.

$$\begin{array}{c}
\ulcorner ab \urcorner \mid \mathtt{rev} \mid \mathsf{id} \\ \hline
\mathsf{R.}a; \mathsf{R.}b; \mathsf{R.}\$; \leftrightarrow \mid \mathtt{rev} \mid \mathsf{id} \\ \hline
\mathsf{R.}b; \mathsf{R.}\$; \leftrightarrow \mid \mathtt{rev} \mid T_a \mid \mathsf{id} \\ \hline
\mathsf{R.}\$; \leftrightarrow \mid \mathtt{rev} \mid T_b \mid T_a \mid \mathsf{id} \\ \hline
\leftrightarrow \mid \mathsf{R.}\$; \leftrightarrow \mid T_b \mid T_a \mid \mathsf{id} \\ \hline
\mathsf{R.}\$; \leftrightarrow \mid T_b \mid T_a \mid \mathsf{id} \\ \hline
\leftrightarrow \mid \mathsf{R.}b; \mathsf{R.}\$; \leftrightarrow \mid T_a \mid \mathsf{id} \\ \hline
\mathsf{R.}b; \mathsf{R.}\$; \leftrightarrow \mid T_a \mid \mathsf{id} \\ \hline
\mathsf{R.}\$; \leftrightarrow \mid \mathsf{R.}b; T_a \mid \mathsf{id} \mid \mathsf{R.}b; \leftrightarrow \\ \hline
\mathsf{R.}\$; \leftrightarrow \mid T_a \mid \mathsf{id} \mid \mathsf{R.}b; \leftrightarrow \\ \hline
\leftrightarrow \mid \mathsf{R.}a; \mathsf{R.}\$; \leftrightarrow \mid \mathsf{id} \mid \mathsf{R.}b; \leftrightarrow \\ \hline
\mathsf{R.}a; \mathsf{R.}\$; \leftrightarrow \mid \mathsf{id} \mid \mathsf{R.}b; \leftrightarrow \\ \hline
\mathsf{R.}\$; \leftrightarrow \mid \mathsf{id} \mid \mathsf{R.}a; \leftrightarrow \mid \mathsf{R.}b; \leftrightarrow \\ \hline
\leftrightarrow \mid \mathsf{R.}\$; \leftrightarrow \mid \mathsf{R.}a; \leftrightarrow \mid \mathsf{R.}b; \leftrightarrow \\ \hline
\mathsf{R.}\$; \leftrightarrow \mid \mathsf{R.}a; \leftrightarrow \mid \mathsf{R.}b; \leftrightarrow
\end{array}$$

Then, any process composed with the above to the right will first see $\mathsf{R}.b$, then $\mathsf{R}.a$, and finally $\mathsf{R}.\$$, i.e., the reversal of $\ulcorner \mathtt{ab} \urcorner$.