Some Basic Properties of Concurrent Computation

Recall, from Lecture 11, that, given a rewrite theory, \( R = (\Sigma, E, R) \) [more generally, we could add to \( R \) a \underline{frozenness map} \( \phi : \Sigma \to \mathcal{P}(\mathcal{N}) \) to restrict rewriting under some arguments, as described in a later lecture], a computation specification is a labeled arrow:

\[
[u] \xrightarrow{x} [v]
\]

prove in the logic \( \mathcal{L}(R) \) of \( R \) by the \underline{Reflexivity}, \underline{Congruence}, \underline{Replacement} and \underline{Transitivity} inference rules in pg. 6 of Lecture 11.

Notes: 1. Recall that, if a frozenness map \( \phi : \Sigma \to \mathcal{P}(\mathcal{N}) \) is given, the \underline{Congruence} and \underline{Replacement} inference rules are then \underline{restricted} to forbid rewriting under frozen positions.

2. A somewhat subtle point worth keeping in mind is that, since for any \( t' \in [t] \) in an \( E \)-equivalence class we have \( [t] = [t'] \), the \underline{Reflexivity} equivalence class rule could have been stated in the more explicit form:

\[
\text{Reflexivity} \quad [t] \xrightarrow{t} [t]
\]

that is, any \( t' \in [t] \) can be used for an idle transition.
finally, recall that a computation of \( R \) has the form:

\[
[u] \xrightarrow{[x]} [v]
\]

where \([x]\) is an equivalence class of proof terms by the equivalences defined in lecture 11, and computations then form a category \( \mathcal{C} \).

1. **Interleaving Descriptions**

Suppose \( R \) defines a Petri net with places \( a, b, c, d, e \) and transitions:

- [1]: \( a \cdot b \rightarrow b \cdot b \cdot a \)
- [2]: \( b \cdot c \rightarrow b \cdot c \cdot c \)
- [3]: \( d \cdot e \rightarrow d \cdot e \cdot a \)

Then we have the concurrent computation:

\[
[a \cdot b \cdot c \cdot d \cdot e] \rightarrow [a \cdot a \cdot b \cdot b \cdot b \cdot d \cdot e]
\]

but we also have; for any permutation \( \sigma \) of the set \{1, 2, 3\}

computations

\[
[a \cdot b \cdot c \cdot d \cdot e] \rightarrow [a \cdot a \cdot b \cdot b \cdot b \cdot d \cdot e]
\]

such that \([ [1 \ 2 \ 3] ] = [\sigma(1)\cdot u_1; \sigma(2)\cdot u_2; \sigma(3)\cdot u_3]\)
For example, for \( \sigma(1) = 2, \sigma(2) = 3, \sigma(3) = 1 \)
we have:

\[
[a \ b \ c \ d \ e] \xrightarrow{[2] a b d e} [a b c c d e] \xrightarrow{[3] a b c c e} [a a b c c d e] \xrightarrow{[1] a c c d e} [a a b b c c d e]
\]

What all such descriptions, called interleaving descriptions, have in common is that the computation is described in a sequentialized form, so that a single transition takes place at each step in the sequential composition.

The obvious question to ask is: Is this always possible. An the answer is: Yes!

**Sequentialization lemma.** Given a rewrite theory \( R = (\Sigma, E, R) \), any computation \( [u] \xrightarrow{[x]} [v] \) in \( \mathcal{F}_2 \) is either:

- \( [x] = [u] \) and \( [u] = [v] \) (idle computation), or
- \( [x] = [\beta_1, \ldots, \beta_n], n \geq 1 \), where each \( \beta \)
  - is a proof term of the form \( t \cdot l(u_1, \ldots, u_n) \)
  - with \( l : (u \rightarrow v) \in R \) involving variables \( x_1, \ldots, x_n \)
  - and \( t, u_1, \ldots, u_n \in T \Sigma \), and \( p \) a term position
  - in \( t \), as explained in more detail below.

Such a proof term is called a one-step rewrite.
Term positions and related notation

We can easily describe the positions of a term by displaying it as a tree, and numbering its nodes as follows:

For example, consider the term: \((0 + x) * ((y + z) * (x + x))\)

Then its positions are:

\[
\varepsilon^* \\
1 + \\
11 0 \quad 12 x \\
21 + \\
211 y \\
212 z \\
221 x \\
222
\]

That is, positions are strings in the monoid \(\mathbb{N}^*\).

Of course, we have a function

\[\text{pos} : T \Sigma \rightarrow \text{P}(\mathbb{N}^*)\]

\[t \mapsto \text{pos}(t)\]

Assigning to each term \(t\) the set of its positions. For example:

\[\text{pos}((0 + x) * ((y + z) * (x + x))) = \{ \varepsilon, 1, 2, 11, 12, 21, 211, 212, 22, 211, 212, 22\} \]

Also, given a term \(t\) and a position \(p \in \text{pos}(t)\) in it, then \(t|_p\) denotes the subterm at that position. For example:

\[(0 + x) * ((y + z) * (x + x))|_{21} = y + z\]
This notion of position allows us to decompose any term \( t \) such that \( t |_p = u \) into a context and the subterm \( u \) "plugged" into such a context at position \( p \). That is, \( t \) can be decomposed as: \( t = t[u]_p \), when \( u = t |_p \)

For example,

\[
(0+x) \ast ((y + z) \ast (x+x)) = (0+x)\ast \left( [y+z] \ast (x+x) \right)
\]

We can also perform surgery on a term by replacing a term \( u = t |_p \) by another term \( u' \) at the same position. For example, we may replace in the above term the subterm \( x+y \) at position 21 by the term \( z \ast (0+x) \) to get:

\[
(0+x) \ast \left( [z \ast (0+x)] \ast (x+x) \right) = 0+x \ast ((z \ast (0+x)) \ast (x+x))
\]

In general terms, if \( t = t[u]_p \) and \( u' \) is another term, we can replace \( u \) by \( u' \) at \( p \) to get \( t[u']_p \).

Given two positions \( p, q \in \text{POS}(t) \) of a term \( t \), any of the following exclusive possibilities can
happen:

1. \( p \) and \( q \) \{not necessarily distinct\} lay on the same path from the root of the tree as a tree, we then call \( p \) and \( q \) nested positions, in the sense that either \( p = q \), or the longer string, say \( q \), is nested inside its prefix string \( p \), i.e.

\[ q = p q' \quad \text{for some} \quad q' \]

maybe \( q' = \epsilon \), in case \( p = q \)

2. \( p \) and \( q \) are on different paths from the root. Then we call them disjoint positions.

Context notation can be generalized as follows:

1. If \( p_1, \ldots, p_k \) are disjoint positions, we have a decomposition

\[ t = t_{p_1} \cdots t_{p_k} \quad \text{where} \quad u_1 = t_{p_1}, \ldots, u_n = t_{p_k} \]
If \( q = pp' \) with \( p' \neq \varepsilon \) we also have a decomposition
\[
t = t \left[ u \left[ v \right]_{p'} \right]_p
\]
where \( u = t | p \) and \( v = t | pp' \).

Of course, this can be iterated to \( p_1p_2 \cdots p_k \)
with
\[
t = t \left[ u_1 \left[ u_2 \left[ \cdots \left[ u_k \right]_{p_k} \right] \right] \right]_{p_2} \]
and \( u_i = t | p_i \cdots p \).

And we can "mix and match" parallel decompositions
and nested ones, e.g.
\[
t \left[ u_1 \left[ u_2 \left[ v \right]_{p_3} \left[ v' \right]_{p_4} \right] \right] \left[ u_3 \left[ u_4 \left[ v'' \right]_{p_6} \right] \right]_{p_2}
\]

Now we can understand more clearly the notion of a
\underline{one-step rewrite step} as a proof term of the form:

\[
\begin{array}{c}
t \left[ l(v_1, \ldots, v_n) \right]_p \\
\left[ t \left[ u(v_1, \ldots, v_n) \right]_p \right] \rightarrow \left[ t \left[ v(v_1, \ldots, v_n) \right]_p \right]
\end{array}
\]

where \( l : U(x_1, \ldots, x_n) \rightarrow V(x_1, \ldots, x_n) \in R \).

What this step does is to \underline{replace the term} \( U(v_1, \ldots, v_n) \)
at position \( p \) by the term \( V(v_1, \ldots, v_n) \) at same position.
This notation has to be taken with a grain of salt, since, even if $x_1, \ldots, x_n$ appear in left-to-right order in $U(x_1, \ldots, x_n)$, the may not appear that way in $V(x_1, \ldots, x_n)$, since: (i) some variables may be dropped, and (ii) some variables may be repeated.

What $U(v_1, \ldots, v_n)$ and $V(v_1, \ldots, v_n)$ really mean is: $U\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$, $V\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$, but the shorthand notation $U(v_1, \ldots, v_n)$ is of course simpler.

**Sequential Systems**

Sequential computation is a degenerate special case of concurrent computation. For example:

1. an automaton
2. a Turing machine [deterministic or non-deterministic]
3. a sequential programming language [with no parallelization optimizations such as speculative execution]

are all sequential systems.

But if rewriting logic is a semantic framework for concurrent computation, it should be possible to characterize the sequential system subcase.

How? Quite simply:
Definition A rewrite theory $R = (\Sigma, E, R)$ is called **sequential** if and only if:

1. It has no **sideways parallelism**, that is, no proof term of the form:

   $t \left[ l(u_1, \ldots, u_n) \right] \rightarrow [v]$

   with $l : \Sigma^* \rightarrow \Sigma^*$ and $p, q$ disjoint

   and $p, q$ disjoint

2. It has no **nested concurrency**, i.e., no proof term of the form:

   $t \left[ l(u_1, \ldots, u_n) [l'(v_1, \ldots, v_m)] \rightarrow [v] \right]$

   with $p$ and $p', q'$ nested positions (for some $p'$)

   is possible at all.

An obvious question is then the following:

If $R$ is sequential, how do its computations in $TR$ look like?

The answer can be given as follows:
Theorem. If $\mathcal{R} = (\Sigma, E, R)$ is sequential, then, any computation $[U] \xrightarrow{[\alpha]} [V]$ has a unique interleaving description as either

1. $[U] \xrightarrow{[\epsilon]} [V]$ idle computation, or

2. $[U] \xrightarrow{[\beta_1]} [U_1] \ldots [U_{m-1}] \xrightarrow{[\beta_m]} [V]$

where unique means up to $E$-equality, i.e.,

$[\beta_1; \ldots; \beta_n] =_{E} (\beta'_1; \ldots; \beta'_n)$

and of course up to associativity of $\cdot$ : $(\beta_1; \beta_2; \beta_3) = (\beta_1; (\beta_2; \beta_3))$

Proof Hints

To prove this theorem, a useful notion is that of an idle proof term, which is a proof term built using the inference rules of $L(R)$ using only the:

- Reflexivity
- Congruence, and
- Transitivity

rules, and showing that if $[U] \xrightarrow{[\alpha]} [V]$ with $[\alpha]$ an idle proof term, then $[U] =_{E} [V]$ and $[\alpha] =_{E} [U]$.

Using this notion, the restrictions of a sequential $\mathcal{R}$, and induction [structural or on depth] of proof terms, one can prove this theorem.