Lecture Notes on
Temporal Types and Propositions

15-814: Types and Programming Languages
Frank Pfenning

Lecture 21
Thursday, November 18, 2021

1 Introduction

In the last lecture we introduced message-passing concurrency and wrote a couple of very simple circuits. One can also write other kinds of programs, for example, a concurrent implementation of queues or other data structures like trees. For this lecture, we continue to focus on circuits.

The goal is for an extension of the type system to ensure that the circuits are *well-timed*. In other words, a timing error should be manifest as a type error. This is the subject of ongoing research, so what we show in is this lecture is a simple model of clocked time with a simple model for the timing of gates (essentially, taking one unit of time).

This time, we start with a discussion of (probably the simplest possible) *temporal logic* and derive an extension of our process calculus from it.

2 Simplest Temporal Logic = Modal Logic K

The basic idea is that we have a new proposition $\bigcirc A$ that expresses that A will be true at the next (discrete) point in time. Or we could say it should be true after one tick of the clock. Obviously, $\bigcirc \bigcirc A$ then means that A should be true after two ticks. What laws should $\bigcirc A$ satisfy? Certainly, if A is necessarily true, that is, its truth is independent of time, then $\bigcirc A$ should be true. For example, we should have $\bigcirc(A \supset A)$. This is captured with the *rule of necessitation*:

$$\frac{\vdash A}{\vdash \bigcirc A}\ \mathsf{Nec}$$

The critical aspect of this rule is that there cannot be any hypotheses on the left of the turnstile ($\vdash$). We cannot turn this into an implication. For example, it is not true that $A \supset \bigcirc A$ for arbitrary A. That's because A may be true right now, but it may longer be true after one tick (for example, if I put down the whiteboard marker I currently hold). Similarly, it is not true in general that $\bigcirc A \supset A$. Just because I will pick up this marker after one tick that doesn't mean I hold it right now.

So it seems like there are not many laws that hold, but the logical connectives interact with $\bigcirc A$. One axiom (called K) that should hold is characteristic for many, so-called *normal* modal logics:

$$\vdash \bigcirc(A \supset B) \supset \bigcirc A \supset \bigcirc B$$

We can read is as follows: if after one tick, both A implies B and A are true, then B must also be true after one tick. In other words, we can reason in the world after one tick the same way we can reason in the present: the logical connectives retain their meaning. Now we can also consider whether $\bigcirc$ distributes over conjunction, disjunction, etc. and write corresponding axioms. The logic with the rule of necessitation and K is called the *modal logic K* and just has the inference rule of necessitation and the axiom K.

We can avoid a multitude of axioms if we use a single inference rule in a sequent calculus where we have antecedents on the left (our assumptions) and a succedent on the right (our desired conclusion).

$$\frac{A_1, \ldots, A_n \vdash B}{\Gamma, \bigcirc A_1, \ldots, \bigcirc A_n \vdash \bigcirc B}\ \bigcirc LR$$

This is a generalized form of necessitation where we allowed to carry forward all the antecedents that will be true after one tick, but not those starting with any other connective (collected in Γ). Note that with this generalized form of necessitation we can now *prove* the axiom K:

$$\dfrac{\dfrac{\dfrac{\vdots}{A \supset B, A \vdash B}}{\bigcirc(A \supset B), \bigcirc A \vdash \bigcirc B}\ \bigcirc LR}{\bigcirc(A \supset B) \vdash \bigcirc A \supset \bigcirc B}\ \supset R}{\cdot \vdash \bigcirc(A \supset B) \supset \bigcirc A \supset \bigcirc B}\ \supset R$$

The inference rule $\bigcirc LR$ is therefore characteristic of the modal logic K and can be found in other temporal or modal logics as well.

This logic is very simple in that we can only reason about truth a fixed number of ticks into the future. For example, we cannot say that a proposition A will always be true (perhaps $\Box A$) or that A will become true at some unknown point in the future (perhaps $\Diamond A$). We can also not look back into the past (perhaps $\bullet A$). Each of these addition would change the nature of the logic in significant ways. Our simple addition is elegant in that we do not need to change any of the existing rules or add a plethora of axioms, just one new rule for sequents.

3 Temporal Types

We add a corresponding new type $\bigcirc\,\tau$ while all other types and rules remain the same. There is only one construct associated with this time, namely one that ticks time forward.

$$\begin{array}{lrcl} \text{Types} & \tau & ::= & \ldots \mid \bigcirc\,\tau \\ \text{Processes} & P & ::= & \ldots \mid \mathsf{tick}\ ;\ P \end{array}$$

There is only one new typing rule:

$$\frac{x_1 : \tau_1, \ldots, x_n : \tau_n \vdash P :: (y : \sigma)}{\Gamma, x_1 : \bigcirc\,\tau_1, \ldots, x_n : \bigcirc\,\tau_n \vdash (\mathsf{tick}\ ;\ P) :: (y : \bigcirc\,\sigma)}\ \bigcirc LR$$

Where do these ticks come from? There are two sources: one is the *cost model*, the other is the programmer. A tick inserted by the cost model is part of the elaboration of the program. For example, the cost model might say that time ticks forward whenever a message is received. Expecting the compiler to insert these ticks then avoids errors in the instrumentation or mismatches with the expected running times. Furthermore, the programmer may need to delay certain operations so that send and receives are temporally synchronized.

In the dynamics we have to instrument the proc, msg and srv objects with a time t and enforce that a message is available at *exactly* the time it is expected by the receiver. We show the rules from last lecture in their temporally annotated form. Since time only passes with the tick constructs, all the rules have so far are parametric in the time t. However, when a message at time t is received by a service, that service must also be at time t. For well-typed (therefore well-timed) programs those times should always

match up precisely.

$$
\begin{array}{lcll}
\mathsf{proc}\ t\ (x \leftarrow P\ ;\ Q) & \mapsto & \mathsf{proc}\ t\ [a/x]P, \mathsf{proc} t\ \ [a/x]Q & \text{(a fresh)} \\
\mathsf{proc}\ t\ (a^+.V) & \mapsto & !\mathsf{msg}\ t\ a^+\ V & \\
\mathsf{proc}\ t\ (\mathsf{case}\ a^+\ K) & \mapsto & \mathsf{srv}\ t\ a^-\ K & \\
\mathsf{proc}\ t\ (a^-.V) & \mapsto & \mathsf{msg}\ t\ a^-\ V & \\
\mathsf{proc}\ t\ (\mathsf{case}\ a^-\ K) & \mapsto & !\mathsf{srv}\ t\ a^-\ K & \\
!\mathsf{msg}\ t\ a^+\ V, \mathsf{srv}\ t\ a^+\ K & \mapsto & \mathsf{proc}\ t\ (V \triangleright K) & \\
!\mathsf{srv}\ t\ a^-\ K, \mathsf{msg}\ t\ a^-\ V & \mapsto & \mathsf{proc}\ t\ (V \triangleright K) & \\
\mathsf{proc}\ t\ (a^+ \leftarrow c^+) & \mapsto & \mathsf{srv}\ t\ c^+\ a^+ & \\
!\mathsf{msg}\ t\ c^+\ V, \mathsf{srv}\ t\ c^+\ a^+ & \mapsto & !\mathsf{msg}\ t\ a^+\ V & \\
\mathsf{proc}\ t\ (a^- \leftarrow c^-) & \mapsto & \mathsf{msg}\ t\ a^-\ c^- & \\
!\mathsf{srv}\ t\ a^-\ K, \mathsf{msg}\ t\ a^-\ c^- & \mapsto & !\mathsf{srv}\ t\ c^-\ K &
\end{array}
$$

Finally, we have a single new rule for advancing time.

$$\mathsf{proc}\ t\ (\mathsf{tick}\ ;\ P)\ \mapsto\ \mathsf{proc}\ (t+1)\ P$$

The progress and preservation theorems now ensure temporal correctness in addition to the correctness of the messages. We won't state them here formally, but they are straightforward considering all we have learned this semester.

4 Example: A Nor Gate

We now revisit the first two examples from the last lecture and express their timing, followed by some examples with more complex timing.

Here is the code for *nor*.

$$\mathit{bit} = (\mathbf{b0} : 1) + (\mathbf{b1} : 1)$$

$$
\begin{array}{l}
x : \mathit{bit}, y : \mathit{bit} \vdash \mathit{nor} :: (z : \mathit{bit}) \\
z \leftarrow \mathit{nor}\ x\ y = \mathsf{case}\ x\ (\ \mathbf{b0} \cdot x' \Rightarrow \mathsf{case}\ y\ (\ \mathbf{b0} \cdot y' \Rightarrow z' \leftarrow z'.\langle\,\rangle\ ;\ z.(\mathbf{b1} \cdot z') \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathbf{b1} \cdot x' \Rightarrow z' \leftarrow z'.\langle\,\rangle\ ;\ z.(\mathbf{b0} \cdot z')\ ) \\
\qquad\qquad\qquad\qquad\qquad\quad \mid \mathbf{b1} \cdot x' \Rightarrow \mathsf{case}\ y\ (\ \mathbf{b0} \cdot y' \Rightarrow z' \leftarrow z'.\langle\,\rangle\ ;\ z.(\mathbf{b0} \cdot z') \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathbf{b1} \cdot x' \Rightarrow z' \leftarrow z'.\langle\,\rangle\ ;\ z.(\mathbf{b0} \cdot z')\ )\ )
\end{array}
$$

We would expect something like the type

$$\mathit{bit} = (\mathbf{b0} : 1) + (\mathbf{b1} : 1)$$

$$x : \mathit{bit}, y : \mathit{bit} \vdash \mathit{nor} :: (z : \bigcirc\, \mathit{bit})$$

which means that the output bit is computed in one clock cycle. The two bits of input x and y need to be received at the same time, which we model by allowing inputs on separate channels within the same clock cycle. However, after the second received we do advance time.

In order to visualize the timing we show the state of the type-checker at the point after the current line has been checked.

$$
\begin{array}{l}
x : \mathit{bit}, y : \mathit{bit} \vdash \mathit{nor} :: (z : \bigcirc\, \mathit{bit}) \\
z \leftarrow \mathit{nor}\, x\, y = \\
\quad \mathsf{case}\ x\ (\ \mathbf{b0} \cdot x' \Rightarrow \mathsf{case}\ y\ (\ \mathbf{b0} \cdot y' \Rightarrow \qquad\qquad \%\ x : \mathit{bit}, y : \mathit{bit}, x' : 1, y' : 1 \vdash z : \bigcirc\, \mathit{bit} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{tick}\ ; \qquad \%\ \cdot \vdash z : \mathit{bit}
\end{array}
$$

We observe here that when tick times forward we clear out the whole context because nothing is known to be available at the next time. On the right-hand side we now have the ability to output a bit, but we first need a continuation $z' : 1$.

$$
\begin{array}{l}
x : \mathit{bit}, y : \mathit{bit} \vdash \mathit{nor} :: (z : \bigcirc\, \mathit{bit}) \\
z \leftarrow \mathit{nor}\, x\, y = \\
\quad \mathsf{case}\ x\ (\ \mathbf{b0} \cdot x' \Rightarrow \mathsf{case}\ y\ (\ \mathbf{b0} \cdot y' \Rightarrow \qquad\qquad \%\ x : \mathit{bit}, y : \mathit{bit}, x' : 1, y' : 1 \vdash z : \bigcirc\, \mathit{bit} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{tick}\ ; \qquad \%\ \cdot \vdash z : \mathit{bit} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad z' \leftarrow z'.\langle\rangle\ ; \quad \%\ z' : 1 \vdash z : \mathit{bit}
\end{array}
$$

Now the table is set for send a $\mathbf{b1}$ along z.

$$
\begin{array}{l}
x : \mathit{bit}, y : \mathit{bit} \vdash \mathit{nor} :: (z : \bigcirc\, \mathit{bit}) \\
z \leftarrow \mathit{nor}\, x\, y = \\
\quad \mathsf{case}\ x\ (\ \mathbf{b0} \cdot x' \Rightarrow \mathsf{case}\ y\ (\ \mathbf{b0} \cdot y' \Rightarrow \qquad\qquad \%\ x : \mathit{bit}, y : \mathit{bit}, x' : 1, y' : 1 \vdash z : \bigcirc\, \mathit{bit} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{tick}\ ; \qquad \%\ \cdot \vdash z : \mathit{bit} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad z' \leftarrow z'.\langle\rangle\ ; \quad \%\ z' : 1 \vdash z : \mathit{bit} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad z.(\mathbf{b1} \cdot z') \\
\qquad\qquad\qquad\qquad\qquad\quad \mid \mathbf{b1} \cdot y' \Rightarrow \ldots\ ) \\
\qquad\qquad \mid \mathbf{b1} \cdot x' \Rightarrow \ldots\ )
\end{array}
$$

The other four branches are entirely analogous so we don't show them. We do want to observe that we could not have used x' or y' in the final send operation because, temporally speaking, they are no longer available.

5 Example: Operations on Streams of Bits

Next we want to transform a stream of bits with on tick between consecutive bits into a stream of output bits at the same rate. Because we want to take the *nor* of each pair of bits we represent the input as a stream of pairs of bits.

$bit = (\mathbf{b0} : 1) + (\mathbf{b1} : 1)$
$bits = \bigcirc\, bit \times \bigcirc\, bits$
$bits2 = (\bigcirc\, bit \times \bigcirc\, bit) \times \bigcirc\, bits2$

The streaming output of this circuit is delayed by the time it takes to compute the *nor*, that is, one tick.

$x : bit, y : bit \vdash nor :: (z : \bigcirc\, bit)$

$in : bits2 \vdash nors :: (out : \bigcirc\, bits)$
$out \leftarrow nors\ in =$
  case in ($\langle\langle x, y\rangle, in'\rangle \Rightarrow$ % $in : bits2, x : \bigcirc\, bit, y : \bigcirc\, bit, in' : \bigcirc\, bits2 \vdash out : \bigcirc\, bits$

Here, when we advance time after the input and pattern matching, we obtain x, y, and in' at the next time.

$x : bit, y : bit \vdash nor :: (z : \bigcirc\, bit)$

$in : bits2 \vdash nors :: (out : \bigcirc\, bits)$
$out \leftarrow nors\ in =$
  case in ($\langle\langle x, y\rangle, in'\rangle \Rightarrow$ % $in : bits2, x : \bigcirc\, bit, y : \bigcirc\, bit, in' : \bigcirc\, bits2 \vdash out : \bigcirc\, bits2$
    tick ; % $x : bit, y : bit, in' : bits2 \vdash out : bits$

Now we can compute the output bit z' from x and y, and the continuation stream from the recursive call. Both of these will only be available after a further tick, which is correct for the output along *out* because its continuation type is $\bigcirc\, bit \times \bigcirc\, bits$.

$x : bit, y : bit \vdash nor :: (z : \bigcirc\, bit)$

$in : bits2 \vdash nors :: (out : \bigcirc\, bits)$
$out \leftarrow nors\ in =$
  case in ($\langle\langle x, y\rangle, in'\rangle \Rightarrow$ % $in : bits2, x : \bigcirc\, bit, y : \bigcirc\, bit, in' : \bigcirc\, bits2 \vdash out : \bigcirc\, bits2$
    tick ; % $x : bit, y : bit, in' : bits2 \vdash out : bits$
    $out' \leftarrow nors\ in'$; % $x : bit, y : bit, in' : bits2, out' : \bigcirc\, bits \vdash out : bits$
    $z' \leftarrow nor\ x\ y$; % $x : bit, y : bit, in' : bits2, out' : \bigcirc\, bits, z' : \bigcirc\, bit \vdash out : bits$
    $out.\langle z', out'\rangle$)

6 Example: Merging Streams

In the next example we explore merging two streams by reading from them alternately. It should be quite evident that the input streams have to be twice as slow as the output stream. Furthermore, one of the two inputs has to be offset by one from the other. So we define:

$bit = (\mathbf{b0} : 1) + (\mathbf{b1} : 1)$
$bits^1 = \bigcirc\, bit \times \bigcirc\, bits^1$
$bits^2 = \bigcirc\bigcirc\, bit \times \bigcirc\bigcirc\, bits^2$

$x : bits^2, y : \bigcirc\, bits^2 \vdash alternate :: (z : \bigcirc\, bits^1)$

We first write the code without consideration for timing. The key is to make the recursive call with the arguments reversed.

$$
\begin{array}{l}
z \leftarrow alternate\ x\ y = \\
\quad case\ x\ (\ \langle a, x' \rangle \Rightarrow z' \leftarrow alternate\ y\ x'\ ; \\
\qquad\qquad\qquad\quad z.\langle a, z' \rangle\ )
\end{array}
$$

Next, we see if we can assign proper types.

$$
\begin{array}{lll}
z \leftarrow alternate\ x\ y = & & \\
\quad case\ x\ (\ \langle a, x' \rangle \Rightarrow & & \%\ a : \bigcirc\bigcirc\, bit, x' : \bigcirc\bigcirc\, bits^2, y : \bigcirc\, bits^2 \vdash z : \bigcirc\, bits^1 \\
\qquad\qquad\qquad\quad \mathsf{tick}\ ; & & \%\ a : \bigcirc\, bit, x' : \bigcirc\, bits^2, y; bits^2 \vdash z : bits^1 \\
\qquad\qquad\qquad\quad z' \leftarrow alternate\ y\ x'\ ; & & \%\ a : \bigcirc\, bit, x' : \bigcirc\, bits^2, y : bits^2, z' : \bigcirc\, bits^1 \vdash z : bits^1 \\
\qquad\qquad\qquad\quad z.\langle a, z' \rangle\ ) & &
\end{array}
$$

7 Example: Duplicating Bits in a Stream

In the case where we want to duplicate the bits in a stream, again the input stream has to arrive at half the rate of the output stream.

$bit = (\mathbf{b0} : 1) + (\mathbf{b1} : 1)$
$bits^1 = \bigcirc\, bit \times \bigcirc\, bits^1$
$bits^2 = \bigcirc\bigcirc\, bit \times \bigcirc\bigcirc\, bits^2$

$x : bits^2 \vdash dup :: (y : \bigcirc\, bits^1)$

Again, we first program without regard to timing.

$x : bits^2 \vdash dup :: (y : \bigcirc\, bits^1)$

$$
\begin{array}{l}
y \leftarrow dup\ x = \\
\mathsf{case}\ x\ (\ \langle a, x' \rangle \Rightarrow y'' \leftarrow dup\ x'\ ; \\
\qquad\qquad\qquad\quad y' \leftarrow y'.\langle a, y'' \rangle\ ; \\
\qquad\qquad\qquad\quad y.\langle a, y' \rangle\ )
\end{array}
$$

Next consider timing.

$x : \mathit{bits}^2 \vdash \mathit{dup} :: (y : \bigcirc\, \mathit{bits}^1)$

$y \leftarrow \mathit{dup}\ x =$
$\mathsf{case}\ x\ (\ \langle a, x' \rangle \Rightarrow$ % $x : \mathit{bits}^2, a : \bigcirc\bigcirc\, \mathit{bit}, x' : \bigcirc\bigcirc\, \mathit{bits}^2 \vdash y : \bigcirc\, \mathit{bits}^1$
$\quad \mathsf{tick}\ ;$ % $a : \bigcirc\, \mathit{bit}, x' \bigcirc\, \mathit{bits}^2 \vdash y : \mathit{bits}^1$
$\quad y'' \leftarrow \mathit{dup}\ x'\ ;$
$\quad y' \leftarrow y'.\langle a, y'' \rangle\ ;$
$\quad y.\langle a, y' \rangle\ )$

The next point is interesting: we need to delay for one tick before we can call *dup* recursively. We do with with $\mathsf{tick}\ ;\ y'' \leftarrow \mathit{dup}\ x'$. Note that $x' : \mathit{bits}^2$ because it occurs underneath the tick. On the outside, then $y'' : \bigcirc\bigcirc\, \mathit{bits}^1$.

$x : \mathit{bits}^2 \vdash \mathit{dup} :: (y : \bigcirc\, \mathit{bits}^1)$

$y \leftarrow \mathit{dup}\ x =$
$\mathsf{case}\ x\ (\ \langle a, x' \rangle \Rightarrow$
% $x : \mathit{bits}^2, a : \bigcirc\bigcirc\, \mathit{bit}, x' : \bigcirc\bigcirc\, \mathit{bits}^2 \vdash y : \bigcirc\, \mathit{bits}^1$
$\quad \mathsf{tick}\ ;$ % $a : \bigcirc\, \mathit{bit}, x' \bigcirc\, \mathit{bits}^2 \vdash y : \mathit{bits}^1$
$\quad y'' \leftarrow (\mathsf{tick}\ ;\ y'' \leftarrow \mathit{dup}\ x')\ ;$ % $a : \bigcirc\, \mathit{bit}, x' : \bigcirc\, \mathit{bits}^2, y'' : \bigcirc\bigcirc\, \mathit{bits}^1 \vdash y : \mathit{bits}^1$
$\quad y' \leftarrow y'.\langle a, y'' \rangle\ ;$
$\quad y.\langle a, y' \rangle\ )$

The same reasoning now applies to the next line, since we want $y' : \bigcirc\, \mathit{bits}^1$.

$x : \mathit{bits}^2 \vdash \mathit{dup} :: (y : \bigcirc\, \mathit{bits}^1)$

$y \leftarrow \mathit{dup}\ x =$
$\mathsf{case}\ x\ (\ \langle a, x' \rangle \Rightarrow$
% $x : \mathit{bits}^2, a : \bigcirc\bigcirc\, \mathit{bit}, x' : \bigcirc\bigcirc\, \mathit{bits}^2 \vdash y : \bigcirc\, \mathit{bits}^1$
$\quad \mathsf{tick}\ ;$ % $a : \bigcirc\, \mathit{bit}, x' \bigcirc\, \mathit{bits}^2 \vdash y : \mathit{bits}^1$
$\quad y'' \leftarrow (\mathsf{tick}\ ;\ y'' \leftarrow \mathit{dup}\ x')\ ;$ % $a : \bigcirc\, \mathit{bit}, x' : \bigcirc\, \mathit{bits}^2, y'' : \bigcirc\bigcirc\, \mathit{bits}^1 \vdash y : \mathit{bits}^1$
$\quad y' \leftarrow (\mathsf{tick}\ ;\ y'.\langle a, y'' \rangle)\ ;$!! % $a : \bigcirc\, \mathit{bit}, x' : \bigcirc\, \mathit{bits}^2, y'' : \bigcirc\bigcirc\, \mathit{bits}^1, y' : \bigcirc\, \mathit{bits}^1 \vdash y : \mathit{bits}^1$
$\quad y.\langle a, y' \rangle\ )$

However, we (or the type-checker) now notices a bug at the line marked with "!!". Because $a : \bigcirc\, \mathit{bit}$, after the tick is will have type $a : \mathit{bit}$ so we cannot send it along $y' : \mathit{bits}^1$. That's because $\mathit{bits}^1 = \bigcirc\, \mathit{bit} \times \bigcirc\, \mathit{bits}^1$. So we have to create a delayed copy of a, carrying it forward by one tick so it can be the second a that is output along y. For the sake of brevity we omit channels in the context we no longer need.

$x : \mathit{bits}^2 \vdash \mathit{dup} :: (y : \bigcirc\, \mathit{bits}^1)$
$x : \mathit{bit} \vdash \mathit{delay} :: (x' : \bigcirc\, \mathit{bit})$

$$
\begin{array}{lll}
y \leftarrow \mathit{dup}\ x = & & \\
\mathsf{case}\ x\ (\ \langle a, x' \rangle \Rightarrow & & \\
\qquad & \% \ x : \mathit{bits}^2, a : \bigcirc\bigcirc \mathit{bit}, x' : \bigcirc\bigcirc \mathit{bits}^2 \vdash y : \bigcirc \mathit{bits}^1 \\
\quad \mathsf{tick}\ ; & \% \ a : \bigcirc \mathit{bit}, x' \bigcirc \mathit{bits}^2 \vdash y : \mathit{bits}^1 \\
\quad y'' \leftarrow (\mathsf{tick}\ ; y'' \leftarrow \mathit{dup}\ x')\ ; & \% \ a : \bigcirc \mathit{bit}, y'' : \bigcirc\bigcirc \mathit{bits}^1 \vdash y : \mathit{bits}^1 \\
\quad a' \leftarrow (\mathsf{tick}\ ; a' \leftarrow \mathit{delay}\ a)\ ; & \% \ a : \bigcirc \mathit{bit}, a' : \bigcirc\bigcirc \mathit{bit}, y'' : \bigcirc\bigcirc \mathit{bits}^1 \vdash y : \mathit{bits}^1 \\
\quad y' \leftarrow (\mathsf{tick}\ ; y'.\langle a', y'' \rangle)\ ; & \% \ a : \bigcirc \mathit{bit}, y' : \bigcirc \mathit{bits}^1 \vdash y : \mathit{bits}^1 \\
\quad y.\langle a, y' \rangle\ ) &
\end{array}
$$

Now everything checks, assuming we can write the *delay* process. In hindsight we should have anticipated that using the same input twice, at different times, requires creating some form of a delayed copy. Note that the additional occurrences of tick we inserted are the programmers responsibility and not due to the cost model since they do not follow a receive operation. Fortunately, writing it is straightforward: it receives an input bit (which cost a tick, according to our cost model) and then sends the same bit.

$$\mathit{bit} = (\mathbf{b0} : 1) + (\mathbf{b1} : 1)$$

$$x : \mathit{bit} \vdash \mathit{delay} :: (y : \bigcirc \mathit{bit})$$

$$
\begin{array}{lll}
y \leftarrow \mathit{delay}\ x = \mathsf{case}\ x\ (\ \mathbf{b0} \cdot x' \Rightarrow & & \% \ x : \mathit{bit}, x' : 1 \vdash y : \bigcirc \mathit{bit} \\
\qquad\qquad\qquad\qquad\quad \mathsf{tick}\ ; & & \% \ \cdot \vdash y : \mathit{bit} \\
\qquad\qquad\qquad\qquad\quad y' \leftarrow y'.\langle\rangle)\ ; & & \% \ y' : 1 \vdash y : \mathit{bit} \\
\qquad\qquad\qquad\qquad\quad y.(\mathbf{b0} \cdot y') & & \\
\qquad\qquad\quad \mid \mathbf{b1} \cdot x' \Rightarrow \mathsf{tick}\ ; & & \\
\qquad\qquad\qquad\qquad\quad y' \leftarrow y'.\langle\rangle\ ; & & \\
\qquad\qquad\qquad\qquad\quad y.(\mathbf{b1} \cdot y')\ ) & &
\end{array}
$$

An alternative solution to the problem would be

$$
\begin{array}{l}
x : \mathit{bits}^2 \vdash \mathit{delays2} :: (y : \bigcirc \mathit{bits}^2) \\
x : \mathit{bits}^2 \vdash \mathit{dup} :: (y : \bigcirc \mathit{bits}^1) \\
y \leftarrow \mathit{dup}\ x = \\
\quad x' \leftarrow \mathit{delays2}\ x\ ; \\
\quad y \leftarrow \mathit{alternate}\ x\ x'
\end{array}
$$

We leave it to Exercise 1 to define *delays2*.

Exercises

Exercise 1 Write $x : \mathit{bits}^2 \vdash \mathit{delays2} :: (y : \bigcirc \mathit{bits}^2)$ which delays the input stream by tick, or argue that it cannot be done. If needed, you may use $a : \mathit{bit} \vdash \mathit{delay} :: (b : \mathit{bit})$ defined in Section 7.

Exercise 2 Rewrite the flip/flip pipeline so it is well-timed.

Exercise 3 Rewrite the latch so it is well-timed.

References