# What is termination?

It would be nice to be able to write

$$P;\ Q$$

which runs $P$ until it terminates, and then runs $Q$.

But what does 'terminates' mean?

We have seen two ways a CSP program can (sort of) terminate:

- *deadlock*: reach a state where it can perform no further actions
- *diverge*: perform an infinite sequence of invisible actions.

But both of these are errors rather than a positive statement that our process is finished.

Therefore we introduce a special event $\checkmark$ (tick) which signals that the process has terminated successfully, and it is this event that triggers $Q$ in $P;\ Q$.

# $SKIP$

$SKIP$ is a process that terminates immediately. It is essentially $\checkmark \rightarrow STOP$, but

- $\checkmark$ only arises through the process $SKIP$ (so $\checkmark \rightarrow STOP$ is not legal syntax), and therefore
- $\checkmark$ is outside $\Sigma$.

Think of $\checkmark$ as a signal that the process *has terminated*, rather than an event like ordinary ones which required the environment to cooperate.

$\Sigma^{\checkmark}$ is the set $\Sigma \cup \{\checkmark\}$.

## Handing over control

$SKIP;\ P = P$ for all $P$: the $\checkmark$ from the first process gets hidden. It becomes a $\tau$ action.

However $STOP;\ P = STOP$ and $\mathbf{div};\ P = \mathbf{div}$, because neither $STOP$ nor $\mathbf{div}$ terminates properly.

Further laws (e.g. associative, distributive and step) will be given later.

# Sequential composition in a declarative world

One might expect the following to be true:

$$(?x : A \to P);\ Q = ?x : A \to (P;\ Q)$$

(noting that as $A \subseteq \Sigma$ we know $\checkmark \notin A$.)

But this implies

$$(?x : A \to SKIP);\ x \to STOP$$
$$= ?x : A \to (SKIP;\ (x \to STOP))$$

which is not true because the $x$'s have different bindings.

The input in $?x : A \to P$ does not assign a value to a *variable* $x$, rather it creates a new *identifier* whose scope is just $P$.

A hole is created in the scope of any existing $x$.

So the law above is only valid if $Q$ has no free instances of the name $x$.

## The use of sequential composition

The declarative semantics means that there is no direct way to pass information from $P$ to $Q$ in $P; Q$.

This usually restricts the cases in which we can use sequencing to cases where $Q$ is independent of what $P$ does.

(The alternative is to put $P; Q$ in parallel with a process which holds the state: $P$ can output to it and $Q$ input.)

The result is that ; is less used than you might expect given the role of sequential composition in sequential languages.

# Another counter

$$ZERO = up \to POS;\ ZERO, \quad \text{where}$$

$$\begin{aligned} POS = {} & up \to POS;\ POS \\ & \Box\ down \to SKIP \end{aligned}$$

$POS$ is a process that terminates as soon as it has communicated one more $down$'s than $up$'s.

# Iteration

$$P^{*} = P;\ P^{*}$$

runs $P$ over and over again, *ad infinititum*.....

Note that this makes more sense in a communicating process world than in situations where a program only gives results when it terminates.

For example, $(a \rightarrow SKIP)^{*}$ is is indistinguishable from $\mu\, p.a \rightarrow p$.
Similarly,

$$COPY = (left?x \rightarrow right!x \rightarrow SKIP)^{*}$$

# Distributed termination

How should $P \;{}_X\|_Y\; Q$ terminate? If $\checkmark$ were a possible member of $X$ and $Y$ there would be four answers:

- If $\checkmark \notin X \cup Y$ then it can never terminate.
- If $\checkmark \in X \setminus Y$ then it will terminate whenever $P$ does.
- If $\checkmark \in Y \setminus X$ then it will terminate whenever $Q$ does.
- If $\checkmark \in X \cap Y$ then it terminates when both $P$ and $Q$ do.

The first is too severe, and the next two are asymmetric and pose questions about what the unterminated process does after the other has terminated.

The last is known as *distributed termination*: the parallel combination has not terminated until both components have.

But $X, Y \subseteq \Sigma$, so we don't have this sort of choice. All CSP parallel operators (even $|||$) use distributed termination.

# Termination and other operators

The fact that $\checkmark \notin \Sigma$ has two other consequences:

- $\checkmark$ is never hidden by the hiding operator $P \setminus X$ (as $X \subseteq \Sigma$), and
- $\checkmark$ is never either the subject or target of renaming $P[\![R]\!]$ (as $R$ is a relation on $\Sigma$).

## Laws of sequencing

$$(P \sqcap Q);\ R = (P;\ R) \sqcap (Q;\ R) \qquad \langle ;\text{-dist-l}\rangle$$

$$P;\ (Q \sqcap R) = (P;\ Q) \sqcap (P;\ R) \qquad \langle ;\text{-dist-r}\rangle$$

$$P;\ (Q;\ R) = (P;\ Q);\ R \qquad \langle ;\text{-assoc}\rangle$$

$$\mathit{SKIP};\ P = P \qquad \langle ;\text{-unit-l}\rangle$$

$$P;\ \mathit{SKIP} = P \qquad \langle ;\text{-unit-r}\rangle$$

The last of the above laws, though intuitively obvious, requires a good deal of care in modelling to make it true.

# A less obvious law

In order to get $\langle ;\text{-unit-r} \rangle$ to work, we have to adopt the following.

It states the principle that termination is something signalled to the environment, rather than negotiated with it.

$$P \mathbin{\Box} SKIP \;=\; P \triangleright SKIP \qquad\qquad \langle \Box\text{-}SKIP \text{ resolve} \rangle$$

$P \triangleright Q$ is the process that can choose to act like $Q$ but can offer the initial choices of $P$.

This law says that any process that has the option to terminate is refined by $SKIP$.

## Two step laws

Step laws now have to account for initial ✓'s. Consider ; .

First, the case where no ✓ is possible.

Provided $x$ is not free in $Q$,

$$(?x : A \to P);\ Q = ?x : A \to (P;\ Q) \qquad [x \notin fv(Q)]\,\langle ;\text{-step}\rangle$$

When ✓ is possible we use $\langle \Box\text{-}SKIP$ resolve$\rangle$ and the following.

When $x$ is not free in $Q$,

$$((?x : A \to P) \triangleright SKIP);\ Q =$$
$$(?x : A \to (P;\ Q)) \triangleright Q \qquad \langle SKIP\text{-};\ \text{-s}$$

# Laws of distributed termination

$$SKIP \; {}_X\|_Y \; SKIP = SKIP \qquad \langle {}_X\|_Y\text{-termination} \rangle$$

$$SKIP \underset{X}{\|} SKIP = SKIP \qquad \langle \underset{X}{\|}\text{-termination} \rangle$$

$$SKIP \gg SKIP = SKIP \qquad \langle \gg\text{-termination} \rangle$$

$$SKIP /\!/_X SKIP = SKIP \qquad \langle /\!/\text{-termination} \rangle$$

See book for more laws involving $;$ and $SKIP$.

## An example of UFP

To prove $ZERO$ is equivalent to $COUNT_0$ we prove that the vector of processes $\langle Z_n \mid n \in \mathbb{N} \rangle$, defined

$$Z_0 = ZERO \quad \text{and} \quad Z_{n+1} = POS;\ Z_n$$

(an inductive definition rather than a recursive one) is a fixed point of the constructive recursion defining $\underline{COUNT}$.

$Z_0 = up \rightarrow POS;\ Z_0 = up \rightarrow Z_1$, and

$$
\begin{aligned}
Z_{n+1} &= (up \rightarrow POS;\ POS \\
&\qquad \Box\ down \rightarrow SKIP);\ Z_n \\
&= (up \rightarrow POS;\ POS;\ Z_n) \\
&\qquad \Box\ (down \rightarrow SKIP;\ Z_n) && \text{by } \langle ;\text{-step}\rangle \text{ etc.} \\
&= (up \rightarrow POS;\ POS;\ Z_n) \\
&\qquad \Box\ (down \rightarrow Z_n) && \text{by } \langle ;\text{-unit-l}\rangle \\
&= up \rightarrow Z_{n+2} \\
&\qquad \Box\ down \rightarrow Z_n
\end{aligned}
$$

This completes the proof.

## ✓ and the traces model

✓'s can now appear in traces, but only as the last event, so $\mathcal{T}$ is extended to be all nonempty prefix-closed subsets of

$$\Sigma^{*\checkmark} = \Sigma^* \cup \{s\hat{}\langle\checkmark\rangle \mid s \in \Sigma^*\}$$

$$\begin{aligned} traces(SKIP) &= \{\langle\rangle, \langle\checkmark\rangle\} \\ traces(P;\ Q) &= (traces(P) \cap \Sigma^*) \\ &\qquad \cup \{s\hat{}t \mid s\hat{}\langle\checkmark\rangle \in traces(P) \\ &\qquad\quad \wedge\, t \in traces(Q)\} \end{aligned}$$

See book for the effects of ✓ on the other clauses of trace semantics.

# $\checkmark$ and failures

Since we need to distinguish $SKIP$ and $SKIP \sqcap STOP$, it follows that $\checkmark$ needs to be in refusal sets.

But if you want $\langle \Box$-$SKIP$ resolve$\rangle$ to be true, this has to be done carefully: if a process can accept $\checkmark$, then it can refuse everything else.

See book for details.

# Interrupt

$P \bigtriangleup Q$ is a process that behaves like $P$ until an initial event of $Q$ occurs, at which point $Q$ takes over.

Somewhere between $\|$ (for $P$) and $\Box$ (for $Q$).

Almost invariably $Q$ takes the form $?x : A \rightarrow Q'$ or $a \rightarrow Q'$, because otherwise $Q$ might perform $\tau$ actions while $P$ is running.

So (over $\mathcal{N}$) $P \bigtriangleup \mathbf{div} = \mathbf{div}$, and in general $P \bigtriangleup \mathbf{div}$ is never stable.

$\bigtriangleup$ can introduce nondeterminism when $\mathit{initials}(Q) \cap \alpha P \neq \{\}$.

## Resetting

For example, if $reset \notin \alpha P$

$$Resettable(P) = P \bigtriangleup reset \rightarrow P$$

is a process that behaves like $P$ but at any time can be sent back to the start by the event $reset$.

# Example

$\triangle$ useful for controlled faults: for example

$$P^{\lightning} = (P \triangle spike \rightarrow CHAOS) \triangle \lightning \rightarrow STOP$$

Suppose we can control $spike$s but not $\lightning$s. Under what circumstances can we build an operator $recover(\cdot)$ so that $recover(P^{\lightning}) = P \mathbin{|||} (\mu\, p.\lightning \rightarrow p)$

## Recovery

We can use $resettable(P^{\lightning} \underset{\{spike\}}{\|} STOP)$

under a double renaming construct which, each time $\lightning$ happens, resets and replays the current trace (without $\lightning$s) to $P$, hiding the replay.

But only when $P$ is deterministic.

## Laws

$\triangle$ is distributive, has left and right unit $STOP$, and over $\mathcal{N}$ is strict in both arguments.

$$
\begin{aligned}
(?x : A \to P(x)) \triangle (?x : B \to Q(x)) \ = \\
?x : A \cup B \to ((P(x) \triangle Q) \sqcap Q(x) \\
\blacktriangleleft x \in A \cap B \blacktriangleright \\
(P(x) \triangle Q) \blacktriangleleft x \in A \blacktriangleright Q(x))
\end{aligned}
$$

$\langle \triangle$–s

## A new operator

$P \,\Theta_A\, Q$ behaves like $P$ until $P$ performs an action in $A$, at which point it starts $Q$.

Like $P$ throwing an exception, hence the throw operator.

Both useful for programming, and also for theory.....

# Laws

$\Theta_A$ is distributive. Over $\mathcal{N}$ it is strict in the left argument but not in the right.

$$(?x : A \to P(x))\,\Theta_B\,Q \;=\; \qquad \langle \Theta_B\text{–step}\rangle$$
$$?x : A \to (P(x)\,\Theta_B\,Q \,\not<\, x \notin B \,\not>\, Q)$$

Not in FDR till 2.91.