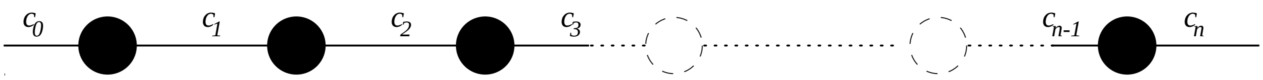
Chains

A common form of network consists of a series of processes, each with one input and one output channel, connected in a chain.

An example is the composition of $COPY$ processes seen earlier:



We had to invent many different channel names for this system.

Piping

The *piping*, or *chaining* operator avoids this problem by combining parallel, hiding and renaming.

It assumes the input and output channels are respectively $left$ and $right$, so that

$$P \gg Q$$

has the $left$ of P, the $right$ of Q, with the other two channels connected and hidden. The $COPY$ chain becomes

$$COPY \gg COPY \gg \ldots \gg COPY$$

The definition of $\gg$

It is a *derived* operator: $P \gg Q \;=$

$$(P[\![^{right,\, mid}/_{mid,\, right}]\!] \;{}_{\{|left, mid|\}}\|_{\{|mid, right|\}}\; Q[\![^{mid,\, left}/_{left,\, mid}]\!]) \setminus \{|\ mid\ |\}$$

where it is assumed $left$, $right$ and the new channel name mid all have the same type T.

Note that the directions of the channels are not implied by the above, but by convention $left$ is input and $right$ is output.

Laws of $\gg$

We need to quote left and right distributive laws separately....why?

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

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

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

The associativity law is closely related to $\langle$hide-${}_X\|_Y$-dist$\rangle$.

Step laws of $\gg$

The full step law is cumbersome (see book), and the following are usually sufficient:

$$\begin{array}{ll} \mathit{left}?x \rightarrow P \gg \mathit{left}?y \rightarrow Q \ = & \\ \qquad \mathit{left}?x \rightarrow (P \gg \mathit{left}?y \rightarrow Q) & \langle \gg\text{-step 1} \rangle \end{array}$$

$$\begin{array}{ll} \mathit{right}!x \rightarrow P \gg \mathit{right}!y \rightarrow Q \ = & \\ \qquad \mathit{right}!y \rightarrow (\mathit{right}!x \rightarrow P \gg Q) & \langle \gg\text{-step 2} \rangle \end{array}$$

$$\begin{array}{ll} \mathit{right}!x \rightarrow P \gg \mathit{left}?y \rightarrow Q \ = & \\ \qquad P \gg Q[x/y] & \langle \gg\text{-step 3} \rangle \end{array}$$

$$\begin{array}{ll} \mathit{left}?x \rightarrow P \gg \mathit{right}!y \rightarrow Q \ = & \\ \qquad \mathit{left}?x \rightarrow (P \gg \mathit{right}!y \rightarrow Q) & \langle \gg\text{-step 4} \rangle \\ \qquad \Box \ \mathit{right}!y \rightarrow (\mathit{left}?x \rightarrow P \gg Q) & \end{array}$$

Example

Let's consider $COPY \gg COPY$. Applying the step laws gives:

$$
\begin{aligned}
& (left?x \rightarrow \; right!x \rightarrow COPY) \gg \\
& (left?y \rightarrow \; right!y \rightarrow COPY) \\
= \quad & left?x \rightarrow \; ((right!x \rightarrow COPY) \gg \\
& \qquad (left?y \rightarrow right!y \rightarrow COPY)) \\
= \quad & left?x \rightarrow \; (COPY \gg (right!x \rightarrow COPY)) \\
= \quad & left?x \rightarrow \; (left?y \rightarrow right!y \rightarrow COPY \gg \\
& \qquad (right!x \rightarrow COPY)) \\
= \quad & left?x \rightarrow \; (right!x \rightarrow (COPY \gg COPY) \\
& \qquad \Box \; left?y \rightarrow \\
& \qquad (right!y \rightarrow COPY \gg right!x \rightarrow COPY))
\end{aligned}
$$

etc.

Traces of $P \gg Q$

Assume P and Q only communicate in the set $\{\!|\ \mathit{left}, \mathit{right}\ |\!\}$.

Then $\langle P \gg Q \rangle\ =$

$$
\begin{aligned}
\{u \in \{\!|\ \mathit{left}, \mathit{right}\ |\!\}^* \quad & \mid \exists\, s \in \mathit{traces}(P), t \in \mathit{traces}(Q). \\
& \quad u \downarrow \mathit{left} = s \downarrow \mathit{left} \\
& \wedge\, s \downarrow \mathit{right} = t \downarrow \mathit{left} \\
& \wedge\, t \downarrow \mathit{right} = u \downarrow \mathit{right}\}
\end{aligned}
$$

Example

$$
\begin{aligned}
ITER \quad = \quad & left?(data, input) \rightarrow \\
& right!(data, F(data, input)) \rightarrow ITER
\end{aligned}
$$

performs one step of an iteration. If

$$F(d, x) \;=\; \left(x + \frac{d}{x}\right)/2$$

then it is Newton's method for approximating the square root.

$$ITER \gg ITER \gg \ldots \gg ITER$$

is a pipeline that applies the iteration n times.

Note that this system has much higher throughput than a single process doing n iterations to each piece of data.

Communications protocols

Suppose we want to build a reliable and secure buffer of words W between two processes, when what we are supplied with is an unreliable channel M of bytes $Byte$ which might lose or corrupt some proportion of its throughput.

You may be able to design T and R such that

$$T \gg M \gg R$$

is a *buffer* of type W, and which sends only encrypted data through M.

A buffer is a process that satisfies the appropriate trace specification $tr \downarrow right \leq tr \downarrow left$ and appropriate liveness (failures) conditions we will see shortly (Chapter 5).

Protocol layering

It is often natural to build a protocol in layers:

- $T3 \gg M \gg R3$ might be a buffer of $Byte$: $T3$ and $R3$ correcting errors.
- $T2 \gg R2$ might be a buffer of W that uses $Byte$ on its internal channel.
- $T1 \gg R1$ might be a buffer of W that passes encrypted W's over its internal channel.

$$T1 \gg T2 \gg T3 \gg M \gg R3 \gg R2 \gg R1$$

is then the system we want.

Notice the "onion-like" structure of this.

Recursing through $\gg$

We can create dynamically expanding chains of processes, for example

$$B^{+} \;=\; \mathit{left}?x \rightarrow (B^{+} \gg \mathit{right}!x \rightarrow \mathit{COPY})$$

has behaviour equivalent to $B^{\infty}_{\langle\rangle}$.

But $\gg$ uses hiding, so this recursion is not obviously guarded.

In fact it is, because $\mathit{right}!x \rightarrow \mathit{COPY}$ has always communicated at least as many unhidden $\mathit{right}.x$'s as hidden $\mathit{left}.x$'s. Thus the trace of

$$P \gg \mathit{right}!x \rightarrow \mathit{COPY}$$

is always at least as long as that of P.

See book for more details.

Enslavement

$$P /\!/_Y Q$$

means that Q is acting as the *slave* of P, with Q's alphabet Y being hidden.

It means the same as

$$(P \ {}_{\Sigma}\|_{Y} \ Q) \setminus Y$$

Usually the slave is named

$$P /\!/ m{:}Q$$

and all the master's communications with the slave use the name m, and it is m-labelled communications that are hidden.

Theory

As a derived operator, we could derive laws, etc. from the ones of the constituent operators.

For example the following is a sort of symmetry law: if $Y \cap Z = \{\}$ then

$$(P /\!\!/_Y Q) /\!\!/_Z R \;=\; (P /\!\!/_Z R) /\!\!/_Y Q \qquad \langle /\!\!/\text{-sym}\rangle$$

so, in particular, if m and n are distinct names then

$$(P /\!\!/ m{:}Q) /\!\!/ n{:}R \;=\; (P /\!\!/ n{:}R) /\!\!/ m{:}Q$$

Remote procedure calls

Enslavement is aften used when one process is providing a computational service to another: some sort of *remote procedure*.

A CPU with a separate FPU:

$$FPU \ = \ fpreq?(data, op) \rightarrow fpout!fpu(data, op) \rightarrow FPU$$

fpu defines the appropriate result for each floating-point calculation.

$$CPU \,/\!/\, fp{:}FPU$$

the CPU would then send requests like $fp.fpreq!((a, b), +)$, and could then carry out some more operations before it required the result of this combination, which it could get by inputting $fp.fpout?x$.

Process state

Recall that CSP treats identifiers declaratively (no assignment).

We can model an assignable state as an explicit parallel process.

A slave process representing a simple variable

$$\begin{array}{rcl} VAR(x) & = & assign?y \to VAR(y) \\ & & \Box\ read!x \to VAR(x) \end{array}$$

A process can be given one of these slaves for each variable, and it assigns to or reads from them by communication.

Sets via enslavement

Some of the most interesting examples use recursion and enslavement to build dynamic tree networks. For example

$$
\begin{array}{rcl}
SET & = & add?x \rightarrow (C_x /\!\!/ m{:}SET) \\
 & & \Box\ del?x \rightarrow SET \\
 & & \Box\ isin?x \rightarrow no \rightarrow SET, \quad \text{where} \\
 & & \\
C_x & = & add?y \rightarrow (C_x \not< y = x \not> m.add!y \rightarrow C_x) \\
 & & \Box\ del?y \rightarrow (E \not< y = x \not> m.del!y \rightarrow C_x) \\
 & & \Box\ isin?y \rightarrow (yes \rightarrow C_x \not< y = x \not> \\
 & & \qquad\qquad m.isin!y \rightarrow m?a : \{yes, no\} \rightarrow a \rightarrow C_x) \\
 & & \\
E & = & add?y \rightarrow m.del!y \rightarrow C_y \\
 & & \Box\ del?y \rightarrow m.del!y \rightarrow E \\
 & & \Box\ isin?y \rightarrow m.isin!y \rightarrow m?a : \{yes, no\} \rightarrow a \rightarrow E
\end{array}
$$

builds a (stack-like) chain, and two slaves would give a binary tree (see quicksort example in book).

A counter via enslavement

$$
\begin{array}{rcl}
Zero_{/\!/} & = & up \rightarrow (Succ /\!/ m{:}Zero_{/\!/}) \\
 & & \Box\ iszero \rightarrow Zero_{/\!/} \\
 & & \\
Succ & = & down \rightarrow (m.iszero \rightarrow NoSucc) \\
 & & \qquad\qquad\quad \Box\ m.down \rightarrow Succ) \\
 & & \Box\ \ up \rightarrow m.up \rightarrow Succ \\
 & & \\
NoSucc & = & iszero \rightarrow NoSucc \\
 & & \Box\ up \rightarrow Succ
\end{array}
$$

This is well-defined and is equivalent to the obvious sequential counter process with the $iszero$ event added to the zero state.

Problems with channel types

The piping and enslavement operators are not directly supported by machine-readable CSP because of the way channels work in CSP$_M$.

- The problem with $\gg$ is the special role given to $left$ and $right$, which must have only one type each in CSP$_M$ (in contrast to the layered protocol, for example).
- The problem with $/\!/$ is the extra name added to events: this would create complexities in the type system as it clashes with the idea that each channel name has a single type.

Link parallel

If (a1,b1), (a2,b2), ..., (an,bn) are pairs of channels where the two in each pair have the same type, then

```
P [a1 <-> b1, a2 <-> b2, ... , an <-> bn] Q
```

puts `P` and `Q` in parallel, joining the given pairs of channels together and and hiding them.

Thus $P \gg Q$ is `P [right <-> left] Q`.

While the above model of $\gg$ is, of course, associative, the link parallel operator is not, in general. Therefore CSP$_M$ insists on bracketing.

Enslavement with link parallel

The effect of enslavement can be achieved, provided process names move from the front of the event to a ‘data’ field:

```
(...(P [call.1 <-> call’, return.1 <-> return’]
                                          Slave)
       [call.2 <-> call’, return.2 <-> return’]
                                          Slave)
       ...
       [call.n <-> call’, return.n <-> return’]
                                          Slave)
```

Note how the same channels of the various copies of `Slave` are attached to different channels of `P`.