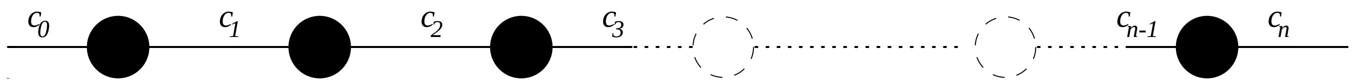
Why we need hiding

When we put processes in parallel it is often natural to regard the communications on internal channels as implementation detail. There is no good reason to distinguish between n $COPY'(c_r, c_{r+1})$ processes arranged



and a parameterised n-place buffer definition

$$BN(\langle\rangle) = c_0?x \to BN(\langle x\rangle)$$

$$\begin{aligned} BN(s\hat{}\langle a\rangle) = {} & c_n!a \to BN(s) \\ & \Box\ (c_0?x \to BN(\langle x\rangle\hat{}s\hat{}\langle a\rangle) \\ & \quad \lessdot \#s < n-1 \gtrdot STOP) \end{aligned}$$

We need to be able to *hide* the internal events $\{| c_1, \ldots, c_{n-1} |\}$.

The hiding operator

Whenever P is a process and $X \subseteq \Sigma$

$$P \setminus X$$

behaves like P except that events from X become τ (internal) actions.

The natural combination of parallel and hiding is then

$$(P\ {}_X\|_Y\ Q) \setminus (X \cap Y) \quad \text{or} \quad (P \underset{Z}{\|} Q) \setminus Z$$

CSP models of realistic parallel systems usually combine parallel and hiding in this way.

Some other languages contain the combination of parallel and hiding, but neither operator individually.

It's natural to hide the internal details of the ABP and some of the routing algorithms in Chapter 4.

Simple specification

We can even hide in Sudoku:

```
assert CHAOS({|select|}) [T= Puzzle
```

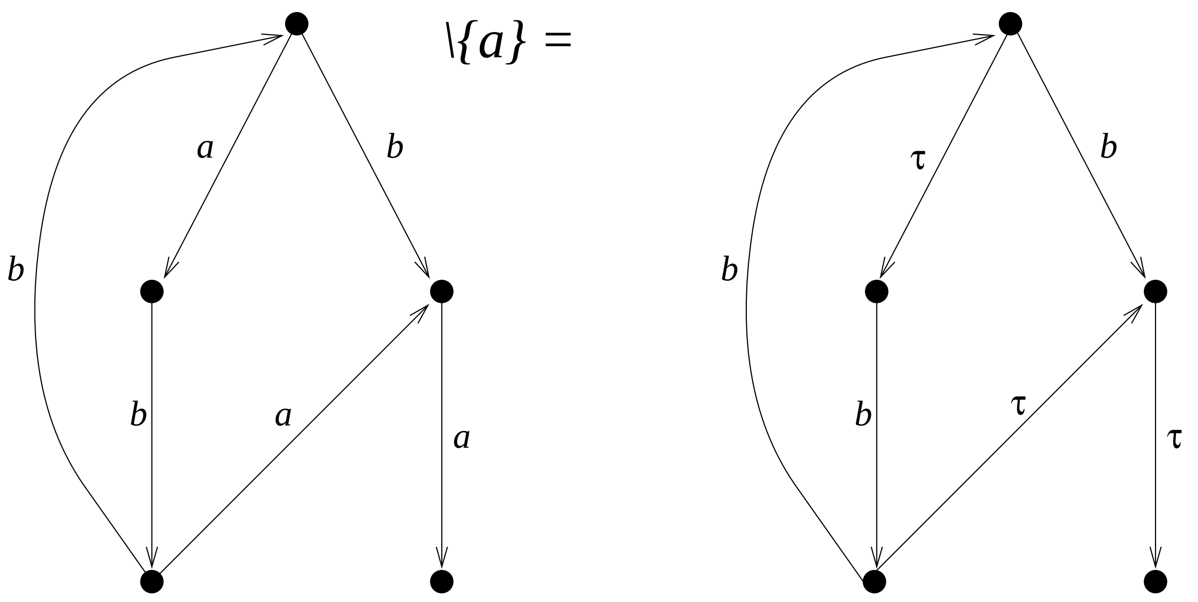
```
assert STOP [T= Puzzle\{|select|}
```

are equivalent:

the debugger can show you the moves.

Hiding in pictures

Hiding leaves the shape of the transition graph of any process alone, but turns appropriate labels into τ's:



If a τ action is enabled, then some action must happen quickly.

Thus no process ever gets stuck in a state with a τ.

States with and without τ's are respectively called *unstable* and *stable*.

Laws of hiding

This a one-place (unary) operator, so the laws look different:

$$(P \sqcap Q) \setminus X = (P \setminus X) \sqcap (Q \setminus X) \qquad \langle\text{hide-dist}\rangle$$

$$(P \setminus Y) \setminus X = (P \setminus X) \setminus Y \qquad \langle\text{hide-sym}\rangle$$

$$(P \setminus Y) \setminus X = P \setminus (X \cup Y) \qquad \langle\text{hide-combine}\rangle$$

$$P \setminus \{\} = P \qquad \langle\text{null hiding}\rangle$$

A prototype step law

$$(a \to P) \setminus X \;=\; \begin{cases} P \setminus X & \text{if } a \in X \\ a \to (P \setminus X) & \text{if } a \notin X \end{cases} \quad \langle\text{hide-step 1}\rangle$$

Note that the case $a \in X$ requires our principle that τ's are not delayable.

This is not a full step law because they apply to processes with arbitrary ranges of initial actions.

The difficult case is then

$$(?x : A \to P) \setminus X$$

when A has members both inside and outside X.

Hidden versus visible

How does

$$(a \to P \,\Box\, b \to Q) \setminus \{a\}$$

behave?

Our intuition about hiding in transition pictures, and the law ⟨hide-sym⟩:

$$((a \to P \,\Box\, b \to Q) \setminus \{a\}) \setminus \{b\} \;=$$
$$((a \to P \,\Box\, b \to Q) \setminus \{b\}) \setminus \{a\}$$

both imply that hiding a cannot completely exclude the b option.

Hidden and visible options

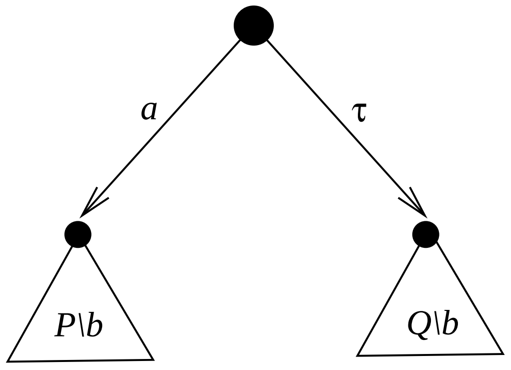
The τ action in

$$(a \to P \,\square\, b \to Q) \setminus \{b\}$$

will occur soon after the start if the a is not communicated quickly: the answer is

$$(a \to P \setminus \{b\}) \rhd Q \setminus \{b\}$$

"Sliding choice" or "Untimed timout".



The step law of hiding

$$(?x : A \to P) \setminus X \;=\; \qquad \langle\text{hide-step}\rangle$$

$$\begin{cases} ?x : A \to (P \setminus X) & \text{if } A \cap X = \{\} \\ \\ (?x : (A \smallsetminus X) \to (P \setminus X)) & \\ \triangleright \sqcap\{(P[a/x]) \setminus X \mid a \in A \cap X\} & \text{if } A \cap X \neq \{\} \end{cases}$$

The nature of hiding means that the ‘step’ of behaviour this law generates may be internal (τ) rather than a visible action.

Moving hiding around

As a network is built up, we can either hide the internal events at each stage, or wait until all processes are in parallel and do it then. It makes no difference, as shown by the following laws:

$$(P\ {}_X\|_Y\ Q) \setminus Z \ =$$

$$(P \setminus Z \cap X)\ {}_X\|_Y\ (Q \setminus Z \cap Y) \qquad \langle \text{hide-}{}_X\|_Y\text{-}$$

provided $X \cap Y \cap Z = \{\}$

$$(P \underset{X}{\|} Q) \setminus Z \ =$$

$$(P \setminus Z) \underset{X}{\|} (Q \setminus Z) \qquad \langle \text{hide-}\underset{X}{\|}\text{-}$$

provided $X \cap Z = \{\}$

These laws are important, because sometimes it is useful to leave all events visible till the end, and sometimes to hide as soon as possible.

Traces of hiding

The traces of $P \setminus X$ are very easy to compute: if we define $s \setminus X$, for any trace s, to be $s \upharpoonright (\Sigma \smallsetminus X)$, then

$$traces(P \setminus X) \ = \ \{s \setminus X \mid s \in traces(P)\}$$

For example,

$$traces(COUNT_0 \setminus \{up\}) = \{\langle down \rangle^n \mid n \in \mathbb{N}\}$$

$$traces(COUNT_0 \setminus \{down\}) = \{\langle up \rangle^n \mid n \in \mathbb{N}\}$$

Do you think these two results of hiding are really as similar as the above suggests?

Hiding and nondeterminism

Nondeterminism often appears where we conceal the internal details of parallel processes in contention for some resource.

$$P \;=\; a \to c \to STOP \;\Box\; b \to d \to STOP$$

offers the environment the choice between a and b, with subsequent behaviour depending on which option is chosen. If we give it the alphabet $X = \{a, b, c, d\}$ of all events it can perform,

$$N \;=\; P \;{}_X\|_{\{a,b\}}\; (a \to STOP \;{}_{\{a\}}\|_{\{b\}}\; b \to STOP)$$

does not change its behaviour.

But hiding the internal interactions of this network: $N \setminus \{a, b\}$ – produces the same behaviour as

$$(c \to STOP) \sqcap (d \to STOP)$$

Divergence

When internal events continue for ever we say that our process is *diverging*.

This can happen when a parallel/hiding combination can perform an infinite sequence of hidden internal actions without a visible one: this is *livelock*.

$\mathbf{div}$ is the process that does nothing but diverge. Externally it looks rather like $STOP$, but internally and theoretically it is different.

$COUNT_0 \setminus \{up\}$ can diverge, but $COUNT_0 \setminus \{down\}$ cannot.

Example

Imagine the *dithering philosophers*, in which a philosopher can put down the first fork without eating:

$$
\begin{aligned}
DITH_i = {} & picksup.i.i \rightarrow \\
& ((picksup.i.i{\oplus}1 \rightarrow eats.i \rightarrow \\
& putsdown.i.i{\oplus}1 \rightarrow putsdown.i.i \rightarrow DITH_i) \\
& \Box\ putsdown.i.i \rightarrow DITH_i)
\end{aligned}
$$

Replacing $PHIL_i$ by $DITH_i$ solves the deadlock problem of the dining philosophers, at the expense of introducing livelock if the $\{|\ picksup, putsdown\ |\}$ events are hidden.

Unbounded nondeterminism

Hiding an infinite set of events X, or the use of $\sqcap S$ for infinite S, can create *unbounded nondeterminism*. Our process might have an infinite number of possible options to pick via τ's from a single state.

Unbounded nondeterminism is sometimes regarded as unrealistic, and it creates some theoretical difficulties. While none of these is insuperable, you should at least be aware

(i) that this is an issue, and

(ii) which operators can create unbounded nondeterminism.

Hiding *versus* guardedness

For obvious reasons, when we recurse through hiding the notion of guardedness (essential for UFP) becomes a lot more difficult:

$$P = a \rightarrow (P \setminus \{a\})$$

is not guarded, and has many fixed points over $\mathcal{T}$.

No recursion in which $\setminus X$ is applied (directly or indirectly) to a recursive call should ever be deemed guarded without very careful justification. This also applies to operators we will see later which use hiding.

Therefore the recursions $X = F(X)$ we apply UFP to should not usually involve hiding, but there is nothing to stop the definition of the process Y using hiding, where Y is the one proved to be a fixed point of F. *This is quite common.*

Injective renaming

A function is 1–1, or *injective*, if $f(x) = f(y)$ implies $x = y$.

If $f : \Sigma \to \Sigma$ is injective, then the process

$$f[P]$$

which performs $f(a)$ whenever P performs a, is just a copy of P using different events.

f may be a *partial* function, so long as every event P uses is in its domain.

The transition system of $f[P]$ is just that of P with f applied to all visible events.

Injective $f[P]$ has many laws: see book.

Examples

If f swaps $down$ and up, then $f[COUNT_0]$ acts like a counter through the *negative* numbers.

Recall

$$COPY = left?x \to right!x \to COPY$$

$$COPY'(a,b) = a?x \to b!x \to COPY'(a,b)$$

If, for all x, $f(left.x) = a.x$ and $f(right.x) = b.x$ then

$$f[COPY] = COPY'(a,b)$$

Similar maps would relate the behaviours of the different $FORK_i$ processes and the different $PHIL_i$ processes, because these two collections both behave identically except for the names of events.

Of course, no f exists such that $f[FORK_i] = PHIL_j$.

Non-injective functions

Uses of renaming in building realistic implementation models are usually injective functions (see “link parallel”), but we can achieve powerful and useful effects by examining the concept of “alphabet transformations” more generally.

The first case is $f[P]$, where f is a non-injective function: some events we could distinguish in P get identified in $f[P]$.

There is no difficulty in seeing how this generalisation works, but obviously it allows subtler effects.

Forgetful renaming

For example, it allows us to forget some details of a process:

$$SPLIT = in?x : T \to \\ ((out1.x \to SPLIT) \lessdot x \in S \gtrdot (out2.x \to SPLIT))$$

To forget about the contents of the messages, apply $forget$ which remembers only the channel name. $forget[SPLIT]$ is equivalent to

$$SPLIT' = in \to (out1 \to SPLIT' \sqcap out2 \to SPLIT')$$

Nondeterminism appears because we have lost the detail which allows to decide which way to send each message.

Why forget?

The transformation on the previous slide appears odd, but

- it allows us to demonstrate that some aspect of the correctness of the system does not depend on precisely how decisions are made, and
- in cases where this is true the details of decision making frequently clutter proofs.
- It will often save states on FDR.

Like hiding, forgetful renaming is a form of *abstraction*.

When $f^{-1}(x)$ ($= \{y \in \Sigma \mid f(y) = x\}$) is infinite (e.g. because the types of forgotten data are) you have to be particularly careful because $f[P]$ might introduce unbounded nondeterminism.

Relations

The function in $f[P]$ can only map each event of P to a single target.

It is surprisingly often useful to map events to several alternatives.

This can be done using *inverse functions*: see Hoare's book, or (more generally) using *relations*. We discuss the latter because it is what CSP$_M$ (in essence) uses.

A relation between X and Y is a set of pairs $(x, y) \in X \times Y$.

If R is a relation and $(x, y) \in R$ we write $x\, R\, y$.

If R is a relation, its *domain* and *range* are respectively

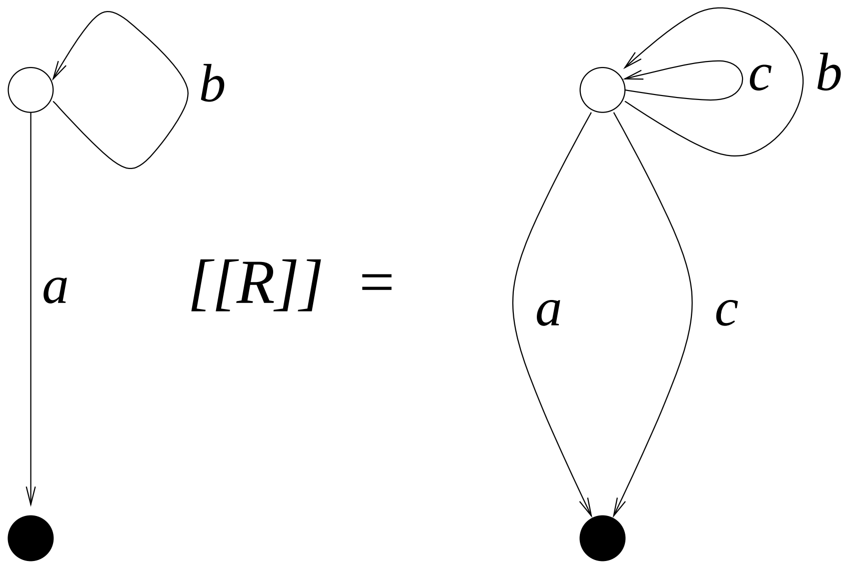
$$dom(R) = \{x \mid \exists\, y . x\, R\, y\}$$
$$ran(R) = \{y \mid \exists\, x . x\, R\, y\}$$

Relational renaming

We are interested in relations R between Σ and itself whose domain includes all the events of P.

$P[\![R]\!]$ is a process which, when P performs a and $a\,R\,b$, can perform b.

If a has several images under R, then each arc labelled a in P's transition diagram gets multiplied in $P[\![R]\!]$'s: if $R = \{(a,a),(a,c),(b,b),(b,c)\}$ then



Laws of renaming

Relational renaming is the most general, so we give laws and traces rules for it:

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

$$(P \,\Box\, Q)[\![R]\!] \;=\; P[\![R]\!] \,\Box\, Q[\![R]\!] \qquad \langle [\![R]\!]\text{-}\Box\text{-dist}\rangle$$

If the initials of P' are A, then those of $P'[\![R]\!]$ are $R(A) = \{y \mid \exists\, x \in A.\, (x, y) \in R\}$:

$$(?x : A \to P)[\![R]\!] \;=\; \qquad \langle [\![R]\!]\text{-step}\rangle$$
$$?y : R(A) \to \bigsqcap\{(P[z/x])[\![R]\!] \mid z \in A \wedge z\, R\, y\}$$

Notice how this last law makes the introduction of nondeterminism clear.

Traces of $P[\![R]\!]$

We just apply R in the obvious way to traces:

$$\langle a_1, \ldots a_n\rangle R^* \langle b_1, \ldots, b_m\rangle \Leftrightarrow n = m \wedge \forall\, i \leq n . a_i R\, b_i$$

$$traces(P[\![R]\!]) = \{t \mid \exists\, s \in traces(P) . s\, R^* t\}$$

Notation

We can use a substitution-like notation:

- $P[\![{}^{a}/{b}]\!]$ means b is replaced by a, and everything else (including a) is unchanged.
- $P[\![{}^{a,\,b}/{b,\,a}]\!]$ means a and b are swapped.
- $P[\![{}^{b,\,c}/{a,\,a}]\!]$ means a is mapped to both b and c.
- $P[\![{}^{a,\,b}/{a,\,a}]\!]$ means a is mapped to both a and b.

The CSP$_M$ notation is essentially the same (though order of events reversed):

```
P[[b <- a]]  P[[b <- a, a <- b]]  P[[a <- b, b <-c]]
```

A magic wand

One-to-many renaming combined with parallel composition can achieve weird and wonderful effects such as variable renaming: for example

$$P[\![^{\mathit{Braeburn},\, \mathit{Cox}}/_{\mathit{apple},\, \mathit{apple}}]\!] \underset{\{|\mathit{Adam}, \mathit{Eve}, \mathit{Braeburn}, \mathit{Cox}|\}}{\|} \mathit{Reg}$$

$$\begin{aligned}
\mathit{Reg} = {} & \mathit{Braeburn} \to \mathit{Reg} \\
& \Box\ \mathit{Adam} \to \mathit{Reg} \\
& \Box\ \mathit{Eve} \to \mathit{Reg}'
\end{aligned}$$

$$\begin{aligned}
\mathit{Reg}' = {} & \mathit{Cox} \to \mathit{Reg}' \\
& \Box\ \mathit{Adam} \to \mathit{Reg} \\
& \Box\ \mathit{Eve} \to \mathit{Reg}'
\end{aligned}$$

Study the book for many more...

Link parallel

Suppose a and b are two channels with the same type. Then

$P[a \leftrightarrow b]Q$ connects a in P to b in Q and hides the result.

It can be defined in terms of hiding and renaming:

$$(P[\![^c/a]\!] \underset{\{|c|\}}{\|} Q[\![^c/b]\!]) \setminus \{|\ c\ |\}$$

Like connecting two wires.

Notation extends to multiple channel pairs:

$$P[a \leftrightarrow b, d \leftrightarrow d, ...]Q$$

Previous presentations of CSP have concentrated on two special cases of this:

Piping or chaining

Written $P \gg Q$, this was equivalent to $P[right \leftrightarrow left]Q$.

Connects the output of P to the input of Q, using a standard pair of named channels.

Problem: doesn't combine well with typed channels. What happens if we want to use $\gg$ for channels with different types?

Enslavement

$P /\!/ Q$ means $(P \underset{\alpha Q}{\|} Q) \setminus \alpha Q$

It was often used in cases where Q was given a name:

$P /\!/ a : Q$

so P's communications with Q get an extra label a.

Does not fit particularly well with CSP$_M$'s channel naming conventions.

Recursive enslavement: Mergesort

$$\begin{aligned}
M &= in.end \to out.end \to M \\
&\quad \Box\ in?x : T \to M1(x) \\
M1(x) &= in.end \to out!x \to out.end \to M \\
&\quad \Box\ in?y : T \to \\
&\qquad ((upto!x \to downto!y \to Mu)\,[upto \leftrightarrow in, upfrom \leftrightarrow out]\,M) \\
&\qquad [downto \leftrightarrow in, downfrom \leftrightarrow out]\,M)
\end{aligned}$$

Mu is in "inputting" mode where next input is to be sent to the up slave; Md the same for the $down$ slave.

$$
\begin{aligned}
Mu &= in.end \to downto.end \to upto.end \to O1 \\
&\quad \Box\ in?y : T \to upto!x \to Md
\end{aligned}
$$

$$
\begin{aligned}
Md &= in.end \to downto.end \to upto.end \to O1 \\
&\quad \Box\ in?y : T \to downto!x \to Mu
\end{aligned}
$$

$$
O1 = upfrom?x \to downfrom?y \to O2(x, y)
$$

$$
\begin{aligned}
O2(x, y) &= x = end \wedge y = end \& out!end \to Mu \\
&\quad x = end \wedge y \neq end \& out!y \to downin?y' \to O2(x, y') \\
&\quad x \neq end \wedge y = end \& out!x \to upin?x' \to O2(x', y) \\
&\quad x \neq end \wedge y \neq end \& (out!x \to upin?x' \to O2(x', y) \\
&\qquad\qquad\qquad\qquad \lessdot x < y \gtrdot \\
&\qquad\qquad\qquad\qquad out!y \to downin?y' \to O2(x, y'))
\end{aligned}
$$