What is CSP?

CSP is a language for processes which interact, both with us and with other processes that may be in parallel with them.

Communicating **S**equential **P**rocesses was invented by Tony Hoare in the mid to late 1970's, in two distinct versions:

- *(CACM, 1978)* A simple imperative programming language with communication between named, sequential components.
- *(Various papers leading up to Hoare's 1985 book)* A rather abstract looking notation in which virtually everything had disappeared except for constructs directly related to communication.

Superficially, these have little in common except for the idea of communication.

The evolution occurred because concurrency (the study of processes running and communicating with each other) proved to be so different and interesting.

This course is about the theoretical second version of CSP, but it provides an ideal basis for understanding what goes on in concurrent systems built in other ways.

What does this course cover?

- CSP language.
- Basic mathematics of CSP.
- Understanding concurrency: nondeterminism, deadlock etc.
- Specifying and implementing systems.
- Tools for exploring and proving things about CSP processes.

Reading List

Course text: *The Theory and Practice of Concurrency*, A.W. Roscoe, Prentice-Hall 1998. Chapters 1–6 (plus parts of the appendices).

(All exercises are drawn from here.)

Further Reading:

- *Communicating Sequential Processes*, C.A.R. Hoare, Prentice-Hall 1985.
- *Concurrent and Real-Time Systems: the CSP approach*, S.A. Schneider, Wiley 2000(!).
- Documentation for FDR and ProBE (the tools we will be using).

Fundamental concepts

We are in a world of interaction.

A process might be part of a distributed computing network, part of a microprocessor, a vending machine or ATM, a piece of a railway or telephone network.

A CSP process is completely described by the ways in which it can communicate with its external environment.

The most important first step is therefore to choose the *alphabet* of communications: an appropriate set of atomic interactions for the world we are modelling.

The alphabet of all events is conventionally written Σ (Sigma).

In the abstracted world of CSP, these are assumed to be instantaneous: the instant when an interaction is agreed.

Example alphabets

- $\{up, down, iszero\}$ simple counter
- $\{in.x, out.x \mid x \in T\}$ a unit that inputs and outputs values (of type T) on one channel each.
- $\{pay.x, change.x \mid x \in M\} \cup$ $\{cheddar.w, gouda.w, parmesan.w, \ldots \mid w \in W\}$ *(M is set of money amounts, W is set of weights)* a cheese shop.

$STOP$

The simplest process is the one that does nothing:

$STOP$

performs no actions at all.

It may seem useless, but we will find that it is useful in programs and specifications, and also provides a simple model of a *deadlocked* system:

Any process, however complex its construction, that can do nothing is equivalent to $STOP$.

Prefixing

If P is a process and $a \in \Sigma$ is any communication

$$a \rightarrow P$$

offers a until the environment accepts it, and then behaves like P.

So we can build processes like

$$up \rightarrow down \rightarrow STOP$$

$$\begin{aligned} & pay.\pounds 5 \rightarrow gouda.500g \rightarrow \\ & \qquad cheddar.1lb \rightarrow change.\pounds 1.23 \rightarrow STOP \end{aligned}$$

Recursion

If we define processes which can return to their initial states, they can communicate for ever:

$$
\begin{aligned}
P_1 &= up \rightarrow down \rightarrow P_1 \\
P_2 &= up \rightarrow down \rightarrow up \rightarrow down \rightarrow P_2 \\
P_u &= up \rightarrow P_d \\
P_d &= down \rightarrow P_u
\end{aligned}
$$

In general, if $F(\cdot)$ is an operator from processes to processes, we can define a recursive process by an equation $P = F(P)$ or in-line $\mu\, P.F(P)$.

Both create a process which runs the context $F(P)$, such that each occurrence of the name P behaves the same as the whole process.

Mutual recursions, like P_u and P_d, define several processes in terms of each other.

But all we can build so far are processes that go through a single sequence of communications...

Choice

If $A \subseteq \Sigma$ is any set of events and $P(a)$ is a process for each $a \in A$, then

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

offers its environment the choice of A and then behaves like the appropriate $P(a)$. For example:

$$\begin{aligned} RUN_A &= ?x : A \to RUN_A \\ REPEAT &= ?x : \Sigma \to x \to REPEAT \end{aligned}$$

(RUN_A is a process to remember, since it is one of the standard simple processes like $STOP$.)

If $A = \{\}$, then $?x : A \to P(x) = STOP$ and $RUN_A = STOP$.

Guarded alternative

Where the different $P(x)$'s are not so conveniently similar we can write:

$$(a \rightarrow P(a) \mid b \rightarrow P(b) \mid \ldots \mid z \rightarrow P(z))$$

For example

$$\begin{array}{rcl} COUNT_0 & = & up \rightarrow COUNT_1 \\ COUNT_{n+1} & = & (up \rightarrow COUNT_{n+2} \\ & & \mid\ down \rightarrow COUNT_n) \end{array}$$

We can clearly create any state machine where each state offers a list of actions with successor states: rather like a menu structure on a computer.

Channels

An event in Σ consists of a channel name plus zero or more data components: up, $cheddar.1lb$, $send.a.b.m$.

In *machine-readable* CSP each channel must be declared with its data type

```
channel up, down
channel cheddar, gouda, parmesan: Weights
channel send, receive: User.User.Message
```

It is always a good idea to follow the type discipline this implies, even when not using the machine readable version.

The data components are sometimes used to transmit data between processes, and sometimes to create arrays of channels.

Input and output

Often a process will want to allow either all

$$c?x \to P(x)$$

or some

$$c?x : A \to P(x)$$

of the communications on one of its channels. These are just forms of $?y : B \to P(y)$ that have the effect of allowing a process to *input* along a channel.

When outputs happen on channels, we often write them $c!x \to P$ rather than $c.x \to P$, though these are nearly synonymous.

(The only differences appear when inputs and outputs are mixed in the same communication: $c?x!y$ and $c?x.y$ have different meanings....see book!)

Example

Provided they are on distinct channels, we allow channel inputs and outputs in the guarded alternative construct:

$$
\begin{aligned}
CS(0) &= pay?x \rightarrow CS(x) \\
CS(x) &= (cheddar?w : \{z \in W \mid z \times V_C \leq x\} \rightarrow CS(x - w \times V_C) \\
&\quad \mid\ gouda?w : \{z \in W \mid z \times V_G \leq x\} \rightarrow CS(x - w \times V_G) \\
&\quad \mid\ parmesan?w : \{z \in W \mid z \times V_P \leq x\} \rightarrow CS(x - w \times V_P) \\
&\quad \mid\ pay?y \rightarrow CS(x + y) \\
&\quad \mid\ change!x \rightarrow CS(0))
\end{aligned}
$$

Note the use of process parameters to maintain and calculate process state.

Parameters appear in a variety of places depending on the programmer's taste: $P(x, y)$, $P^{x,y}$, $P_y(x)$ etc., but are always treated *declaratively* (i.e., you can't assign to them): programming the data components of CSP systems is essentially *functional* programming.

Two buffer processes

Buffers are processes that act as loss-free, order-preserving handlers of data:

$$COPY \;=\; left?x : T \rightarrow right!x \rightarrow COPY$$

is a one-place buffer, and the infinite parameterised mutual recursion

$$\begin{array}{rcl} B^{\infty}_{\langle\rangle} & = & left?x : T \rightarrow B^{\infty}_{\langle x\rangle} \\ B^{\infty}_{s\hat{}\langle y\rangle} & = & (left?x : T \rightarrow B^{\infty}_{\langle x\rangle\hat{}s\hat{}\langle y\rangle} \\ & & \mid right!y \rightarrow B^{\infty}_{s}) \end{array}$$

can hold any number of items.

External choice operator

The guarded alternative operator is a bit clumsy because it uses an arbitrary length list of explicitly guarded (by communication) processes.

The *external choice* operator $\Box$ "generalises" it:

$$P \Box Q$$

offers the environment the choice between the initial actions of P and Q, and behaves like the one whose action is picked.

Every guarded alternative can be replaced by $\Box$ (one in place of each "|" in the text).

Note that this is a larger change to the syntax structure than it is to the written text.

External *versus* guarded choice

Almost all practical uses of $\Box$ are in places where guarded alternative would have worked (i.e. between disjointly guarded processes), but

- we always use the $\Box$ form anyway, and
- we have to understand what $P \Box Q$ means in the general case.

It is best to think of guarded alternative as being a stepping-stone to understanding $\Box$*, rather than having a proper place in the language.*

If $A \cap B = \{\}$, it is obvious that

$$(?x : A \to P(x)) \Box (?x : B \to Q(x)) = ?x : A \cup B \to R(x)$$

where $R(x)$ is $P(x)$ or $Q(x)$ depending on whether x is in A or B.

But what happens when $A \cap B \neq \{\}$?

Ambiguity in $P \,\square\, Q$

What happens if the environment selects an initial event common to P and Q?

The implementation doesn't have the information to know which way to jump.

It would be asymmetric and give extra overhead to make it prioritised: e.g., we would have to wait to make sure P could not do an event before allowing Q to act.

The answer, therefore, is that the choice is *nondeterministic*: the process is free to pick either path and the environment has no control over which. For example, in

$$(a \to b \to STOP) \,\square\, (a \to c \to STOP)$$

the environment cannot control whether it is offered b or c after a.

Nondeterministic choice

Unpredictable behaviour arises naturally in various ways in CSP, and we need to be able to reason about it cleanly.

Therefore there is a *nondeterministic choice* operator. $P \sqcap Q$ can behave like P or like Q as it wishes: the environment has no control. (It must, however, pick one.)

$$(a \to b \to STOP) \,\square\, (a \to c \to STOP)$$
$$a \to (b \to STOP \sqcap c \to STOP)$$

If S is a nonempty set of processes, then $\bigsqcap S$ can choose to behave like any member of S:

$$\bigsqcap\{P_1, \ldots, P_n\} = P_1 \sqcap \ldots \sqcap P_n$$

Though these operators are not ones we would often expect to find in programs representing complete implementations, we will see that they have many uses.

$Chaos_A$

RUN_A has no nondeterminism: it always offers the whole of A.

Here are two other processes with the same traces but rather different behaviour:

- $Chaos_A = STOP \sqcap ?x : A \to Chaos_A$
  The process that can accept or refuse any element of A.
- $DF_A = \bigsqcap\{a \to DF_A \mid a \in A\}$
  Can always select any member of A, but *must* select one. This is the most nondeterministic *deadlock free* process over A.

Refinement

P refines Q if every behaviour of P is one of Q. It is equivalent to

$$P \sqcap Q = Q$$

and written

$$P \sqsupseteq Q$$

You can never tell, using P, that you are not in fact dealing with Q.

Anywhere that Q works, P is guaranteed to work as well, so P represents an improvement (not necessarily proper) on Q.

Refinement will prove to be exceptionally important throughout this course!

Conditional choice

$\Box$ and $\sqcap$ are both ways of selecting between pairs of alternatives, neither of which is found in "ordinary" programming languages.

There is also a way that just about all languages have:

$$\mathit{if}\ b\ \mathit{then}\ P\ \mathit{else}\ Q$$

has the obvious meaning, and can be written equivalently as

$$P \between b \between Q$$

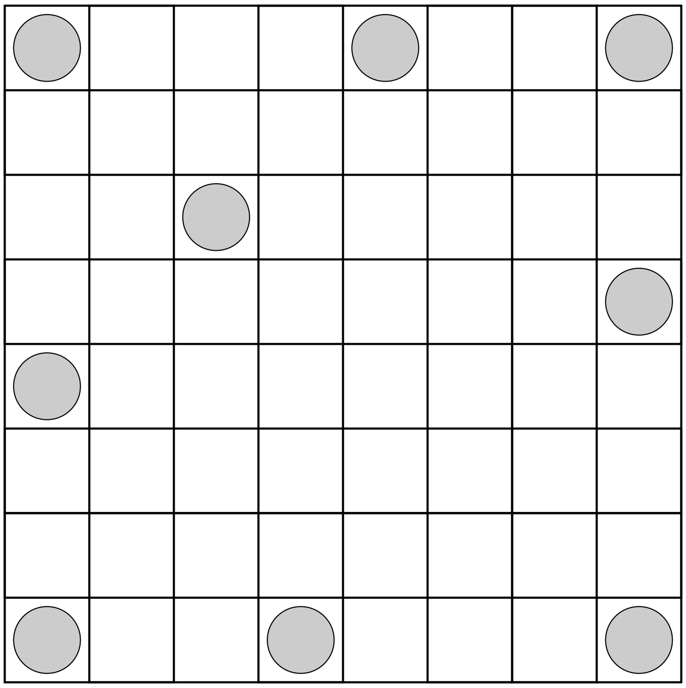
(designed to be an infix operator like $\Box$ and $\sqcap$).

b will depend on parameters and input values.

Example

$$
\begin{array}{lll}
\multicolumn{3}{l}{Counter(n,m) \ =} \\
 & (down & \rightarrow Counter(n-1,m)) \lessdot n > 0 \gtrdot STOP \\
\Box & (up & \rightarrow Counter(n+1,m)) \lessdot n < 7 \gtrdot STOP \\
\Box & (left & \rightarrow Counter(n,m-1)) \lessdot m > 0 \gtrdot STOP \\
\Box & (right & \rightarrow Counter(n,m+1)) \lessdot m < 7 \gtrdot STOP
\end{array}
$$

Note the way $\Box$ and $STOP$ are combined to collapse 9 different cases to 1:



Machine readable CSP

For full details see Appendix B.

A CSP script mixes process and non-process definitions, plus channel and type declarations, comments etc: it is a functional program with processes (and finite sets).

Written in ASCII, so operators look slightly different:
`->  []  |~|`

Guarded alternative not supported, and non-channel inputs $?x : A \to P(x)$ can only be written using *indexed* external choice:

```
[] x:A @ x -> P(x)
```

$\bigsqcap\{P(x) \mid x \in I\}$ is similar: `|~| x:I @ P(x)`

Conditionals achieved by `if..then..else..` and pattern matching, with

```
if b then P else STOP
```

abbreviated `b & P` (see *Counter* definition earlier).

ProBE

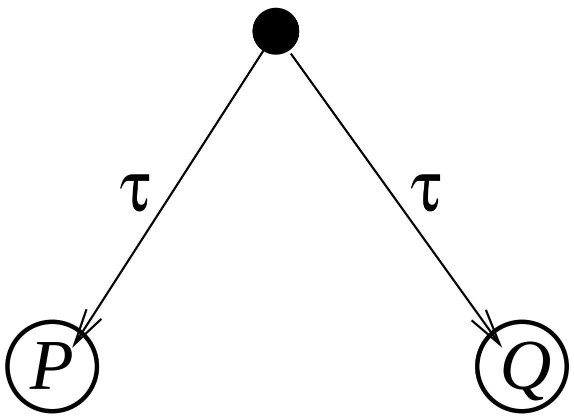
Process **B**ehaviour **E**xplorer.

A tool that allows us to explore the actions of a CSP process, by giving a choice of communications, and calculating the process that results from the one we pick, and so on.....

It is a CSP *animator*, or *simulator*.

Good for helping you understand what you have written, and seeing if it behaves as you expected!

Note that nondeterministic choice `P|~|Q` is implemented via `tau` (τ) actions, which are *internal* and *invisible* to the environment.



Operational semantics

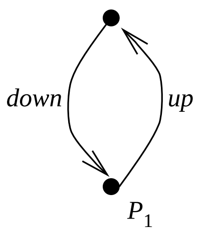
ProBE actually calculates the *operational semantics* of a process P: the graph

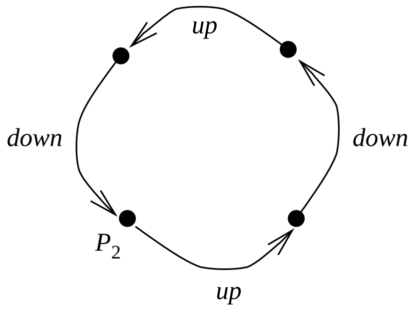
- whose root is P,
- whose nodes are CSP terms that P can become after some actions, and
- where the edges from each node Q are the initial actions (τ or in Σ) that Q can perform.

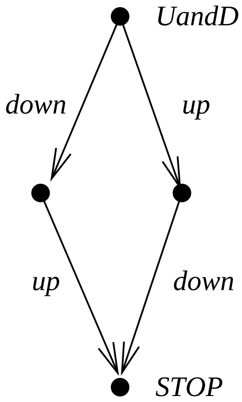
In this way a process is identified with a picture of its behaviour as a simple *state machine*, *automaton* or *labelled transition system*.

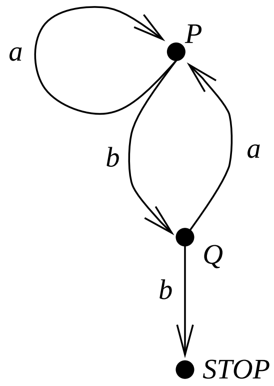
The formalities of this can be studied in Chapter 7, but in this course we will be using these pictures only as an *informal* aid to understanding CSP.

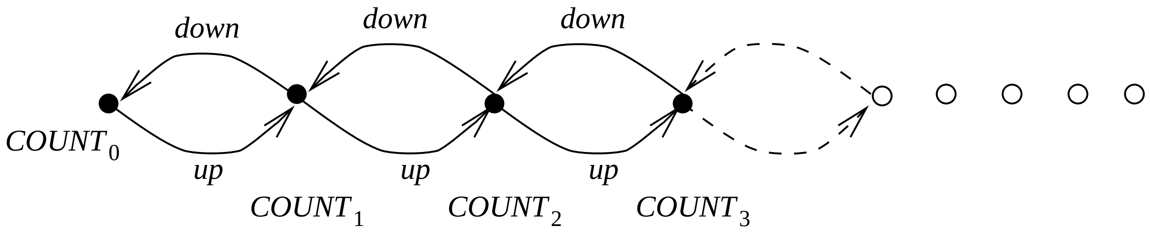
Examples


down
up
P_1


up
down
down
P_2
up


UandD
down
up
up
down
STOP


a
P
b
a
Q
b
STOP


down
down
down
$COUNT_0$
up
up
up
$COUNT_1$
$COUNT_2$
$COUNT_3$

The mathematics of CSP

There are three main ways of formalising what CSP means:

- The operational semantics discussed earlier.
- The algebraic properties of operators like $\square$ and $\sqcap$.
- Identifying a process with the set of its behaviours.

While each of these can tell you something by itself, they give a more complete picture together and help to explain each other.

Algebra

Here are some laws we are all familiar with:

$x + y$	$=$	$y + x$	*commutative*, or *symmetry*
$x \times y$	$=$	$y \times x$	ditto
$x \cup y$	$=$	$y \cup x$	ditto
$(x + y) + z$	$=$	$x + (y + z)$	*associativity*
$(x + y) \times z$	$=$	$(x \times z) + (y \times z)$	(right) *distributive* law
$0 + x$	$=$	x	*unit* law
$\{\} \cap x$	$=$	$\{\}$	*zero* law
$x \cup x$	$=$	x	*idempotence*

What similar laws do CSP operators satisfy?

Laws of choice

The choice between P and P is no choice at all:

$$P \,\Box\, P = P \qquad \langle\Box\text{-idem}\rangle$$

$$P \sqcap P = P \qquad \langle\sqcap\text{-idem}\rangle$$

The choice between P and Q is the same as that between Q and P:

$$P \,\Box\, Q = Q \,\Box\, P \qquad \langle\Box\text{-sym}\rangle$$

$$P \sqcap Q = Q \sqcap P \qquad \langle\sqcap\text{-sym}\rangle$$

And the choice between P, Q and R is the same however bracketed:

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

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

These *idempotence*, *symmetry* and *associative* laws are just what are needed for $\bigsqcap S$ to make sense: see exercises.

Distributivity

If $F(\cdot)$ is a CSP construct, what is the difference between

$$F(P \sqcap Q) \quad \text{and} \quad F(P) \sqcap F(Q)?$$

If F doesn't run more than one copy of its argument, *none*.

There is then no way of telling whether the choice was made before or after applying F.

Just about all individual CSP operators satisfy this principle, and therefore have *distributive* laws over $\sqcap$ (and also $\bigsqcap$):

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

$$P \;\Box\; \bigsqcap S \;=\; \bigsqcap\{P \;\Box\; Q \mid Q \in S\} \qquad \langle\Box\text{-Dist}\rangle$$

$$a \to (P \sqcap Q) \;=\; (a \to P) \sqcap (a \to Q) \qquad \langle\text{prefix-dist}\rangle$$

$$\begin{array}{l} ?x : A \to (P \sqcap Q) \;=\; \\ \quad (?x : A \to P) \sqcap (?x : A \to Q) \end{array} \qquad \langle\text{input-dist}\rangle$$

While there are distributive laws over other operators, "distributivity" unqualified always means over $\sqcap$ and $\bigsqcap$.

When distributivity fails

Operators that run their arguments more than once are usually not distributive, for example recursion:

$$\mu\, p.((a \to p) \sqcap (b \to p)) \;\neq$$
$$(\mu\, p.a \to p) \sqcap (\mu\, p.b \to p)$$

This is because in $F(P \sqcap Q)$, the two copies may behave differently, while in $F(P) \sqcap F(Q)$ they must behave alike.

Distribution the other way

Note that in set theory $\cup$ and $\cap$ each distribute over the other.

In CSP there is a similar phenomenon with $\square$ and $\sqcap$. For we have not only $\langle\square$-dist$\rangle$, but also

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

Both processes have the same options after the first visible action (obviously), and both can refuse exactly the same sets of actions on the first step.

This is a theoretically important law, but not one that gets used much in practice!

A step law

Step laws allow us to calculate the first step actions of a process, and are therefore central to our understanding of CSP:

The step law of $\Box$ is

$$\begin{array}{ll} (?x : A \to P) \,\Box\, (?x : B \to Q) \;= & \\ \quad ?x : A \cup B \to \;\; ((P \sqcap Q) & \\ \qquad\qquad \lessdot x \in A \cap B \gtrdot & \langle \Box\text{-step} \rangle \\ \qquad\qquad (P \lessdot x \in A \gtrdot Q)) & \end{array}$$

Note how this simply formalises what we said about $\Box$ before.

All operators other than prefix-choice, $\sqcap$ and recursion will have a step law.

In essence the behaviour of any new operator can be understood via its distribution properties and its step law. To see how this works have a look at Chapter 11, though this is not for the mathematically faint-hearted!

More laws

The laws of $STOP$:

$$STOP \; = ?x : \{\} \rightarrow P \qquad \langle STOP\text{-step}\rangle$$

$$STOP \;\Box\; P \;=\; P \qquad \langle \Box\text{-unit}\rangle$$

The law of recursion:

$$\mu\, p.P \;=\; P[\mu\, p.P/p] \qquad \langle \mu\text{-unwind}\rangle$$

See the book for laws of $\lessdot b \gtrdot$.

What are laws for?

- Provide intuition and understanding to us.
- Give sanity tests for any proposed mathematical theory/semantice.
- Can be used to prove processes equal to each other.
- With more effort, provide a complete *algebraic* semantics for the language (see Chapter 11).

Traces

A *trace* is a sequence of visible communications that a process might communicate: a process's history.

In general it might be finite or infinite (the latter being the history over an infinite time) but we will consider only the finite case in this course.

In any case every *prefix* (initial subsequence) of an infinite trace is a finite one.

$traces(P)$ is the set of all P's (finite) traces, and is necessarily a *nonempty*, *prefix-closed* subset of Σ^* (the finite sequences formed from Σ).

If P and Q are two processes such that $traces(P) = traces(Q)$, then we write $P =_T Q$ and say they are *trace equivalent*.

The set of all such subsets of Σ^* is called the *traces model* and written $\mathcal{T}$. This provides an important tool for understanding and specifying processes.

Trace notation

- $\langle a_1, a_2, \ldots, a_n \rangle$ is the sequence containing a_1, a_2 to a_n in that order. Note that $\langle a, a, b \rangle$, $\langle a, b \rangle$ and $\langle b, a \rangle$ are all different.
- $\langle \rangle$ is the empty sequence.
- $s\hat{\ }t$ is the *concatenation* of s and t: $\langle a, b \rangle\hat{\ }\langle b, a \rangle = \langle a, b, b, a \rangle$.
- If $t = s\hat{\ }w$, then s is a *prefix* of t, written $s \leq t$ (a partial order).

Working out $traces(P)$

There is a rule for each CSP operator that shows the effect it has on traces:

- $traces(STOP) = \{\langle\rangle\}$
- $traces(a \to P) = \{\langle\rangle\} \cup \{\langle a\rangle\hat{~}s \mid s \in traces(P)\}$ – this process has either done nothing, or its first event was a followed by a trace of P.
- $traces(?x : A \to P) = \{\langle\rangle\} \cup \{\langle a\rangle\hat{~}s \mid a \in A \wedge s \in traces(P[a/x])\}$ – this is similar: $P[a/x]$ means the substitution of the value a for all free occurrences of the identifier x.

More clauses of $traces(P)$

- $traces(c?x : A \to P) = \{\langle\rangle\} \cup \{\langle c.a\rangle \hat{}\, s \mid a \in A \wedge s \in traces(P[a/x])\}$ – the same except for the use of the channel name.
- $traces(P \,\Box\, Q) = traces(P) \cup traces(Q)$ – this process offers the traces of P and those of Q.
- $traces(P \sqcap Q) = traces(P) \cup traces(Q)$ – since this process can behave like either P or Q.
- $traces(\bigsqcap S) = \bigcup\{traces(P) \mid P \in S\}$
- $traces(P \mathrel{\lessdot} b \mathrel{\gtrdot} Q) = traces(P)$ if b evaluates to $true$; and $traces(Q)$ if b evaluates to $false$.

Note that the traces of $P \sqcap Q$ and $P \,\Box\, Q$ are the same: this strongly suggests that traces do not give a *complete* description of processes.

Recursion

The recursion $p = Q$ (or equivalently $\mu\, p.Q$) must (thanks to $\langle\mu$-unwind$\rangle$) satisfy

$$traces(\mu\, p.Q) = traces(Q[\mu\, p.Q/p])$$

In other words, $traces(\mu\, p.Q)$ is a value Y satisfying $Y = F(Y)$, where $F(X)$ is the traces of Q when a process with traces X is substituted for p in Q.

For example, if Q is $a \rightarrow p$,

$$F(X) = \{\langle\rangle\} \cup \{\langle a\rangle\hat{\ } s \mid s \in X\}$$

Of course, not all functions from $\mathcal{T}$ to $\mathcal{T}$ have a fixed point, just as $x \mapsto x + 1$ has none over the natural numbers, **BUT**

all CSP definable functions have a fixed point, and a *least* one (that is a subset of all others), and that is the correct value for recursions.

For example, the fixed point of $F(p) = a \rightarrow p$ is

$$\{\langle\rangle, \langle a\rangle, \langle a, a\rangle, \langle a, a, a\rangle, \ldots\}$$

which is obviously the right answer.

Mutual recursion

The mathematics of mutual recursion is the same, except that instead of having functions from $\mathcal{T}$ to $\mathcal{T}$ we now have ones from $\mathcal{T}^\Lambda$ to itself where Λ is an *indexing* set with one member for each mutually defined process. $\mathcal{T}^\Lambda$ is the set of *vectors* of members of $\mathcal{T}$ indexed by Λ.

Λ may be finite: in

$$\begin{array}{rcl} P & = & (a \to P) \square (b \to Q) \\ Q & = & (c \to Q) \square (b \to P) \end{array}$$

Λ has size 2.

Or it may be infinite: in $COUNT$ we have $\Lambda = \mathbb{N}$ and the function operates on infinite vectors of trace-sets.

The value of a mutual recursion is the vector $\underline{X} \in \mathcal{T}^\Lambda$ which is the least fixed point of the function from $\mathcal{T}^\Lambda$ to itself the recursion generates. (The one with fewest traces in each component.)

Fixed point theories

Two different ways of looking at this: partial orders and metric spaces.

Partial orders give more generality (work for all recursions), while metric spaces only work for *guarded* ones, but the metric theory gives the most useful proof rules.

Full details can be found in Appendix A and Section 8.2; here we will give an overview of both.

Monotonicity and continuity

A function of $\mathcal{T}$ is *monotonic* if $P \subseteq Q$ implies $F(P) \subseteq F(Q)$, and *continuous* if $P_i \subseteq P_{i+1}$ for all i implies

$$F(\bigcup_{i=0}^{\infty} P_i) = \bigcup_{i=0}^{\infty} F(P_i)$$

Any distributive operator has both these properties. $P \subseteq Q$ implies $P \sqcap Q = Q$ (in traces) and so

$$F(Q) = F(P \sqcap Q) = F(P) \sqcap F(Q)$$

Hence monotone, and the definition of continuity is just restricted distributivity (identifying $\cup$ and $\sqcap$).

Thus all the individual CSP operators have these properties, and since it can be shown that compositions of monotone (continuous) operators are monotone (continuous), *all* CSP terms have these properties over $\mathcal{T}$.

(See Appendix A for detailed results, and how to handle the construct of recursion itself here. In fact, monotonicity is implied by continuity.)

Tarski's theorem

Many versions of this (see Appendix A), but the one we need is that if $(\mathcal{X}, \leq)$ is a partial order in which (a) there is a least element $\bot$ and (b) each chain $P_0 \leq P_1 \leq P_2 \leq \ldots$ has a least upper bound $\bigsqcup_{i=0}^{n} P_i$, then every continuous function has a least fixed point given by

$$\bigsqcup\{f^n(\bot) \mid n \in \mathbb{N}\}$$

Since $(\mathcal{T}, \subseteq)$ has both these properties (least element being $STOP$), it follows that every CSP definable single recursion $\mu\, p.F(p)$ has traces

$$\bigcup_{i=0}^{\infty} F^n(STOP)$$

and that the value of a mutual recursion is given by

$$\bigcup_{i=0}^{\infty} F^n(\underline{STOP})$$

where $\underline{STOP}$ is an appropriate vector of $STOP$'s and $\bigcup$ is componentwise union on the vectors.

Guarded recursions

A recursion is *guarded* if every recursive call is preceded by a communication, either directly

$$P = (a \to P) \square (b \to P)$$

or indirectly

$$Q = a \to (Q \square b \to Q)$$

(We will have to refine this definition a bit when more operators are introduced later.)

In practice, very nearly all sensible recursions are guarded.

Intuitively, if we want to know the first n steps of the behaviour of a guarded recursion $\mu\, p.F(p)$, all we have to do is unwind the recursion n times:

$$F^n(\mu\, p.F(p))$$

This is what lies behind the *metric* theory of recursion, and the principle of *Unique Fixed Points* (UFP):

If $P = F(P)$ is a guarded recursion (perhaps mutual), and Q is a process (or vector) such that $Q =_T F(Q)$ (in traces) then $P =_T Q$.

A simple metric

If $P \in \mathcal{T}$, its n-place restriction $P \downarrow n$ is just $\{s \in P \mid \#s \leq n\}$: the traces of length n or less.

We can define a metric (rather odd compared to the ones you may have seen in mathematics courses) over $\mathcal{T}$ by

$$\begin{array}{rcl} d(P,P) & = & 0, \quad \text{and otherwise} \\ d(P,Q) & = & 2^{-n}, \quad n \text{ maximal such that } P \downarrow n = Q \downarrow n \end{array}$$

Thus the longer it takes to tell P and Q apart, the closer they are.

This makes $\mathcal{T}$ into a *complete metric space* (i.e., if P_i is a sequence such that for any $\epsilon > 0$ we can find m with all P_i, $i > m$ within ϵ of each other, then P_i converges to a limit P').

Any guarded recursion corresponds to a *contraction mapping* over $\mathcal{T}$: $d(F(P), F(Q)) \leq d(P,Q)/2$, and so has a unique fixed point by the Banach contraction mapping theorem (see Appendix A).

All the above is easily modified to take account of mutual recursions.

Using UFP

Here are some trivial examples of the unique fixed point principle. Recall the recursions:

$$P_1 \quad = \quad up \rightarrow down \rightarrow P_1$$

$$P_2 \quad = \quad up \rightarrow down \rightarrow up \rightarrow down \rightarrow P_2$$

$$\begin{aligned} P_u &= up \rightarrow P_d \\ P_d &= down \rightarrow P_u \end{aligned}$$

Using $\langle\mu$-unwind$\rangle$ twice, it is easy to see that P_1 satisfies the definition of P_2. UFP (applied to the guarded P_2 recursion) then proves them equivalent.

Similarly, unwinding P_u twice shows that it satisfies the definition of P_1, proving them equivalent via UFP on the P_1 recursion.

Thus P_1, P_2 and P_u are all equivalent (which is hardly surprising).

Mutual UFP

Most interesting uses of UFP seem to be on mutual recursions (usually one-step tail recursions where we are defining one process for each state a system can get into). The following process is an integer counter process

$$\begin{aligned} ZCOUNT_n &= up \rightarrow ZCOUNT_{n+1} \\ &\quad \Box\, down \rightarrow ZCOUNT_{n-1} \end{aligned}$$

A bit of thought tells you that the index is actually irrelevant here: we might suspect that all $ZCOUNT_n$ behave like

$$AROUND = up \rightarrow AROUND \,\Box\, down \rightarrow AROUND$$

This can be *proved* by UFP considering the vector of processes $\underline{A}$ (one component for each integer) with each component $AROUND$.

If F_{ZC} is the function of the $ZCOUNT$ recursion, $F_{ZC}(\underline{A}) = \underline{A}$:

$$\begin{aligned} (F_{ZC}(\underline{A}))_n &= up \rightarrow A_{n+1} \,\Box\, down \rightarrow A_{n-1} \\ &= up \rightarrow AROUND \,\Box\, down \rightarrow AROUND \\ &= AROUND \;=\; A_n \end{aligned}$$

Traces and laws

Note that every law we state implies that the trace sets of the two sides are always equal. For example, it must always be true (thanks to ⟨prefix-dist⟩) that

$$a \to (P \sqcap Q) =_T (a \to P) \sqcap (a \to Q)$$

Since we can work out both sides in terms of $traces(P)$ and $traces(Q)$ using the trace semantics, this provides a test of our semantics that could, in principle, go wrong.

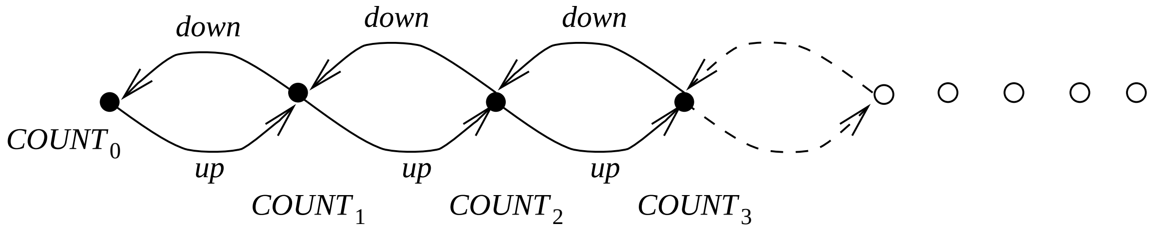
However in each case it is fairly easy to show that the implied result really is true, for example both sides above reduce to

$$\{\langle\rangle\} \cup \{\langle a\rangle\hat{\ }s \mid s \in traces(P) \cup traces(Q)\}$$

Such results are boring to prove, but worth the effort if you can avoid stating a false law.

Traces and pictures

Evidently $traces(P)$ can also be computed from the LTS (transition graph) of P: possible sequences of actions, ignoring τ's.



This creates both an opportunity:

perhaps an easier way to work out traces (it is how tools tend to do it);

and another obligation:

to show that the traces got this way are the same as those calculated the other way.

Fortunately the second of these only becomes an issue when we have given a formal operational semantics *and in this course we do not do this*!

So what you should do for now is to realise that $traces(P)$ and the LTS of P are related, and know that the related congruence results can be proved.

Trace specifications

Traces allow us to formulate many useful specifications of processes.

A *behavioural trace specification* asserts some property of each trace of the process P: if R is a condition on traces,

$$P \ \textbf{sat} \ R(tr) \quad \text{means} \quad \forall\, tr \in traces(P).R(tr)$$

(tr is the identifier conventionally used to represent an arbitrary trace.)

$R(tr)$ is usually expressed in some combination of predicate logic and trace notation.

In practice, just about all useful specifications on $traces(P)$ are lifted from individual trace specifications in this way.

Trace specifications are very good at limiting behaviour, but do nothing to force it: note that P and $P \sqcap STOP$ have exactly the same traces.

More trace notation

- If s is a finite sequence, $\#s$ denotes the *length* of s.
- If $s \in \Sigma^*$ and $A \subseteq \Sigma$ then $s \upharpoonright A$ means the sequence s *restricted* to A: the sequence whose members are those of s which are in A. $\langle\rangle \upharpoonright A = \langle\rangle$ and $(s\hat{\ }\langle a\rangle) \upharpoonright A = (s \upharpoonright A)\hat{\ }\langle a\rangle$ if $a \in A$, $s \upharpoonright A$ otherwise.
- If $s \in \Sigma^*$ then $s \downarrow c$ can mean two things depending on what c is.
  - If c is an *event* in Σ then it means the number of times c appears in s (i.e., $\#(s \upharpoonright \{c\})$).
  - If c is a *channel name* (associated with a non-trivial data type) it means the sequence of values (without the label c) that have been communicated along c in s. For example

$$\langle c.1, d.1, c.2, c.3, e.4\rangle \downarrow c = \langle 1, 2, 3\rangle$$

Example trace specifications

- P_1, P_2 and P_u all satisfy

$$tr \downarrow down \leq tr \downarrow up \leq tr \downarrow down + 1$$

- The specification of $COUNT_n$ is similar but less restrictive:

$$tr \downarrow down \leq tr \downarrow up + n$$

- $B^{\infty}_{\langle\rangle}$ and $COPY$ both satisfy the basic *buffer* specification:

$$tr \downarrow right \leq tr \downarrow left$$

Proof rules for sat

In Hoare's book the main way of handling specifications like these is through a series of proof rules such as:

$$STOP \ \textsf{sat} \ (tr = \langle\rangle)$$

$$\frac{\forall\, a \in A.P(a) \ \textsf{sat} \ R_a(tr)}{?\,a : A \rightarrow P \ \textsf{sat} \ (tr = \langle\rangle \vee \exists\, a \in A.\, \exists\, tr'.\ tr = \langle a\rangle\hat{}\,tr' \wedge R_a(tr'))}$$

$$\frac{P \ \textsf{sat} \ R(tr) \wedge Q \ \textsf{sat} \ R(tr)}{P \ \Box \ Q \ \textsf{sat} \ R(tr)}$$

$$\frac{P \ \textsf{sat} \ R(tr) \wedge Q \ \textsf{sat} \ R(tr)}{P \sqcap Q \ \textsf{sat} \ R(tr)}$$

which essentially translate the trace semantic rules into logic, and general inferences such as:

$$\frac{P \ \textsf{sat} \ R(tr) \wedge \forall\, tr.R(tr) \Rightarrow R'(tr)}{P \ \textsf{sat} \ R'(tr)}$$

$$\frac{P \ \textsf{sat} \ R(tr) \wedge P \ \textsf{sat} \ R'(tr)}{P \ \textsf{sat} \ R(tr) \wedge R'(tr)}$$

Proof rule for recursion

Just as a recursive program calls upon itself, the proof rule has a "circular" feel to it. It is a form of *fixed point induction*. The following is the single recursion form:

Suppose $\mu\, P.F(P)$ *is a recursive definition, and that* X *is the (least) fixed point which it defines in* $\mathcal{T}$*. Suppose* $R(tr)$ *is a trace specification such that:*

- $STOP$ **sat** $R(tr)$, *and*
- Y **sat** $R(tr) \Rightarrow F(Y)$ **sat** $R(tr)$

then X **sat** $R(tr)$.

In other words, if the recursion preserves the specification, and is satisfied by $STOP$, then it is satisfied by the fixed point.

$STOP$ and trace specifications

The requirement that $STOP$ **sat** $R(tr)$ seems uncomfortable, since $STOP$ is a pretty useless process, and so you would not expect it to satisfy many sensible specifications.

But $traces(STOP) \subseteq traces(P)$ for *all* P, so

$$P \textbf{ sat } R(tr) \Rightarrow STOP \textbf{ sat } R(tr)$$

In other words, *any* behavioural trace specification satisfied by *any* process is satisfied by $STOP$.

This type of specification can ban a process from *doing* any incorrect action, but it cannot ban it from *not doing* anything.

This is another illustration of the inadequacy of traces as a complete description of processes.

Nevertheless, perhaps a majority of practical specifications applied to CSP processes are pure trace specifications.

Example

To prove $\mu\, p.a \rightarrow b \rightarrow p$ **sat** $tr \downarrow b \leq tr \downarrow a$.

- $STOP$ **sat** $tr = \langle\rangle$ and $tr = \langle\rangle \Rightarrow tr \downarrow b \leq tr \downarrow a$.

  Hence $STOP$ **sat** $tr \downarrow b \leq tr \downarrow a$.
- Assume P **sat** $tr \downarrow b \leq tr \downarrow a$. Then
  - $b \rightarrow P$ **sat** $tr = \langle\rangle \vee tr = \langle b\rangle\hat{}tr' \wedge$
    $tr' \downarrow b \leq tr' \downarrow a$

    Hence $b \rightarrow P$ **sat** $tr \downarrow b \leq tr \downarrow a + 1$.
  - $a \rightarrow (b \rightarrow P)$ **sat** $tr = \langle\rangle \vee tr = \langle a\rangle\hat{}tr' \wedge$
    $tr' \downarrow b \leq tr' \downarrow a + 1$

    Hence $a \rightarrow b \rightarrow P$ **sat** $tr \downarrow b \leq tr \downarrow a$.
- Thus $\mu\, p.a \rightarrow b \rightarrow p$ **sat** $tr \downarrow b \leq tr \downarrow a$.

Trace refinement

Recall that $P \sqsupseteq Q$ if $P \sqcap Q = Q$. We can interpret this in traces as *trace refinement*:

$$P \sqsupseteq_T Q \equiv traces(P) \subseteq traces(Q)$$

(Note the reversed containment: the *more* traces a process has, the *less* refined it is.)

Three vital facts about refinement:

1. $P \sqsubseteq_T P' \wedge P' \sqsubseteq_T P'' \Rightarrow P \sqsubseteq_T P''$ *Transitivity*
2. $P \sqsubseteq_T P' \Rightarrow C[P] \sqsubseteq_T C[P']$ for any process context $C[\cdot]$ (syntax with a slot to insert a process). *Monotonicity*
3. $P \sqsubseteq_T P'$ and P **sat** $R(tr)$ implies P' **sat** $R(tr)$.

Together these justify *step-wise* and *compositional* development.

The last one helps to explain the use of reverse containment.

Characteristic processes

Let R be any specification such that $STOP$ **sat** $R(tr)$.

If $P \in S \Rightarrow P$ **sat** $R(tr)$, then $(\bigsqcap S)$ **sat** $R(tr)$ because $t \in traces(\bigsqcap S) \Rightarrow \exists\, P \in S . t \in traces(P)$.

It follows that $P_R = \bigsqcap\{P \mid P \textbf{ sat } R(tr)\}$ is the most nondeterministic process satisfying $R(tr)$:

$$P \sqsupseteq_T P_R \Leftrightarrow P \textbf{ sat } R(tr)$$

Thus satisfaction of any behavioural trace specification (other than $false$) can be tested by refinement against its *characteristic process*.

Of course we can use any process that is trace equivalent to $\bigsqcap\{P \mid P \textbf{ sat } R(tr)\}$. (This is just as well.)

The FDR refinement checker is the main tool we have for proving that CSP processes meet specifications. The fact that it does this by answering the question $P \sqsubseteq Q$? makes finding efficient representations of characteristic processes a very important skill.

Examples of characteristic processes over $\mathcal{T}$

- RUN_A The process never communicates outside A.
- $B^{\infty}_{\langle\rangle}$ The traces buffer specification

  $$Buff(tr) \equiv tr \in (\{\mid left, right \mid\}^* \wedge tr \downarrow right \leq tr \downarrow left$$

- $COPY$ The one-place buffer specification

  $$Buff(tr) \wedge \#(tr \downarrow left) \leq \#(tr \downarrow right) + 1$$

Using FDR

Traces checks are inserted into files thus:

```
assert Spec [T= Impl
```

In order to run, both specification and implementation must be *finite state*. Namely, their transition pictures (operational semantics) must be finite.

Thus $COUNT_0$ and $B^{\infty}_{\langle\rangle}$ may not be used.

The easiest way to violate this rule is (as in these two examples) via an infinite parameter type.

When a check fails, you can get debugging information back (a trace that the implementation performed which violates the specification).

Afters and initials

If P is any process, $initials(P)$ (sometimes written P^0) is the set of all its initial events

$$initials(P) = \{a \mid \langle a \rangle \in traces(P)\}$$

For example, $initials(STOP) = \{\}$ and $initials(?x : A \to P(x)) = A$.

If $s \in traces(P)$ then P/s ('P *after* s') is P after the trace s is complete. Over the traces model, P/s can be computed

$$traces(P/s) = \{t \mid s\hat{\ }t \in traces(P)\}$$

P/s is not an ordinary part of the CSP language, because it is not implementable:

$$(STOP \sqcap a \to P)/\langle a \rangle = P$$

but the process on the left hand side cannot be forced to accept the a.

It is used for discussing and describing behaviour, e.g.

$$(P \,\Box\, Q)/s = P/s \quad \text{if } s \in traces(P) \setminus traces(Q)$$