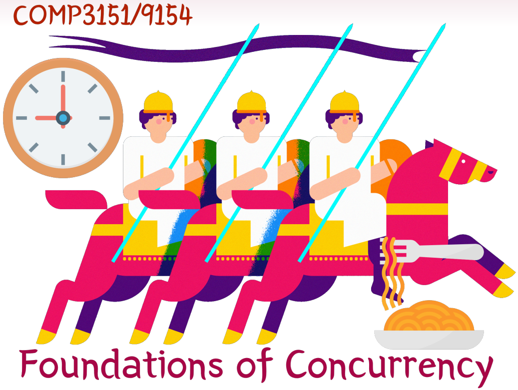


The Calculus of Communicating Systems

Johannes Åman Pohjola
UNSW
Term 2 2022

Where we are at

Last lecture, we studied *compositional* proof techniques.

This lecture, we'll take a brief detour into the world of *process algebra*, a high level formalism for describing concurrent systems.

Many of the resources for this lecture were borrowed from Graham Hutton.

CCS

The *Calculus of Communicating Systems*:

- Is a *process algebra*, a simple formal language to describe concurrent systems.

CCS

The *Calculus of Communicating Systems*:

- Is a *process algebra*, a simple formal language to describe concurrent systems.
- Is given semantics in terms of *labelled transition systems*.

CCS

The *Calculus of Communicating Systems*:

- Is a *process algebra*, a simple formal language to describe concurrent systems.
- Is given semantics in terms of *labelled transition systems*.
- Was developed by Turing-award winner Robin Milner in the 1980s.
- Has an abstract view of synchronization that applies well to message passing.

CCS

The *Calculus of Communicating Systems*:

- Is a *process algebra*, a simple formal language to describe concurrent systems.
- Is given semantics in terms of *labelled transition systems*.
- Was developed by Turing-award winner Robin Milner in the 1980s.
- Has an abstract view of synchronization that applies well to message passing.

Why do we learn this?

This gives us a symbolic way to describe our transition diagrams, and reason about them algebraically rather than diagramatically.

Processes

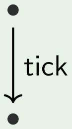
Processes in CCS are defined by equations:

Example

The equation:

$$\mathbf{CLOCK} = \mathrm{tick}$$

defines a process **CLOCK** that simply executes the *action* “tick” and then terminates. This process corresponds to the first location in this *labelled transition system* (LTS):



An LTS is like a transition diagram, save that our transitions are just abstract actions and we have no initial or final location.

Action Prefixing

Example

$$\mathbf{CLOCK}_2 = \text{tick.tock}$$

defines a process called $\mathbf{CLOCK}_2$ that executes the action “tick” then the action “tock” and then terminates.

$$\bullet \xrightarrow{\text{tick}} \bullet \xrightarrow{\text{tock}} \bullet$$

The process:

$$\mathbf{CLOCK}_3 = \text{tock.tick}$$

has the same actions as $\mathbf{CLOCK}_2$ but arranges them in another order.

Action Prefixing

Example

$$\textbf{CLOCK}_2 = \text{tick.tock}$$

defines a process called $\textbf{CLOCK}_2$ that executes the action "tick" then the action "tock" and then terminates.

$$\bullet \xrightarrow{\text{tick}} \bullet \xrightarrow{\text{tock}} \bullet$$

The process:

$$\textbf{CLOCK}_3 = \text{tock.tick}$$

has the same actions as $\textbf{CLOCK}_2$ but arranges them in another order.

Definition

If a is an action and P is a process, then $x.P$ is a process that executes x before P. This brackets to the right, so:

$$x.y.z.P = x.(y.(z.P))$$

Action Prefixing

Example

$$\textbf{CLOCK}_2 = \text{tick.tock}$$

defines a process called $\textbf{CLOCK}_2$ that executes the action “tick” then the action “tock” and then terminates.

$$\bullet \xrightarrow{\text{tick}} \bullet \xrightarrow{\text{tock}} \bullet$$

The process:

$$\textbf{CLOCK}_3 = \text{tock.tick}$$

has the same actions as $\textbf{CLOCK}_2$ but arranges them in another order.

Definition

If a is an action and P is a process, then $x.P$ is a process that executes x before P. This brackets to the right, so:

$$x.y.z.P = x.(y.(z.P))$$

Stopping

More precisely, we should write:

$$\textbf{CLOCK}_2 = \text{tick.tock.}\textbf{STOP}$$

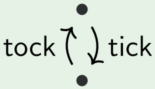
where **STOP** is the trivial process with no transitions.

Loops

Up to now, all processes make a finite number of transitions and then terminate. Processes that can make a infinite number of transitions can be pictured by allowing loops:

Example (Loops)

= the process that diverges executing "tick" transitions



= the process that alternates "tick" and "tock" forever

We accomplish loops in CCS using recursion.

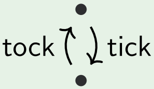
Loops

Up to now, all processes make a finite number of transitions and then terminate. Processes that can make a infinite number of transitions can be pictured by allowing loops:

Example (Loops)

= the process that diverges executing "tick" transitions

$$\mathbf{CLOCK}_4 = \text{tick}.\mathbf{CLOCK}_4$$



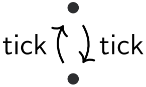
= the process that alternates "tick" and "tock" forever

$$\mathbf{CLOCK}_5 = \text{tick}.\text{tock}.\mathbf{CLOCK}_5$$

We accomplish loops in CCS using recursion.

Equality of Processes

These two processes are physically different:



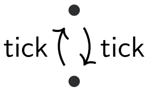
$\mathbf{CLOCK}_4 = \mathrm{tick}.\mathbf{CLOCK}_4$

$\mathbf{CLOCK}_6 = \mathrm{tick}.\mathrm{tick}.\mathbf{CLOCK}_6$

Equality of Processes

These two processes are physically different:

$$\mathbf{CLOCK}_4 = \text{tick}.\mathbf{CLOCK}_4$$

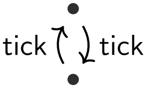


$$\mathbf{CLOCK}_6 = \text{tick}.\text{tick}.\mathbf{CLOCK}_6$$

But they both have the same behaviour — an infinite sequence of “tick” transitions.

Equality of Processes

These two processes are physically different:



$$\mathbf{CLOCK}_4 = \mathrm{tick}.\mathbf{CLOCK}_4 \qquad \mathbf{CLOCK}_6 = \mathrm{tick}.\mathrm{tick}.\mathbf{CLOCK}_6$$

But they both have the same behaviour — an infinite sequence of “tick” transitions.

Informal Definition

We consider two process to be equal if an external observer cannot distinguish them by their actions.
We will refine this definition later.

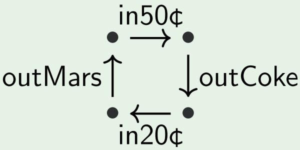
A Simple Vending Machine

Vending Machines are very common examples for process algebra.

Example (An inflexible machine)

Suppose I define my vending machine as:

$$\mathbf{VM}_1 = \text{in50¢.outCoke.in20¢.outMars}.\mathbf{VM}_1$$



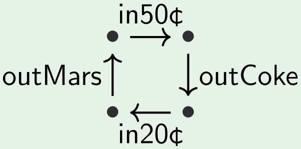
A Simple Vending Machine

Vending Machines are very common examples for process algebra.

Example (An inflexible machine)

Suppose I define my vending machine as:

$$\mathbf{VM}_1 = \mathrm{in50¢}.\mathrm{outCoke}.\mathrm{in20¢}.\mathrm{outMars}.\mathbf{VM}_1$$



This machine is not very flexible:

- It only accepts exact money.
- The customer has no choice: The machine dispenses Coke and Mars bars alternately.

Choice

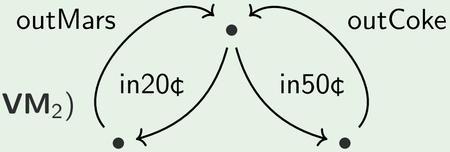
To make a more flexible kind of vending machine, we need a (nondeterministic) choice operator.

Choice

To make a more flexible kind of vending machine, we need a (nondeterministic) choice operator.

Example

$$\mathbf{VM}_2 = (\text{in50¢.outCoke.}\mathbf{VM}_2) + (\text{in20¢.outMars.}\mathbf{VM}_2)$$



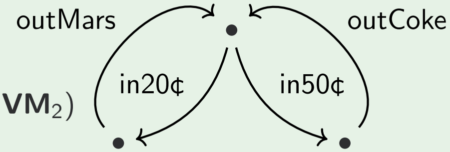
Here we have a process $\mathbf{VM}_2$ that repeatedly either inputs 50¢ and outputs a coke, or inputs 20¢ and outputs a mars bar.

Choice

To make a more flexible kind of vending machine, we need a (nondeterministic) choice operator.

Example

$$\mathbf{VM}_2 = (\text{in50¢}.\text{outCoke}.\mathbf{VM}_2) + (\text{in20¢}.\text{outMars}.\mathbf{VM}_2)$$



Here we have a process $\mathbf{VM}_2$ that repeatedly either inputs 50¢ and outputs a coke, or inputs 20¢ and outputs a mars bar.

Definition

If P and Q are processes then $P + Q$ is a process which can either behave as the process P or the process Q.

Choice Equalities

Observe that we have the following identities about choice:

$$P + (Q + R) \quad = \quad (P + Q) + R \qquad\qquad \text{(associativity)}$$

Choice Equalities

Observe that we have the following identities about choice:

$$\begin{aligned} P + (Q + R) &= (P + Q) + R && \text{(associativity)} \\ P + Q &= Q + P && \text{(commutativity)} \end{aligned}$$

Choice Equalities

Observe that we have the following identities about choice:

$$
\begin{aligned}
P + (Q + R) &= (P + Q) + R && \text{(associativity)}\\
P + Q &= Q + P && \text{(commutativity)}\\
P + \mathbf{STOP} &= P && \text{(neutral element)}
\end{aligned}
$$

Choice Equalities

Observe that we have the following identities about choice:

$$\begin{array}{lcll} P + (Q + R) & = & (P + Q) + R & \text{(associativity)} \\ P + Q & = & Q + P & \text{(commutativity)} \\ P + \mathbf{STOP} & = & P & \text{(neutral element)} \\ P + P & = & P & \text{(idempotence)} \end{array}$$

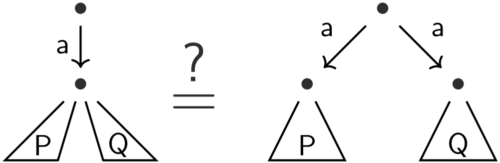
Choice Equalities

Observe that we have the following identities about choice:

$$\begin{array}{lcll} P + (Q + R) & = & (P + Q) + R & \text{(associativity)} \\ P + Q & = & Q + P & \text{(commutativity)} \\ P + \mathbf{STOP} & = & P & \text{(neutral element)} \\ P + P & = & P & \text{(idempotence)} \end{array}$$

What about the equation:

$$a.(P + Q) \quad \overset{?}{=} \quad (a.P) + (a.Q)$$

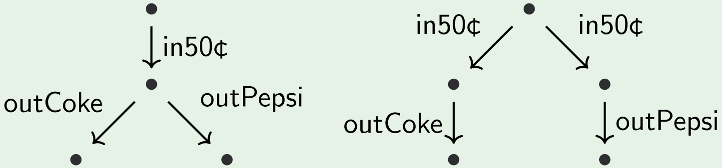


Branching Time

Example

$$
\begin{aligned}
\mathbf{VM}_3 &= \mathsf{in50¢.(outCoke + outPepsi)} \\
\mathbf{VM}_4 &= \mathsf{(in50¢.outCoke) + (in50¢.outPepsi)}
\end{aligned}
$$

Or in pictures:

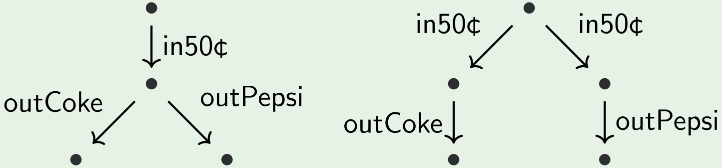


Branching Time

Example

$$\begin{aligned} \mathbf{VM}_3 &= \mathsf{in50¢}.(\mathsf{outCoke} + \mathsf{outPepsi}) \\ \mathbf{VM}_4 &= (\mathsf{in50¢}.\mathsf{outCoke}) + (\mathsf{in50¢}.\mathsf{outPepsi}) \end{aligned}$$

Or in pictures:



Reactive Systems

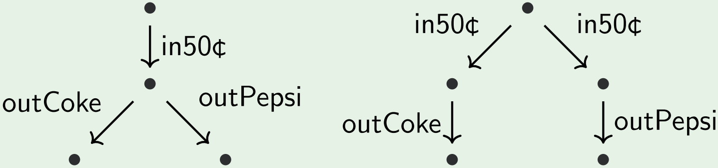
$\mathbf{VM}_3$ allows the customer to choose which drink to vend after inserting 50¢. In $\mathbf{VM}_4$ however, the machine makes the choice when the customer inserts a coin.

Branching Time

Example

$$\begin{aligned}\mathbf{VM}_3 &= \mathsf{in50¢.(outCoke + outPepsi)}\\ \mathbf{VM}_4 &= \mathsf{(in50¢.outCoke) + (in50¢.outPepsi)}\end{aligned}$$

Or in pictures:



Reactive Systems

VM$_3$ allows the customer to choose which drink to vend after inserting 50¢. In **VM**$_4$ however, the machine makes the choice when the customer inserts a coin.
They are **different** in this *reactive* view, but they have the same behaviours!

Equivalences

The equation

$$a.(P + Q) = (a.P) + (a.Q)$$

is usually not admitted for this reason.

Equivalences

The equation

$$a.(P+Q) = (a.P) + (a.Q)$$

is usually not admitted for this reason.

Exercise

It is possible to construct two processes that are equal assuming this equation but do not have the same set of behaviours (and thus do not satisfy the same LTL properties).

Equivalences

The equation

$$a.(P + Q) = (a.P) + (a.Q)$$

is usually not admitted for this reason.

Exercise

It is possible to construct two processes that are equal assuming this equation but do not have the same set of behaviours (and thus do not satisfy the same LTL properties).

If we do admit it, then our notion of equality is very coarse (it is called partial trace equivalence). This is enough if we want to prove safety properties, but progress is not guaranteed. Johannes: Explain why

Equivalences

The equation

$$a.(P + Q) = (a.P) + (a.Q)$$

is usually not admitted for this reason.

Exercise

It is possible to construct two processes that are equal assuming this equation but do not have the same set of behaviours (and thus do not satisfy the same LTL properties).

If we do admit it, then our notion of equality is very coarse (it is called partial trace equivalence). This is enough if we want to prove safety properties, but progress is not guaranteed. Johannes: Explain why

Terminology

Our notion of equality without this equation is called (strong) bisimilarity.

Exercises

- A clock that can stop at any time.

Exercises

- A clock that can stop at any time.
- A clock that ticks or tocks at each cycle.

Exercises

- A clock that can stop at any time.
- A clock that ticks or tocks at each cycle.
- A clock that ticks each cycle or tocks each cycle.

Exercises

- A clock that can stop at any time.
- A clock that ticks or tocks at each cycle.
- A clock that ticks each cycle or tocks each cycle.
- A vending machine for Mars and Coke that gives change.

Parallel Composition

Definition

If P and Q are processes then $P \mid Q$ is the parallel composition of their processes — i.e. the non-deterministic interleaving of their actions.

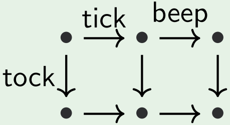
Parallel Composition

Definition

If P and Q are processes then $P \mid Q$ is the parallel composition of their processes — i.e. the non-deterministic interleaving of their actions.

Example (Clocks)

$$\mathbf{ACLOCK} = \mathsf{tick}.\mathsf{beep} \mid \mathsf{tock}$$



$$\mathbf{CCLOCK} = \mathbf{TICLK} \mid \mathbf{TOCLK}$$

$$\mathbf{TICLK} = \mathsf{tick}.\mathbf{TICLK}$$

$$\mathbf{TOCLK} = \mathsf{tock}.\mathbf{TOCLK}$$

Exercise: Express these processes without parallel composition.

Synchronization

In CCS, every action a has an opposing *coaction* $\overline{a}$ (and $\overline{\overline{a}} = a$):

Actions:	tick	tock	$\mathsf{in50¢}$	$\mathsf{outCoke}$	$\ldots$
Coactions:	$\overline{\mathsf{tick}}$	$\overline{\mathsf{tock}}$	$\overline{\mathsf{in50¢}}$	$\overline{\mathsf{outCoke}}$	$\ldots$

Synchronization

In CCS, every action a has an opposing *coaction* $\overline{a}$ (and $\overline{\overline{a}} = a$):

Actions:	tick	tock	in50¢	outCoke	...
Coactions:	$\overline{\text{tick}}$	$\overline{\text{tock}}$	$\overline{\text{in50¢}}$	$\overline{\text{outCoke}}$	...

It is a convention to think of an action as an output event and a coaction as an input event. If a system can execute both an action and its coaction, it may execute them both simultaneously by taking an internal transition marked by the special action τ.

Synchronization

In CCS, every action a has an opposing *coaction* $\overline{a}$ (and $\overline{\overline{a}} = a$):

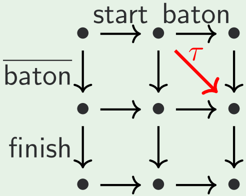
Actions: tick tock in50¢ outCoke . . .

Coactions: $\overline{\text{tick}}$ $\overline{\text{tock}}$ $\overline{\text{in50¢}}$ $\overline{\text{outCoke}}$ . . .

It is a convention to think of an action as an output event and a coaction as an input event. If a system can execute both an action and its coaction, it may execute them both simultaneously by taking an internal transition marked by the special action τ.

Example (Relay Race)

$$\begin{aligned} \mathbf{RACE} &= \mathbf{RUN}_1 \mid \mathbf{RUN}_2 \\ \mathbf{RUN}_1 &= \text{start.baton} \\ \mathbf{RUN}_2 &= \overline{\text{baton}}\text{.finish} \end{aligned}$$



Expansion Theorem

Let P and Q be processes. By expanding recursive definitions and using our existing equations for choice we can express P and Q as n-ary choices of action prefixes:

$$P = \sum_{i \in I} \alpha_i.\ P_i \text{ and } Q = \sum_{j \in J} \beta_j.\ Q_j.$$

Expansion Theorem

Let P and Q be processes. By expanding recursive definitions and using our existing equations for choice we can express P and Q as n-ary choices of action prefixes:

$$P = \sum_{i \in I} \alpha_i.\ P_i \text{ and } Q = \sum_{j \in J} \beta_j.\ Q_j.$$

Then, the parallel composition can be expressed as follows:

$$P \mid Q = \sum_{i \in I} \alpha_i.(P_i \mid Q) + \sum_{j \in J} \beta_j.(P \mid Q_j) + \sum_{i \in I,\ j \in J,\ \alpha_i = \overline{\beta}_j} \tau.(P_i \mid Q_j).$$

Expansion Theorem

Let P and Q be processes. By expanding recursive definitions and using our existing equations for choice we can express P and Q as n-ary choices of action prefixes:

$$P = \sum_{i\in I} \alpha_i.\ P_i \text{ and } Q = \sum_{j\in J} \beta_j.\ Q_j.$$

Then, the parallel composition can be expressed as follows:

$$P \mid Q = \sum_{i\in I} \alpha_i.(P_i \mid Q) + \sum_{j\in J} \beta_j.(P \mid Q_j) + \sum_{i\in I,\ j\in J,\ \alpha_i=\overline{\beta}_j} \tau.(P_i \mid Q_j).$$

From this, many useful equations are derivable:

$$\begin{array}{lcl} P \mid Q & = & Q \mid P \\ P \mid (Q \mid R) & = & (P \mid Q) \mid R \\ P \mid \textbf{STOP} & = & P \end{array}$$

Restriction

We wish a way to say “these are all the processes that there are”, in other words, to force synchronization to happen and not allow certain actions to be taken alone.

Restriction

We wish a way to say “these are all the processes that there are”, in other words, to force synchronization to happen and not allow certain actions to be taken alone.

Definition

If P is a process and a is an action (not τ), then $P \setminus a$ is the same as the process P except that the actions a and $\overline{a}$ may not be executed. We have

$$(a.P) \setminus b \quad = \quad a.(P \setminus b) \quad \text{if } a \notin \{b, \overline{b}\}$$

Restriction

We wish a way to say “these are all the processes that there are”, in other words, to force synchronization to happen and not allow certain actions to be taken alone.

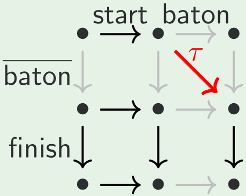
Definition

If P is a process and a is an action (not τ), then $P \setminus a$ is the same as the process P except that the actions a and $\overline{a}$ may not be executed. We have

$$(a.P) \setminus b \quad = \quad a.(P \setminus b) \quad \text{if } a \notin \{b, \overline{b}\}$$

Example (Relay Race)

$$\begin{array}{rcl}
\mathbf{RACE} & = & (\mathbf{RUN}_1 \mid \mathbf{RUN}_2) \setminus \text{baton} \\
\mathbf{RUN}_1 & = & \text{start}.\text{baton} \\
\mathbf{RUN}_2 & = & \overline{\text{baton}}.\text{finish}
\end{array}$$



Another Example

A man that eats every time a clock ticks:

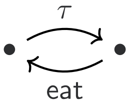
$$\begin{aligned}
\mathbf{CLOCK_4} &= \mathsf{tick}.\mathbf{CLOCK_4} \\
\mathbf{MAN} &= \overline{\mathsf{tick}}.\mathsf{eat}.\mathbf{MAN} \\
\mathbf{EXAMPLE} &= (\mathbf{MAN} \mid \mathbf{CLOCK_4}) \setminus \mathsf{tick}
\end{aligned}$$

Another Example

A man that eats every time a clock ticks:

$$\begin{aligned}
\mathbf{CLOCK}_4 &= \mathsf{tick}.\mathbf{CLOCK}_4 \\
\mathbf{MAN} &= \overline{\mathsf{tick}}.\mathsf{eat}.\mathbf{MAN} \\
\mathbf{EXAMPLE} &= (\mathbf{MAN} \mid \mathbf{CLOCK}_4) \setminus \mathsf{tick}
\end{aligned}$$

After deriving the picture, we get:



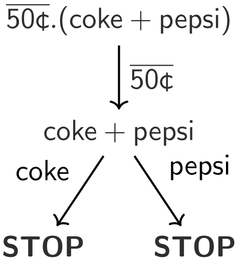
Semantics

Up until now, our semantics were given informally in terms of pictures. Now we will formalise our semantic intuitions.

Semantics

Up until now, our semantics were given informally in terms of pictures. Now we will formalise our semantic intuitions.

Our set of locations in our labelled transition system will be the set of all CCS processes. Locations can now be labelled with what process they are:



We will now define what transitions exist in our LTS by means of a set of *inference rules*. This technique is called *operational semantics*.

Inference Rules

In logic we often write:

$$\frac{A_1 \qquad A_2 \qquad \cdots \qquad A_n}{C}$$

To indicate that C can be proved by proving all assumptions A_1 through A_n. For example, the classical logical rule of modus ponens is written as follows:

$$\frac{A \Rightarrow B \qquad A}{B}\ \textsc{Modus Ponens}$$

Operational Semantics

Operational Semantics

$$\frac{}{a.P \xrightarrow{a} P}\textsc{Act} \qquad \frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'}\textsc{Choice}_1 \qquad \frac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'}\textsc{Choice}_2$$

$$\frac{P \xrightarrow{a} P'}{P \mid Q \xrightarrow{a} P' \mid Q}\textsc{Par}_1 \qquad \frac{Q \xrightarrow{a} Q'}{P \mid Q \xrightarrow{a} P \mid Q'}\textsc{Par}_2 \qquad \frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{\bar{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}\textsc{Sync}$$

$$\frac{P \xrightarrow{a} P' \qquad a \notin \{b, \bar{b}\}}{P \setminus b \xrightarrow{a} P' \setminus b}\textsc{Restrict}$$

Operational Semantics

$$\frac{}{a.P \xrightarrow{a} P}\textsc{Act} \qquad \frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'}\textsc{Choice}_1 \qquad \frac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'}\textsc{Choice}_2$$

$$\frac{P \xrightarrow{a} P'}{P \mid Q \xrightarrow{a} P' \mid Q}\textsc{Par}_1 \qquad \frac{Q \xrightarrow{a} Q'}{P \mid Q \xrightarrow{a} P \mid Q'}\textsc{Par}_2 \qquad \frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{\overline{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}\textsc{Sync}$$

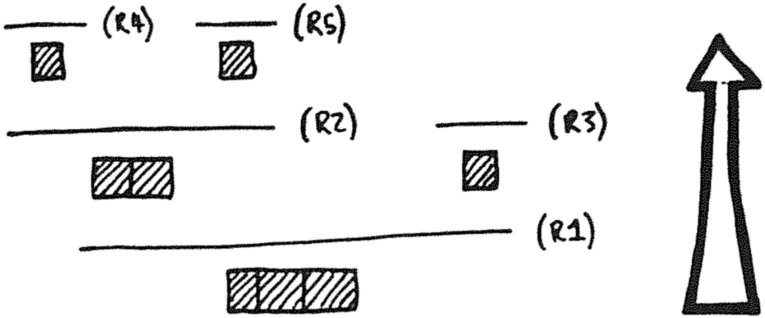
$$\frac{P \xrightarrow{a} P' \qquad a \notin \{b, \overline{b}\}}{P \setminus b \xrightarrow{a} P' \setminus b}\textsc{Restrict}$$

Bisimulation Equivalence

Two processes (or locations) P and Q are bisimilar iff they can do the same actions and those actions themselves lead to bisimilar processes. All of our previous equalities can be proven by induction on the semantics here.

Proof Trees

The advantages of this rule presentation is that they can be “stacked” to give a neat tree like derivation of proofs.



Exercise: Show $((a.P) + Q) \mid \overline{a}.R \xrightarrow{\tau} P \mid R$

Value Passing

We add synchronous channels into CCS by letting actions take parameters.

Actions:	$\mathsf{a}(3)$	$\mathsf{c}(15)$	$\mathsf{x}(True)$	$\ldots$
Coactions:	$\overline{\mathsf{a}}(x)$	$\overline{\mathsf{c}}(y)$	$\overline{\mathsf{c}}(z)$	$\ldots$

Value Passing

We add synchronous channels into CCS by letting actions take parameters.

Actions:	$\mathrm{a}(3)$	$\mathrm{c}(15)$	$\mathrm{x}(\mathit{True})$	$\ldots$
Coactions:	$\overline{\mathrm{a}}(x)$	$\overline{\mathrm{c}}(y)$	$\overline{\mathrm{c}}(z)$	$\ldots$

The parameter of an action is the value to be sent, and the parameter of a coaction is the variable in which the received value is stored.

Value Passing

We add synchronous channels into CCS by letting actions take parameters.

Actions:	$\mathsf{a}(3)$	$\mathsf{c}(15)$	$\mathsf{x}(\mathit{True})$	$\ldots$
Coactions:	$\overline{\mathsf{a}}(x)$	$\overline{\mathsf{c}}(y)$	$\overline{\mathsf{c}}(z)$	$\ldots$

The parameter of an action is the value to be sent, and the parameter of a coaction is the variable in which the received value is stored.

Example (Buffers)

A one-cell sized buffer is implemented as:

$$\mathbf{BUFF} = \overline{\mathsf{in}}(x).\mathsf{out}(x).\mathbf{BUFF}$$

Larger buffers can be made by stitching multiple **BUFF** processes together! This is one (overkill) way to model asynchronous communication in CCS.

Guards

Rather than add **if** statements, we add the notion of a *guard*:

Guards

Rather than add **if** statements, we add the notion of a *guard*:

Definition

If P is a value-passing CCS process and φ is a formula about the variables in scope, then $[\varphi]P$ is a process that executes just like P if φ holds for the current state, and like **STOP** otherwise.

Guards

Rather than add **if** statements, we add the notion of a *guard*:

Definition

If P is a value-passing CCS process and φ is a formula about the variables in scope, then $[\varphi]P$ is a process that executes just like P if φ holds for the current state, and like **STOP** otherwise.

We can define an **if** statement like so:

$$\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q \quad \equiv \quad ([\varphi].P) + ([\neg\varphi].Q)$$

Assignment

Most process algebras have no notion of state. Some presentations of value passing CCS also include assignment to update variables in the state:

Assignment

Most process algebras have no notion of state. Some presentations of value passing CCS also include assignment to update variables in the state:

Definition

If P is a process and x is a variable in the state, and e is an expression, then $[\![x := e]\!]P$ is the same as P except that it first updates the variable x to have the value e before making the transition.

Assignment

Most process algebras have no notion of state. Some presentations of value passing CCS also include assignment to update variables in the state:

Definition

If P is a process and x is a variable in the state, and e is an expression, then $[\![x := e]\!]P$ is the same as P except that it first updates the variable x to have the value e before making the transition.

With this, our value-passing CCS is now just as expressive as Ben-Ari's pseudocode. Moreover, the connection between CCS and transition diagrams is formalised, enabling us to reason symbolically about processes rather than semantically.

Process Algebra

This was an example of a *process algebra*. There are many such algebras and they have been very influential on the design of concurrent programming languages.

Process Algebra

This was an example of a *process algebra*. There are many such algebras and they have been very influential on the design of concurrent programming languages. Other process algebras include:

- The Algebra of Communicating Processes (Bergstra and Klop, 1982) which distinguishes between deadlock and termination.
- The Communicating Sequential Processes formalism (Hoare, 1978) with a more refined treatment of nondeterminism.
- The π-calculus (Milner et al. 1992), a derivative of CCS that allows for first class channels and processes.

Process Algebra

This was an example of a *process algebra*. There are many such algebras and they have been very influential on the design of concurrent programming languages. Other process algebras include:

- The Algebra of Communicating Processes (Bergstra and Klop, 1982) which distinguishes between deadlock and termination.
- The Communicating Sequential Processes formalism (Hoare, 1978) with a more refined treatment of nondeterminism.
- The π-calculus (Milner et al. 1992), a derivative of CCS that allows for first class channels and processes.

There are dozens of equivalences other than strong bisimulation that are useful for various scenarios.

What Now?

Next, we'll discuss distributed algorithms and commitment and consensus topics.