Chapter 2. A Calculus of Communicating Systems

Goals for Chapter 2:

- Syntax of CCS
  - Symbols for actions and for agents
  - Operators (combinators) of CCS
  - Rules for writing well-formed agent expressions.
- Semantics of CCS gives the meaning of agent expressions in terms of the meaning of operators used in the expressions.
  - Transition trees, transition graphs and their use in representing the behaviour of systems.
  - Inference trees as means to prove or disprove transitions of agent expressions.
- CCS as a formalism for the modelling of the behaviour of concurrent systems.

In this chapter, we firstly introduce a simple calculus based on synchronisation, where agent expressions have no value parameters. This calculus is called the basic calculus. Then, we extend the basic calculus to deal with value-passing, i.e. to deal with agent expressions with value parameters. The extended calculus is called the full calculus or the value-passing calculus.

1 Actions and Transitions

Actions

We assume the following infinite sets

- $\mathcal{A}$: the set of names, ranged over by a, b, c,..., e.g.

  $geth$, $puth$, $ackin$

- $\overline{\mathcal{A}}$: the set of co-names, ranged over by $\overline{a}$, $\overline{b}$, $\overline{c}$, ..., e.g.

  $\overline{geth}$, $\overline{puth}$, $\overline{ackin}$

- $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$: the set of labels, ranged over by l, l'. $\mathcal{L}$ comprises almost all, but not quite all, of the actions that agents can perform. We shall need one more special action, namely the internal, silent action τ, which is not a member of $\mathcal{L}$.
- Extend complementation to the whole of $\mathcal{L}$, so that $\overline{\overline{a}} = a$.
- In the basic calculus, labels have no value parameters.

Transitions

The behaviour of an agent will be defined in terms of its possible transitions from states to states (recall that we identify agents with states). A transition from one state to another state is accomplished by, or produces, an action. We write such transitions as

$$P \xrightarrow{l} Q.$$

This is a transition from state (agent) P to state (agent) Q by action l. This transition says that agent P performs action l and then behaves as agent Q does. For example, intuitively, the following should be valid transitions:

- $a.\mathbf{0} \xrightarrow{a} \mathbf{0}$
- $a.P + Q \xrightarrow{a} P$
- $a.P \mid Q \xrightarrow{a} P \mid Q$

Action Prefix

With the definition of $Hammer$ as in Chapter 1, we have

$$Hammer \xrightarrow{geth} Busyhammer \qquad (1)$$
$$Busyhammer \xrightarrow{puth} Hammer \qquad (2)$$

These transitions define the behaviour of $Hammer$ as precisely as its equational definitions: transition (1) defines the behaviour of

$$Hammer \stackrel{def}{=} geth.Busyhammer$$

and transition (2) defines the behaviour of

$$Busyhammer \stackrel{def}{=} puth.Hammer$$

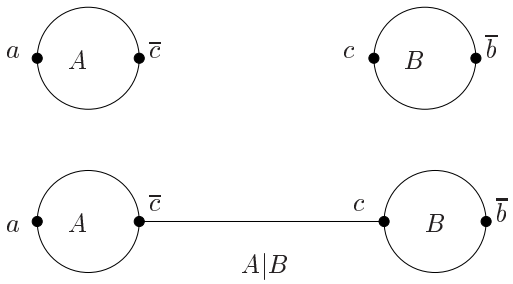
Thus, these transitions define the meaning of the Prefix combinators $geth.$ and $puth.$. Namely, transition (1) defines the meaning of $geth.Busyhammer$ and transition (2) defines the meaning of $puth.Hammer$. In general, agent $a.P$ has the transition $a.P \xrightarrow{a} P$ for every a and P.

Transitions of composite agents

At this point we shall only consider composite agents built with the parallel composition and restriction. Later in this chapter we will discuss other operators.

How to determine the transitions of $A|B$ from those of A and B? Consider the following A and B.

$$A \stackrel{def}{=} a.A' \qquad A' \stackrel{def}{=} \overline{c}.A$$
$$B \stackrel{def}{=} c.B' \qquad B' \stackrel{def}{=} \overline{b}.B$$



There are two kinds of transitions that can be performed by $A|B$:

1. A can do any of its actions in the context $A|B$, leaving B undisturbed. Similarly, B can do any of its actions, leaving A undisturbed. This can be written as follows, where α is any action:

   Since $A \xrightarrow{\alpha} A'$ we infer $A|B \xrightarrow{\alpha} A'|B$

   Since $B \xrightarrow{\alpha} B'$ we infer $A|B \xrightarrow{\alpha} A|B'$

   These are formally written as transition rules

   $$\frac{A \xrightarrow{\alpha} A'}{A|B \xrightarrow{\alpha} A'|B} \qquad \text{and} \qquad \frac{B \xrightarrow{\alpha} B'}{A|B \xrightarrow{\alpha} A|B'}$$

2. A communication between A and B takes the form of a handshake. It changes the states of the component agents simultaneously. Here, l is any visible action (cannot be τ):

   Since $A \xrightarrow{\overline{l}} A'$ and $B \xrightarrow{l} B'$ we infer $A|B \xrightarrow{\tau} A'|B'$

   We write this formally as

   $$\frac{A \xrightarrow{\overline{l}} A' \quad B \xrightarrow{l} B'}{A|B \xrightarrow{\tau} A'|B'}$$

Important: Features of silent actions τ:

- They are perfect (or completed) actions.
- They do not represent a potential for communication.
- They are not observable, i.e. cannot be communicated upon by agents.
- Only external actions of agents are observable, i.e. important as far as our understanding of systems' behaviour is concerned.

We use a single symbol τ to represent **all** such handshakes. Let the set of actions be $Act = \mathcal{L} \cup \{\tau\}$. Let α and β range over Act.

Now, we consider hiding ports by restricting actions. We define the Restriction operator (construct) $\backslash L$, where L is a set of names, as follows: if $P \xrightarrow{\alpha} P'$, then $P\backslash L \xrightarrow{\alpha} P'\backslash L$, provided that $\alpha, \overline{\alpha} \notin L$. **Important**: If L is a singleton set, i.e. it has only one element, say a, then we write $P\backslash a$ instead of the proper $P\backslash\{a\}$.

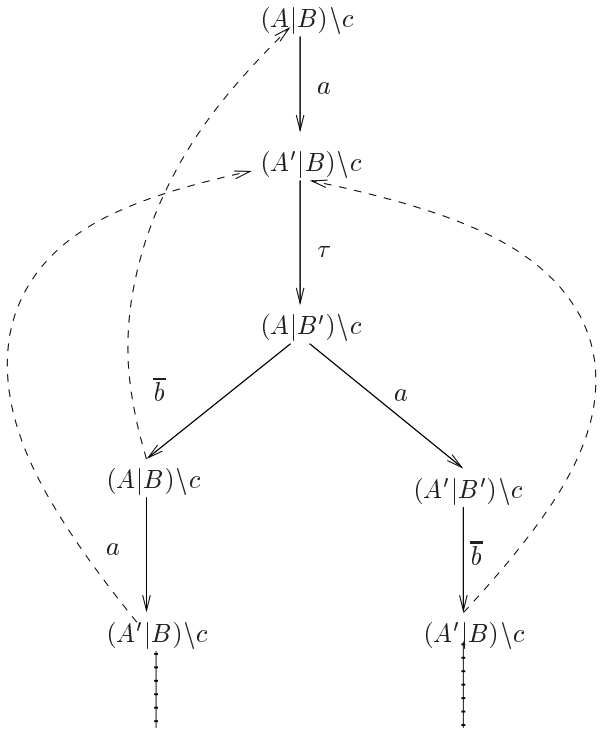
Consider the following agent, where A and B are as before.

$(A|B)\backslash c$

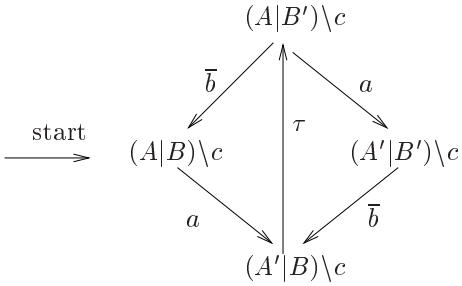
- $(A|B)\backslash c$ cannot perform c or $\overline{c}$ as they are forbidden by the Restriction operator.
- But, it can perform action τ which results from the communication on complementary ports c and $\overline{c}$.

Transition trees, graphs and behavioural equivalence

Now we can write down all the possible transitions of agent $(A|B)\backslash c$. We arrange these transitions into a tree, called the transition or derivation tree.

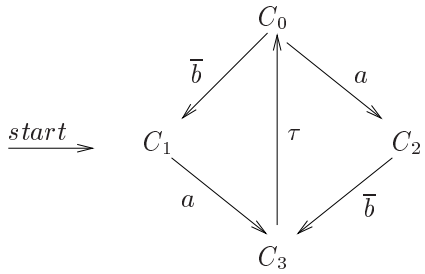


Notice that this tree is infinite, repeating the state $(A'|B)\backslash c$. Thus, if we fold the tree up along the dashed arrows in the tree, and draw the right $\overline{b}$ arc towards the first occurrence of $(A'|B)\backslash c$, and draw the left $\overline{b}$ arc towards the top $(A|B)\backslash c$, then we will get we have the following graph, called the transition graph or derivation graph, of agent $(A|B)\backslash c$.



The arrow labelled 'start' is not a part of the graph. It only indicates the starting state (node).

Imagine that you have another graph as follows.



The defining equations for C_0, C_1, C_2, and C_3 are as follows.

$$C_0 \stackrel{def}{=} \overline{b}.C_1 + a.C_2 \qquad C_1 \stackrel{def}{=} a.C_3$$
$$C_2 \stackrel{def}{=} \overline{b}.C_3 \qquad C_3 \stackrel{def}{=} \tau.C_0$$

We easily notice that the above two graphs are the same except for the names of the nodes. Namely, if we replace $(A|B')\backslash c$, $(A|B)\backslash c$, $(A'|B')\backslash c$, and $(A'|B)\backslash c$ in the first graph with C_0, C_1, C_2, and C_3 respectively, we obtain the second graph.

If we identify the behaviour of an agent with its transition graph we have $(A|B)\backslash c = C_1$ because the agents on the two sides of the equation have the same transition graph when we ignore the names given to the states (agents). In short, the two agents have the same transitions.

In the course of this module we will often perform a reasoning similar to the following. We rewrite the definition of C_0 by combining the four equations into a single defining equation:

$$C_0 \stackrel{def}{=} a.\overline{b}.\tau.C_0 + \overline{b}.a.\tau.C_0$$

Notice, that $C_1 = a.\tau.C_0$, so we have $(A|B)\backslash c = a.\tau.C_0$. Further, we should be able to ignore some of the τ action, since they are not observable, to have $(A|B)\backslash c = a.C$, where $C \stackrel{def}{=} a.\overline{b}.C + \overline{b}.a.C$.

If you consider $a.C$ as a specification of some system, and if $(A|B)\backslash c$ is a design or an implementation of that system, then the equation $(A|B)\backslash c = a.C$ is the statement of correctness of the design, implementation with respect to the specification. Checking whether or not $(A|B)\backslash c = a.C$ holds amounts to verifying the correctness. We will do quite a lot of such reasoning in the coming weeks.

Constructing transition graphs and derivation trees

One of the skills you will learn in this module is to construct transition graphs and derivation trees (see Section 5) for agents. For this you will use transition rules for agent's operators as descried in Section 4. You may find the following algorithm useful.

1. Choose an agent expression with which to start. If there is one which represents an initial state of the agent then choose that.

2. Find all the possible next actions for the agent you have chosen using the transition rules for the outermost operators of the agent. Draw an arc labelled with the action name for each of these actions. Remember, identical actions may generate different next states, so some arcs from a node may have the same action labelling them.

3. Work out the agent expressions which should appear as next states at the end of each of the arcs and add these to the graph if necessary. This is because it is quite possible that some of the states might already exist as a node in the graph, so you may not have to add new ones for every arc. When you construct the transition tree for the agent, you must list all the nodes separately (even if the repeat themselves) up-to a certain point.

4. Check to see if any node in the graph so far have any arcs missing. If there are such nodes, then go back to step 2 for each such node and repeat the process.

5. All nodes should now have arcs for all possible next actions. You may now find that you need to redraw the graph a number of times until you get the clearest and simplest version of it. It is a good idea to leave this stage until last, when you have worked out all the states and all labelled arcs.

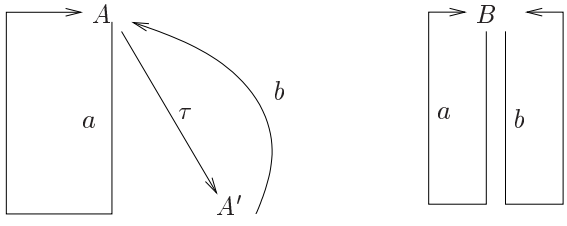
2 The Pre-emptive Power of τ

Previously we mentioned that we intend to drop as many internal actions as possible when dealing with equations between agents. So, how may of them can be dropped? Or should we drop all the internal actions? The question can be asked in a formal way: Which of the following two equations should be accepted as a general law?

$$(1) \quad P = \tau.P \qquad (2) \quad \alpha.P = \alpha.\tau.P$$

The first "law" allows us to drop any internal action of an agent while the second "law" allows to drop any but the first internal action of an agent. Which of them is valid? Consider the following example.

$$A \stackrel{def}{=} a.A + \tau.b.A \quad B \stackrel{def}{=} a.B + b.B$$



Recall that the τ action in agent A is an internal action of A, i.e. a completed communication between two components of A. Thus, the environment of A cannot interact with A through τ. In other words, the environment may only interact with A through the external action a by performing an $\overline{a}$, or through b by performing $\overline{b}$. However, action $\overline{a}$ of the environment and thus the action a of

A may be blocked (or preempted) by the internal τ. If the system A decides to perform τ instead of a forever, the communication between A and the environment through a may be blocked forever. We conclude that agent A is not totally controllable by the environment. Suppose the environment E may only communicate with A through a, e.g.

$$E \stackrel{def}{=} \overline{a}.E$$

Then, the system $(A|E)\setminus\{a\}$ may not be able to perform a communication on a. This happens when A performs an initial τ instead of a. Notice that E is always willing to interact with A, but A may refuse to react.

On the other hand, agent B is controllable by the environment. The environment can perform an $\overline{a}$ to force B to perform a and thus the alternative $b.B$ is discarded. Therefore, we should treat A and B differently. Formally $A = B$ should not hold. This implies that equation (1) should not be taken as a law.

3 The Syntax of CCS

We have

- $\mathcal{A}$ – set of names
- $\overline{\mathcal{A}}$ – set of co-names
- $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$ – set of labels
- $Act = \mathcal{L} \cup \{\tau\}$ – set of actions
- $\overline{\overline{a}} = a$, $\overline{\tau} = \tau$
- Relabelling function:

$$f(\overline{l}) = \overline{f(l)} \quad f(\tau) = \tau$$

We further assume

- $\mathcal{X}$ – set of agent variables, ranged over by $X, Y, \ldots$
- $\mathcal{K}$ – set of agent constants, ranged over by $A, B, \ldots,$
- We use I or J for indexing sets, for example $\{E_i : i \in I\}$ is a family of expressions indexed by I.

Agent expressions

The set $\mathcal{E}$ of agent expressions is the smallest set which includes $\mathcal{X}$ and $\mathcal{K}$ and contains the following expressions, where E, E_i are already in $\mathcal{E}$:

1. $\alpha.E$, a Prefix ($\alpha \in Act$)
2. $\sum_{i\in I} E_i$, a Summation (I an indexing set)

3. $E_1|E_2$, a Composition

4. $E\backslash L$, a Restriction ($L \subseteq \mathcal{L}$)

5. $E[f]$, a Relabelling (f is a relabelling function)

This is a very abstract mathematical definition. A definition of this style is often seen in Universal Algebra Theory. In algebraic terms, the above definition says that $< \mathcal{E}, ., +, |, \backslash, [f] >$ is the term algebra (or initial algebra) generated by the generators in $\mathcal{X}$ and $\mathcal{K}$ by using the five combinators. When defining a language like $\mathcal{E}$, computer scientists often use a structurally inductive definition which gives the (syntactic) rules for the language. The above algebraic definition of $\mathcal{E}$ is equivalent to the following inductive definition.
$\mathcal{E}$ is generated by the following rules:

1. If $X \in \mathcal{X}$, then $X \in \mathcal{E}$

2. If $A \in \mathcal{K}$, then $A \in \mathcal{E}$

3. If $E \in \mathcal{E}$ and $\alpha \in Act$, then $\alpha.E \in \mathcal{E}$

4. If $E_i \in \mathcal{E}$ for $i \in I$, the $\sum_{i \in I} E_i \in \mathcal{E}$

5. If $E_1, E_2 \in \mathcal{E}$, the $E_1|E_2 \in \mathcal{E}$

6. If $E \in \mathcal{E}$ and $L \subseteq \mathcal{L}$, then $E\backslash L \in \mathcal{E}$

7. If $E \in \mathcal{E}$ and f is a relabelling function, then $E[f] \in \mathcal{E}$

8. $\mathcal{E}$ only contains those expressions constructed by using the above rules

We use E, F, ... to range over $\mathcal{E}$. The first two rules are the base cases for the inductive definition, and they define the generators including $\mathbf{0}$. The next five rules are the induction steps that define how to generate composite elements (i.e. expressions) of $\mathcal{E}$ from given elements by using each of the five combinators. Rule 8 defines the closeness of $\mathcal{E}$ with respect to the generators and the combinators to include only the elements obtained by the rules 1-7.
Computer scientists also often use BNF to define a language. $\mathcal{E}$ can be defined by the following BNF where E stands for any expression in $\mathcal{E}$ to be defined:

$$
\begin{array}{rcl}
E & ::= & A \\
 & | & X \\
 & | & \alpha.E \\
 & | & \sum_{i \in I} E \\
 & | & E|E \\
 & | & E\backslash L \\
 & | & E[f]
\end{array}
$$

Here $A \in \mathcal{K}$, $X \in \mathcal{X}$, $\alpha \in \mathcal{A}$, $L \subseteq \mathcal{L}$ and f is a renaming function, and I is an indexing set.

About summation

- For any sets I and $\{E_i \ : \ i \in I\}$, $\sum_{i \in I} E_i$ is the summation of all the E_i's.
- When $I = \{1, 2\}$, then $\sum_{i \in I} E_i$ is written as $E_1 + E_2$
- When $I = \{1, 2, \ldots\}$, then $\sum_{i \in I} E_i \equiv E_1 + E_2 + \ldots$
- When $I = \emptyset$, then $\mathbf{0} \stackrel{def}{=} \sum_{i \in I} E_i$ is the inactive, deadlocked agent
- $\sum_{i \in I} E_i$ is also written as $\sum\{E_i \ : \ i \in I\}$
- When I is understood, $\tilde{E}$ for $\{E_i \ : \ i \in I\}$, and $\sum \tilde{E}$ or $\sum_i E_i$, for $\sum_{i \in I} E_i$.

The combinators have decreasing binding power in the following order: Restriction and Relabelling, Prefix, Composition, Summation (see pages 17 and 18). Recall that

$$R + a.P|b.Q\backslash L \qquad \text{means} \qquad R + ((a.P)|(b.(Q\backslash L)))$$

Agent variables and agent constants

From the definition of agent expressions, an agent expression may contain (free) agent variables, e.g. $a.X|b.Y$ contains agent variables X and Y. For an agent expression E, we use $Vars(E)$ to denote the set of (free) agent variables in E:

$$Vars(a.X|b.Y) = \{X, Y\}$$

- An agent expression E is called an agent if it does not contain (free) agent variables. We denote the set of agents by $\mathcal{P}$, and we let $P, Q, \ldots$ range over agents.
- Each Constant is an agent, and has a defining equation of the form

  $$A \stackrel{def}{=} P$$

  for example

  $$A \stackrel{def}{=} a.A' \text{ and } A' \stackrel{def}{=} \overline{c}.A$$

These two definitions guarantee that an agent indeed defines a system, whose behaviour is completely defined by the agent expression. We should also notice that agent constants can be defined in terms of each other, i.e. by mutual recursion. For example, the constants A and A' above. This definitional mechanism is the only way that agents with infinite behaviour can be defined in the calculus.

An agent expression E with agent variables represents a set of agents corresponding to different instantiations of these variables. Thus, an agent expression with variables does not define a system. Instead, it defines a class of systems. At this moment, we cannot see the significance of the use of agent variables since we are only dealing with agents. Agent variables will become important when we formally deal with equations of agent expressions.

4 Operational Semantics

In this section we formally define the meaning of agent expressions in terms of all their possible transitions. In other words, we represent the behaviour of an agent by a graph or tree of all its transitions. For this purpose, we use the general notion of a labelled transition system (LTS) defined as follows.
A LTS is a triple $(S, T, \{ \xrightarrow{t} : \ t \in T\})$ which consists of

- a set S of states (or nodes),
- a set T of (transition) labels, and
- a family of transition relations: $\xrightarrow{t} \subseteq S \times S$, for $t \in T$.

If $(s_1, s_2) \in \xrightarrow{t}$ for two states $s_1, s_2 \in S$ and a label $t \in T$, then

- we say that there is a transition from s_1 to s_2 by t, and
- denote this fact by $s_1 \xrightarrow{t} s_2$.

Thus, a LTS is a labelled directed graph

- the nodes of the graph are the states in S, and
- each edge is labelled with a label in T.

To define the semantics of $\mathcal{E}$ by a LTS, we

- take S to be $\mathcal{E}$, the agent expressions,
- take T to be Act, the actions,
- then define each transition relation $\xrightarrow{\alpha}$ over $\mathcal{E}$

After we define these transition relations, the LTS provides an operational interpretation for each agent expression $E \in \mathcal{E}$:

- starting from E as the initial state, any state that E can move to (reach) at any time during its execution is always one of the states (nodes) of the LTS,
- at any state of E we make progress by performing one of the possible actions (on outward-pointing arcs) from that state (node).

Thus, the LTS can be thought of as an abstract state machine which simulates step by step the behaviour of each agent expression.

The definition of transition relations $\xrightarrow{a}$ follows from the structure of expressions, i.e. the definition is given by induction on the structure of expressions on the left of $\xrightarrow{a}$. In other words, we define the transitions of a composite agent (that appears on the left of $\xrightarrow{a}$) in terms of transitions of its component agents. This is called Structural Operational Semantics, or SOS for short. For example, we have already seen that

From $A \xrightarrow{a} A'$ we infer $A|B \xrightarrow{a} A'|B$

In general, we will have a rule for Composition as follows.

From $E \stackrel{\alpha}{\rightarrow} E'$ infer $E|F \stackrel{\alpha}{\rightarrow} E'|F$

We write this as

$$\frac{E \stackrel{\alpha}{\rightarrow} E'}{E|F \stackrel{\alpha}{\rightarrow} E'|F}$$

and call it a transition rule. The rules for the CCS operators have the following general form.

$$\frac{\text{(0 or more) Hypotheses}}{\text{Conclusion}}$$

The Hypotheses contain several transitions of the components (sub-expressions) of the agent expression that appears on the left of the transition in the Conclusion. The Conclusion is a transition of that expression. We read the rules as follows: "from Hypotheses, we infer Conclusion".

4.1 Transition Rule for Prefix

Expression $\alpha.E$ can perform action α and then proceeds as agent expression E. This means that $\alpha.E$ has a transition from the initial state $\alpha.E$ to state E without any hypotheses:

$$\textbf{Act} \quad \frac{}{\alpha.E \stackrel{\alpha}{\rightarrow} E}$$

4.2 Transition Rules for Summation

Agent expression $\sum_{i \in I} E_i$ has an action α if any one summand E_j (where $j \in I$) has an action α:

$$\textbf{Sum}_j \quad \frac{E_j \stackrel{\alpha}{\rightarrow} E'_j}{\sum_{i \in I} E_i \stackrel{\alpha}{\rightarrow} E'_j} \quad (j \in I)$$

For $I = \{1, 2\}$

$$\textbf{Sum}_1 \quad \frac{E_1 \stackrel{\alpha}{\rightarrow} E'_1}{E_1 + E_2 \stackrel{\alpha}{\rightarrow} E'_1} \qquad \textbf{Sum}_2 \quad \frac{E_2 \stackrel{\alpha}{\rightarrow} E'_2}{E_1 + E_2 \stackrel{\alpha}{\rightarrow} E'_2}$$

For $I = \emptyset$, there is no rule for $\mathbf{0} \stackrel{def}{=} \sum_i E_i$. This means that $\mathbf{0}$ does not have any transitions.
Example. For agent expression $a.E + b.\mathbf{0} + c.F$, we have the following transitions:

$a.E + b.0 + c.F \stackrel{a}{\rightarrow} E$	by **Act** then by $\textbf{Sum}_1$
$a.E + b.0 + c.F \stackrel{b}{\rightarrow} \mathbf{0}$	by **Act** then by $\textbf{Sum}_2$
$a.E + b.\mathbf{0} + c.F \stackrel{c}{\rightarrow} F$	by **Act** then by $\textbf{Sum}_3$

This means the first step transition of $a.E + b.\mathbf{0} + c.F$ can be one of the above three (but not all the three) transitions.

4.3 Transition Rules for Composition

If E can perform an action α, E can perform this action in the context $E|F$ without disturbing F; similarly, if F can perform an action α, it can perform that action in context of $E|F$ without disturbing E. This captures the fact that the components E and F of $E|F$ can act concurrently with and independently of each other. The concurrency is formalised by the following two transition rules:

$$\textbf{Com}_1 \qquad \frac{E \xrightarrow{\alpha} E'}{E|F \xrightarrow{\alpha} E'|F}$$

$$\textbf{Com}_2 \qquad \frac{F \xrightarrow{\alpha} F'}{E|F \xrightarrow{\alpha} E|F'}$$

The components E and F of $E|F$ may communicate with each other through complementary actions (ports):

$$\textbf{Com}_3 \qquad \frac{E \xrightarrow{l} E' \quad F \xrightarrow{\overline{l}} F'}{E|F \xrightarrow{\tau} E'|F'}$$

Important: Communications between two components are synchronised. Communication is the result of two co-actions from the two participants happening synchronously. Thus, each act of communication is a handshake between precisely two agents.

Examples. For $(a.E + b.F)|(\overline{b}.E + c.F)$, the following are some of the first step transitions of the expression.

$$(a.E + b.F)|(\overline{b}.E + c.F) \xrightarrow{a} E|(\overline{b}.E + c.F)$$
$$(a.E + b.F)|(\overline{b}.E + c.F) \xrightarrow{b} F|(\overline{b}.E + c.F)$$
$$(a.E + b.F)|(\overline{b}.E + c.F) \xrightarrow{\overline{b}} (a.E + b.F)|E$$
$$(a.E + b.F)|(\overline{b}.E + c.F) \xrightarrow{c} (a.E + b.F)|F$$
$$(a.E + b.F)|(\overline{b}.E + c.F) \xrightarrow{\tau} F|E$$

For expression $a.b.E|c.d.F$, the following are some of the first two step transitions.

$$a.b.E|c.d.F \xrightarrow{a} b.E|c.d.F \xrightarrow{b} E|c.d.F$$
$$a.b.E|c.d.F \xrightarrow{a} b.E|c.d.F \xrightarrow{c} b.E|d.F$$
$$a.b.E|c.d.F \xrightarrow{c} a.b.E|d.F \xrightarrow{a} b.E|d.F$$
$$a.b.E|c.d.F \xrightarrow{c} a.b.E|d.F \xrightarrow{d} a.b.E|F$$

Important: We represent independent actions of an agent expression as interleaved although they may actually occur simultaneously in real systems.

4.4 Transition Rule for Restriction

When a port l ($\in L$) is restricted in an agent expression E, there will be no interaction between E through port l with any other agent outside E. Thus, E cannot perform a restricted action l. This is described by the following transition rule.

$$\textbf{Res} \qquad \frac{E \xrightarrow{\alpha} E'}{E\backslash L \xrightarrow{\alpha} E'\backslash L} \qquad (\alpha,\ \overline{\alpha} \notin L)$$

4.5 Transition Rule for Relabelling

$$\textbf{Rel} \quad \frac{E \stackrel{\alpha}{\rightarrow} E'}{E[f] \stackrel{f(\alpha)}{\rightarrow} E'[f]}$$

Example.

1. $(a.E)\backslash a$ and $(\overline{a}.F)\backslash a$ have no transitions. They both deadlock.
2. $(a.E|\overline{a}.F)\backslash a$ cannot have a a-transition or $\overline{a}$-transition. But it has the transition

   $$(a.E|\overline{a}.F)\backslash a \stackrel{\tau}{\rightarrow} (E|F)\backslash a$$

   because $(a.E|\overline{a}.F)$ has a τ-transition to $(E|F)$, and τ is different from a.
3. $((a.E)\backslash a)|((\overline{a}.F)\backslash a)$ deadlocks.
4. $(a.E)|((\overline{a}.F)\backslash a)$ only has an a-transition. It has no τ transition or $\overline{a}$-transition.

4.6 Transition Rule for Constants

$$\textbf{Con} \quad \frac{P \stackrel{\alpha}{\rightarrow} P'}{A \stackrel{\alpha}{\rightarrow} P'} \qquad (A \stackrel{def}{=} P)$$

For example, if $A \stackrel{def}{=} a.A'$, then, using the above rule, we can infer $A \stackrel{a}{\rightarrow} A'$ from $a.A' \stackrel{a}{\rightarrow} A'$.

4.7 Summary of Transition (SOS) Rules for CCS operators

Remember that labels α in the transition rules below stand for any visible or silent action. Note, that in $\textbf{Com}_3$ label l stands for an arbitrary visible action, similarly $\overline{l}$: i.e. l and $\overline{l}$ cannot be τ. Set L is a set of visible actions.

$$\textbf{Act} \quad \frac{}{\alpha.E \stackrel{\alpha}{\rightarrow} E}$$

$$\textbf{Sum}_j \quad \frac{E_j \stackrel{\alpha}{\rightarrow} E'_j}{\sum_{i\in I} E_i \stackrel{\alpha}{\rightarrow} E'_j} \quad (j \in I)$$

$$\textbf{Com}_1 \quad \frac{E \stackrel{\alpha}{\rightarrow} E'}{E|F \stackrel{\alpha}{\rightarrow} E'|F}$$

$$\textbf{Com}_2 \quad \frac{F \stackrel{\alpha}{\rightarrow} F'}{E|F \stackrel{\alpha}{\rightarrow} E|F'}$$

$$\textbf{Com}_3 \quad \frac{E \stackrel{l}{\rightarrow} E' \quad F \stackrel{\overline{l}}{\rightarrow} F'}{E|F \stackrel{\tau}{\rightarrow} E'|F'}$$

$$\textbf{Res} \quad \frac{E \stackrel{\alpha}{\rightarrow} E'}{E\backslash L \stackrel{\alpha}{\rightarrow} E'\backslash L} \qquad (\alpha,\ \overline{\alpha} \notin L)$$

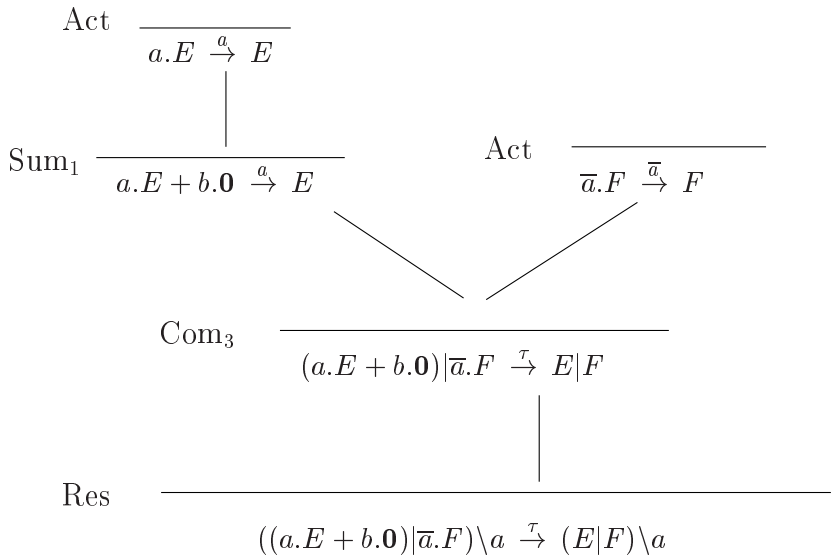
Rel $\dfrac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$

Con $\dfrac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} \quad (A \stackrel{def}{=} P)$

This set of rules is complete. This means that there are no transitions except those which can be inferred by the rules.

Inference trees

Now we can use the transition rules to justify whether or not a transition is possible for an agent expression. For example consider the transition $((a.E + b.\mathbf{0})|\overline{a}.F)\backslash a \xrightarrow{\tau} (E|F)\backslash a$. We infer it, or derive it, as follows:

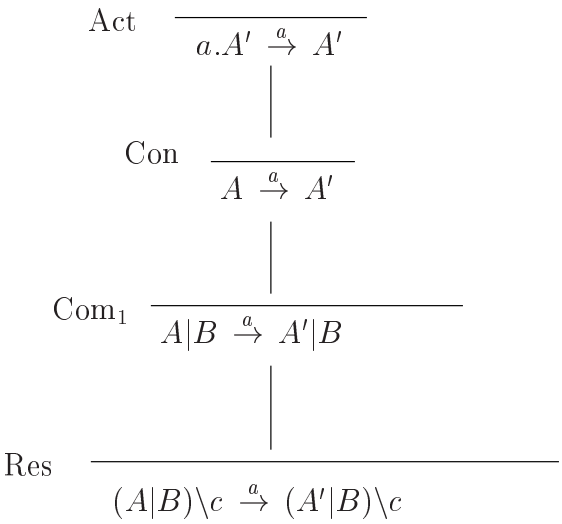


We infer $(A|B)\backslash c \xrightarrow{a} (A'|B)\backslash c$, where

$A \stackrel{def}{=} a.A' \qquad B \stackrel{def}{=} c.B'$

$A' \stackrel{def}{=} \overline{c}.A \qquad B' \stackrel{def}{=} \overline{b}.B$

as follows:



Inference trees demonstrate the relationship between transition graphs and the transitional semantics for operators. The semantic rules allow us to prove or disprove the correctness of the arcs connecting agent expressions in any transition graphs we construct. We do this by constructing inference diagram using just the rules we have stated in the transitional semantics. Each inference tree has the following characteristics.

- The root of each successful tree contains the transition we are trying to prove.
- Each node in the tree consists of a transition labelled by the rule which was used to derive it.
- A node may only refer to transitions already proved correct. For this reason all leaf nodes in an inference tree must be an instance of the use of the rule **Act**, since this is the only rule with the empty set of hypotheses.
- If we try to build a tree for an invalid transition, then we will be unable to complete the tree as just described.

To finish this section, we give inference trees for two somewhat complicated CCS expressions. Both were involved in the year 2000 examination.

Example

Fill in the gaps on the right-hand side of the following transition, and construct the inference tree to prove its validity.

$$((a.P + b.Q)[e/b] \mid (\overline{c}.R)[e/c]) \setminus e \xrightarrow{\tau} (???|???) \setminus e$$

You can guess the correct expressions that fill in the gaps, but you can also start building the inference tree and in the process of doing so you will work out the required expressions.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{b.Q \xrightarrow{b} Q}\,Act}{a.P + b.Q \xrightarrow{b} Q}\,Sum}{(a.P + b.Q)[e/b] \xrightarrow{e} Q[e/b]}\,Rel \qquad \dfrac{\dfrac{}{\overline{c}.R \xrightarrow{\overline{c}} R}\,Act}{\overline{c}.R[e/c] \xrightarrow{\overline{e}} R[e/c]}\,Rel}{((a.P + b.Q)[e/b]|\overline{c}.R)[e/c]) \xrightarrow{\tau} ((Q[e/b]|R[e/c])}\,Com_3}{((a.P + b.Q)[e/b]|\overline{c}.R)[e/c]) \setminus e \xrightarrow{\tau} ((Q[e/b]|R[e/c]) \setminus e}\,Res$$

Notice that we do not need to draw the vertical lines indicating the premises of transition rules.
The next example involved all CCS operators:
Fill in the gaps on the right-hand side of the following transition, and construct the inference tree to prove its validity, where $A \stackrel{def}{=} a.\mathbf{0} + \overline{b}.c.\mathbf{0}$.

$$(A \mid (a.b.\mathbf{0} + c.d.\mathbf{0})[d/a][b/d]) \setminus b \stackrel{\tau}{\to} (???|???[d/a][b/d]) \setminus b$$

Try to do this yourself before you study the answer.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\overline{b}.c.\mathbf{0} \stackrel{\overline{b}}{\to} c.\mathbf{0}}\,Act}{a.\mathbf{0} + \overline{b}.c.\mathbf{0} \stackrel{\overline{b}}{\to} c.\mathbf{0}}\,Sum}{A \stackrel{\overline{b}}{\to} c.\mathbf{0}}\,Con \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{a.b.\mathbf{0} \stackrel{a}{\to} b.\mathbf{0}}\,Act}{a.b.\mathbf{0} + c.d.\mathbf{0} \stackrel{a}{\to} b.\mathbf{0}}\,Sum}{(a.b.\mathbf{0} + c.d.\mathbf{0})[d/a] \stackrel{d}{\to} b.\mathbf{0}[d/a]}\,Rel}{(a.b.\mathbf{0} + c.d.\mathbf{0})[d/a][b/d] \stackrel{b}{\to} b.\mathbf{0}[d/a][b/d]}\,Rel}{(A|(a.b.\mathbf{0} + c.d.\mathbf{0})[d/a][b/d]) \stackrel{\tau}{\to} (c.\mathbf{0}|b.\mathbf{0}[d/a][b/d])}\,Com_3}{(A|(a.b.\mathbf{0} + c.d.\mathbf{0})[d/a][b/d]) \setminus b \stackrel{\tau}{\to} (c.\mathbf{0}|b.\mathbf{0}[d/a][b/d]) \setminus b}\,Res$$

5 Derivatives and Derivation Trees

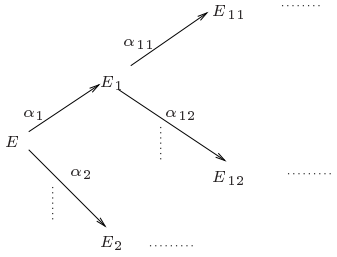
This section introduces the concepts of derivatives and derivation trees.

Derivatives

- Whenever $E \stackrel{\alpha}{\to} E'$, we call
  - the pair (α, E') an immediate derivative of E,
  - α an action of E,
  - E' an α-derivative of E.
- Whenever $E \stackrel{\alpha_1}{\to} \ldots \stackrel{\alpha_n}{\to} E'$, we call
  - $(\alpha_1 \ldots \alpha_n, E')$ a derivative of E,
  - $\alpha_1 \ldots \alpha_n$ an action-sequence of E,
  - E' an $\alpha_1 \ldots \alpha_n$-derivative of E (there may be several of them).
- The empty sequence ε is an action sequence of E, and E is itself the ε-derivative of E.

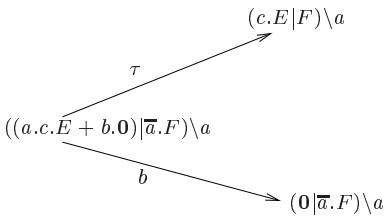
Derivation trees

Collect the derivatives of E into the derivation tree of E



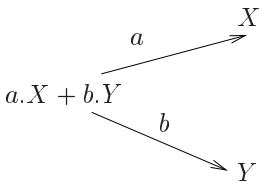
- For each expression at a non-terminal node, its immediate derivations are represented by outgoing arcs.
- A tree may be finite or infinite.
- A tree is called total if the expressions at terminal nodes have no immediate derivatives.
- Otherwise, it is called partial.

Example



- This tree is partial as the upper terminal has a c-derivative $(E|F)\backslash a$.
- The lower terminal node $(\mathbf{0}|\overline{a}.F)\backslash a$ has no derivatives, whatever F is.

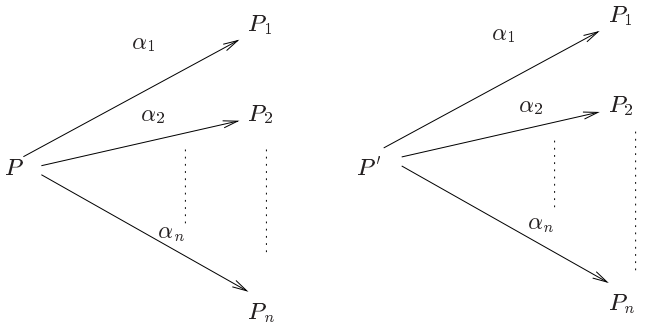
Example



- This tree is total but indeterminate, since the agent variables are uninstantiated. Any instantiation of these variables will allow us to develop the tree further.
- All immediate derivatives of an agent are agents.

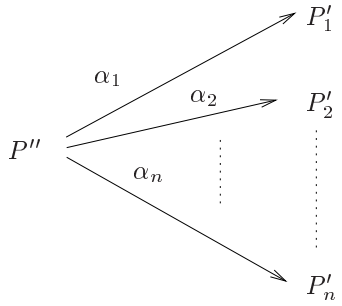
Behavioural equivalence

Let us use derivations trees to discuss a behavioural relation (equivalence) between agents. Consider agent expressions P and P' below.



P $\qquad$ $P' \stackrel{def}{=} \sum\{\alpha_i.P_i : 1 \leq i \leq n\}$

We conclude that P and P' should be related (equivalent), though P and P' may be very different expressions. More generally, if each P'_i is equivalent to P_i, and



we should have P, P' and P'' equivalent.

We conclude that one way to define a behavioural relation or equivalence between agents is to relate those agents which have the same derivation trees except for the names in their nodes. An alternative is to insist that the derivation trees are the same modulo (a) names in the nodes and (b) some silent actions τ in certain places in the derivation trees. This will be explained later.

6 Sorts Revisited

Recall our earlier definition.

Definition: For any $L \subseteq \mathcal{L}$, if the actions of P and all its derivatives are members of $L \cup \{\tau\}$, then we say P has sort L, and write $P : L$.

Proposition: For every E and L, L is a sort of E if and only if, whenever $E \xrightarrow{\alpha} E'$, then

1. $\alpha \in L \cup \{\tau\}$
2. L is a sort of E'

The syntactic sort of an agent expression

Assign a sort to an expression

- Assigned each X the sort $\mathcal{L}(X)$
- Assign each Constant A the sort $\mathcal{L}(A)$
- $\mathcal{L}(l.E) = \{l\} \cup \mathcal{L}(E)$
- $\mathcal{L}(\tau.E) = \mathcal{L}(E)$
- $\mathcal{L}(\sum_i E_i) = \bigcup_i \mathcal{L}(E_i)$
- $\mathcal{L}(E|F) = \mathcal{L}(E) \cup \mathcal{L}(F)$
- $\mathcal{L}(E\backslash L) = \mathcal{L}(E) - (L \cup \overline{L})$
- $\mathcal{L}(E[f]) = \{f(l) : l \in \mathcal{L}(E)\}$

- For $A \stackrel{def}{=} P$, then the following inclusion must hold

$$\mathcal{L}(P) \subseteq \mathcal{L}(A)$$

The last mean that we must assign a generous (big enough) sort to every Constant. For a given constant A defined by $a.b.c.A$ we are not allowed to assign the sort $\{a, b\}$ because $a.b.c.A$ can perform a, b and c. But, we are allowed to assign A the sort $\{a, b, c, d\}$ for example, and any S such that $S \supseteq \{a, b, c\}$.

Questions and answers

1. Is $\mathcal{L}(E)$ a sort of E? Yes!

2. May E perform all actions in $\mathcal{L}(E)$? Not necessarily! For example $\mathcal{L}((a.b.\mathbf{0})\backslash a) = \{b\}$, but $(a.b.\mathbf{0})\backslash a$ can never perform b. This means that $\mathcal{L}(E)$ may not be the smallest sort of E. In fact the problem to check whether a sort is the smallest sort of an expression is not decidable.

   $\mathcal{L}(E)$ is called the syntactic sort of E.

3. Why do we need such a sort?

   - To get an impression what actions E may perform.
     E.g. let P be $((a.\mathbf{0} + b.\mathbf{0})|(\overline{b}.\mathbf{0} + c.\mathbf{0}))\backslash b$,

     $$\mathcal{L}(P) = \{a, c\}$$

   - Some equational laws (to be discussed in the next chapter) depend upon sorts, e.g.

     $$(E|F)\backslash b = (E\backslash b)|F \qquad \text{provided } b, \overline{b} \notin \mathcal{L}(F)$$

7 The Value-Passing Calculus

We have established the language of the calculus for describing concurrent systems without value passing. Now we will see how we can use this simple calculus to deal with value passing. The main idea is to encode the expressions with value parameters as the expressions without value parameters. Consider the following examples.

Example. The buffer cell

$$C \stackrel{def}{=} in(x).C'(x) \qquad C'(x) \stackrel{def}{=} \overline{out}(x).C$$

Assume all values belong to a fixed set V

- The parameterized Constant $C'(x)$ becomes a family of Constants $\{C_v : v \in V\}$, meaning that for each v if the value of x is v then $C'(x)$ behaves like C'_v.

- The Prefix $\overline{out}(x).$ becomes a family of Prefixes $\{\overline{out_v}. : v \in V\}$, meaning that for each v if the value of x is v then $\overline{out}(x)$ is to output v and this can be recorded as $\overline{out}_v$.

- Thus the single defining equation for C' becomes a family of defining equations

$$\{C'_v \stackrel{def}{=} \overline{out_v}.C \quad : \quad v \in V\}$$

  This means that for each v, agent C'_v outputs v and then becomes agent C.

- Prefix '$in(x).$' means that it can accept any input value. Thus, it becomes '$\sum_{v \in V} in_v.$', because $in(x)$ means 'input a value' or 'may input any value', depending which one is provided by the environment.

- Since '$in(x).$' binds x in $in(x).C'(x)$, the defining equation for C becomes

$$C \stackrel{def}{=} \sum_{v \in V} in_v.C'_v$$

- The buffer cell can be defined as

$$C \stackrel{def}{=} \textstyle\sum_{v \in V} in_v.C'_v$$

$$C'_v \stackrel{def}{=} \overline{out_v}.C \qquad\qquad v \in V$$

Example. *Jobber*

$$Jobber \stackrel{def}{=} in(job).Start(job)$$

$$Start(job) \stackrel{def}{=} \begin{array}{l} \textbf{if } easy(job) \textbf{ then } Finish(job) \\ \textbf{else if } hard(job) \textbf{ then } UseH(job) \\ \textbf{else } Usetool(job) \end{array}$$

- The first defining equation becomes

$$Jobber \stackrel{def}{=} \textstyle\sum_{job \in V} in_{job}.Start_{job}$$

- *Start* takes the parameter *job*. The second equation becomes a family of equations, one for each $job \in V$.

- But for each equation, the right-hand side is determined by the predicates *easy* and *hard*. Thus the defining equation for *Start* becomes

$$Start_{job} \stackrel{def}{=} \begin{cases} Finish_{job} & (\text{if } easy(job)) \\ UseH_{job} & (\text{if } \neg easy(job) \wedge hard(job)) \\ Usetool_{job} & (\text{if } \neg easy(job) \wedge \neg hard(job)) \end{cases}$$

Remarks

- In theory, we do not need a larger calculus to deal with both value-passing and synchronisation.

- In practice, it is very tedious if systems with value-passing are always specified as families of equations.
- The intention is to enlarge the set $\mathcal{E}$ of agent expressions to a larger set $\mathcal{E}^+$ to express value-passing.
- We assign each agent Constant $A \in \mathcal{K}$ an arity, a non-negative integer representing the number of parameters it takes.
- $A(\in \mathcal{K})$ with arity 0 does not carry any parameter.
- We assume that value expressions e and boolean expressions b are built from value variables $x, y, \ldots$ together with value constants v by using value operators. For example, $x + y$, 5×2, $(2 > 3) \wedge \neg(x \leq 3)$.

7.1 Syntax

Agent expressions of the full calculus are defined as follows:

1. If $X \in \mathcal{X}$, then $X \in \mathcal{E}^+$
2. If $A \in \mathcal{K}$ with arity n, then $A(e_1, \ldots e_n) \in \mathcal{E}^+$
3. If $E \in \mathcal{E}^+$ and $a \in \mathcal{A}$, then $a(x).E, \overline{a}(e).E, \tau.E \in \mathcal{E}^+$
4. If $E_i \in \mathcal{E}^+$ for $i \in I$, then $\sum_{i \in I} E_i \in \mathcal{E}^+$
5. If $E_1, E_2 \in \mathcal{E}^+$, then $E_1 | E_2 \in \mathcal{E}^+$
6. If $E \in \mathcal{E}^+$ and $L \subseteq \mathcal{L}$, then $E \backslash L \in \mathcal{E}^+$
7. If $E \in \mathcal{E}^+$ and f is a relabelling function, then $E[f] \in \mathcal{E}^+$
8. If $E \in \mathcal{E}^+$, then **if** b **then** $E \in \mathcal{E}^+$
9. $\mathcal{E}^+$ only contains expressions constructed by using the above rules.

Also, each Constant A with arity n has a defining equation

$$A(x_1, \ldots, x_n) \stackrel{def}{=} E$$

where E contains no agent variables.

Remark

Agent **if** b **then** E **else** E' can be defined as **if** b **then** $E +$ **if** $\neg b$ **then** E'.

Syntactical Equality

For $E_1, E_2 \in \mathcal{E}^+$, we use $E_1 \equiv E_2$ to mean that E_1 and E_2 are syntactically identical. Note the difference between '$\equiv$' and '$=$'

$$P + Q = Q + P \quad P|Q = Q|P \quad P + P = P$$
$$Jobshop = Strongjobber|Strongjobber$$

But

$$P + Q \not\equiv Q + P \quad P|Q \not\equiv Q|P \quad P + P \not\equiv P$$

$$Jobshop \not\equiv Strongjobber|Strongjobber$$

7.2 Translation of $\mathcal{E}^+$ to $\mathcal{E}$

Surprisingly, the value-passing calculus can be faithfully (but not so neatly) represented within the basic calculus by translating the first calculus into the second calculus.

- If E has a free value variable x, then it can be treated as $\{E[v/x] : v \in V\}$.
- A value expression e without variables is identified with the value v to which it evaluates.

For $E \in \mathcal{E}^+$, let $\widehat{E}$ be its translated form in $\mathcal{E}$. We define the translation recursively on the structure of agent expressions.

The translation

1. If $F \equiv X$, $\hat{F} \equiv X$
2. If $F \equiv a(x).E$, $\widehat{F} \equiv \sum_{v \in V} a_v.E\widehat{\{v/x\}}$
3. If $F \equiv \overline{a}(e).E$, $\widehat{F} \equiv \overline{a_e}.\widehat{E}$.
4. If $F \equiv \tau.E$, $\widehat{F} \equiv \tau.\widehat{E}$
5. If $F \equiv \sum_{i \in I} E_i$, $\widehat{F} \equiv \sum_{i \in I} \widehat{E_i}$
6. If $F \equiv E_1|E_2$, $\widehat{F} \equiv \widehat{E_1}|\widehat{E_2}$
7. If $F \equiv E\backslash L$, $\widehat{F} \equiv \widehat{E}\backslash\{l_v : l \in L, v \in V\}$
8. If $F \equiv E[f]$, $\widehat{F} \equiv \widehat{E}[\hat{f}]$, where

   $$\hat{f}(l_v) = f(l)_v$$

9. If $F \equiv \textbf{if } b \textbf{ then } E$, then

   $$\widehat{F} \equiv \begin{cases} \widehat{E} & \text{if } b = true \\ \mathbf{0} & \text{Otherwise} \end{cases}$$

10. If $F \equiv A(e_1, \ldots, e_n)$, $\widehat{F} \equiv A_{e_1,\ldots,e_n}$

11. Furthermore, a single defining equation $A(\tilde{x}) \stackrel{def}{=} E$ is translated into the indexed set of defining equations:

$$\{A_{\tilde{v}} \stackrel{def}{=} E\{\widehat{\tilde{v}/\tilde{x}}\} : \tilde{v} \in V^n\}$$

where,

$\tilde{x} = x_1, \ldots, x_n$
$\tilde{v} = v_1, \ldots, v_n$
$V^n = \{v_1, \ldots, v_n \ : \ v_1, \ldots, v_n \in V\}$

Examples. What follows is a list of five examples of value-passing CCS expressions and their translations into the basic CCS.

1. $B \stackrel{def}{=} a(x).b(y).B(x,y)$

$B(x,y) \stackrel{def}{=} \overline{c}(x+1).\overline{d}(y+2).B$

$B_\varepsilon \stackrel{def}{=} \sum_{v_1 \in V} a_{v_1}. \sum_{v_2 \in V} b_{v_2}.B_{(v_1,v_2)}$

$B_{(v_1,v_2)} \stackrel{def}{=} \overline{c_{(v_1+1)}}.\overline{d_{(v_2+2)}}.B_\varepsilon \qquad$ for $v_1, v_2 \in V$

2. $F \equiv (F_1|F_2)\backslash b$

$F_1 \equiv a(x).\overline{b}(x).\mathbf{0}$

$F_2 \equiv b(y).\overline{c}(y+1).\mathbf{0}$

$F \equiv ((a(x).\overline{b}(x).\mathbf{0})|(b(y).\overline{c}(y+1).\mathbf{0}))\backslash b$

$\widehat{F} \equiv (\widehat{F_1}|\widehat{F_2})\backslash\{b_v : v \in V\}$

$\widehat{F_1} \equiv \sum_{v \in V} a_v.\widehat{F_{11}}$

$\widehat{F_{11}} \equiv \overline{b}_v.\mathbf{0}$

$\widehat{F_1} \equiv \sum_{v \in V} a_v.\overline{b}_v.\mathbf{0}$

$\widehat{F_2} \equiv \sum_{v \in V} b_v.\overline{c}_{v+1}.\mathbf{0}$

$\widehat{F} \equiv (\sum_{v \in V} a_v.\overline{b}_v.\mathbf{0})|(\sum_{v \in V} b_v.\overline{c}_{v+1}.\mathbf{0}))\backslash\{b_v : v \in V\}$

3. $F_1' \equiv F_1[d/b]$

$F' \equiv (F_1'|F_2)\backslash b$

$\widehat{F_1'} \equiv \widehat{F_1}[d_v/b_v : v \in V]$

$\equiv (\sum_{v\in V} a_v.\overline{b}_v.\mathbf{0})[d_v/b_v : v \in V]$

$\equiv \sum_{v\in V} a_v.\overline{d}_v.\mathbf{0}$

$\widehat{F'} \equiv ((\sum_{v\in V} a_v.\overline{d}_v.\mathbf{0})|(\sum_{v\in V} b_v.\overline{c}_{v+1}.\mathbf{0}))\backslash b$

F_2 and $\widehat{F_2}$ cannot do anything in F' and $\widehat{F'}$, respectively.

4. Suppose that V is the set of integers

$F \equiv in(x).(\textbf{if } x < 5 \textbf{ then } \overline{out}(x).\mathbf{0})$

$E \equiv \textbf{if } x < 5 \textbf{ then } \overline{out}(x).\mathbf{0}$

$F \equiv in(x).E$

$\widehat{F} \equiv \sum_{v\in V} in_v.E\{\widehat{v/x}\}$

$E\{v/x\} \equiv \textbf{if } v < 5 \textbf{ then } \overline{out}(v).\mathbf{0}$

$$E\{\widehat{v/x}\} \equiv \begin{cases} \overline{out_v}.\mathbf{0} & \text{if } v < 5 \\ \mathbf{0} & \text{otherwise} \end{cases}$$

$$\widehat{F} \equiv \sum_{v\in V} in_v.\begin{cases} \overline{out_v}.\mathbf{0} & \text{if } v < 5 \\ \mathbf{0} & \text{otherwise} \end{cases}$$

The semantics of F is defined as that of $\widehat{F}$

5. The buffer cell in value-passing calculus is defined by

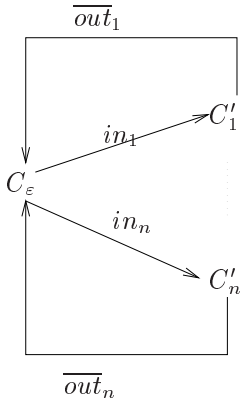
$C \stackrel{def}{=} in(x).C'(x) \qquad C'(x) \stackrel{def}{=} \overline{out}(x).C$

It can be translated into the basic CCS as follows:

$C_\varepsilon \stackrel{def}{=} \sum_{v\in V} in_v.C_v'$

$C_v' \stackrel{def}{=} \overline{out_v}.C_\varepsilon \qquad v \in V$

The derivation graph is for C_ε and all its resulting derivatives is as follows.



8 Summary

- Syntax of the basic calculus
  - Symbols for names—$\mathcal{A}$.
  - Symbols for labels—$\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$.
  - symbols for actions—$Act = \mathcal{L} \cup \{\tau\}$.
  - Action τ.
  - Rules for writing valid agent expressions.
  - How Composition and Restriction work together in composing agents.
- Semantics of the basic calculus
  - Transition (SOS) rules for CCS combinators, and a rule for constants.
  - Transition graphs.
  - Inference trees.
  - Derivation trees and graphs, and their use in showing behavioural equivalence.
  - Formal definition of a sort for an agent expression, and the syntactic sort of an agent expression.
- Value-Passing (full) calculus
  - The syntax of the full calculus.
  - Translation of the full calculus into the basic one.
  - Semantics of the full calculus is defined in terms of the basic. calculus.

9 Exercises

1. (a) Consider agent $Filter_K$, a one-place buffer-like agent that repeatedly accepts integers on channel in and outputs them on channel $\overline{out}$ provided that they belong to the set K. If input integers are not members of K, they are not output. In the value passing CCS, write a specification of $Filter_K$.

(b) A set of integers K of cardinality n is implemented by an agent $Tape(S)$, where S is any fixed sequence of all elements of K. Only one element of S can be accessed by the user at any time. The first element of S is accessible in the initial state of the tape agent. In order to access the next element of S the user needs to perform action $\overline{up}$. In order to access the subsequent elements of S the user needs to perform further actions $\overline{up}$. When the tape agent is not in the initial state, namely when the lth element of S is accessible for $1 < l \leq n$, the user can access the previous element of S, and similarly the preceding elements of S, by performing a suitable number of actions $\overline{down}$. When an element x of S is accessible, the user can read x by action $read(x)$. In the value passing CCS, write a specification of $Tape(S)$.

(c) Construct an agent $FILTER_K$ by combining $Tape(S)$, where S is any fixed sequence of all elements of K, and agents *Checker* and *ReceiveSend*, which you need to define, such that $FILTER_K$ implements $Filter_K$. Informally, *ReceiveSend* receives an integer x on *in*, send it to *Checker* to verify the membership of K, and outputs x on $\overline{out}$ if instructed by *Checker*. Agent *Checker* receives an integer x from *ReceiveSend*, uses $Tape(S)$ to check if x is an element of S (S is a fixed sequence of elements of K), moves $Tape(S)$ to its original state, and finally notifies *ReceiveSend* of the result of the check. In your construction, you will need to use relabelling, restriction and parallel composition combinators as well as other combinators.

(d) Draw a flow diagram for your construction in (c).

2. Consider CCS extended with two unary combinators g_1 and g_2 which are defined operationally as follows, where a and b are visible actions.

$$\frac{E \stackrel{b}{\to} E'}{g_1(E) \stackrel{\tau}{\to} g_2(E)} \qquad \frac{E \stackrel{\tau}{\to} E'}{g_1(E) \stackrel{\tau}{\to} g_1(E')} \qquad \frac{E \stackrel{a}{\to} E'}{g_2(E) \stackrel{ok}{\to} \mathbf{0}} \qquad \frac{E \stackrel{\tau}{\to} E'}{g_2(E) \stackrel{\tau}{\to} g_2(E')}$$

Draw transition graphs for the agents $g_1(\tau.(a.\mathbf{0}+b.\mathbf{0}))$, $g_1(\tau.(a.\mathbf{0}+b.\mathbf{0}+\tau.b.\mathbf{0}))$ and $g_1(\tau.(a.\mathbf{0}+\tau.b.\mathbf{0}))$.

3. Let ; be the sequential composition combinator defined by the following transition rules, where α is any visible or silent action.

$$\frac{E \stackrel{\alpha}{\to} E'}{E;F \stackrel{\alpha}{\to} E';F} \qquad \frac{F \stackrel{\alpha}{\to} F'}{E;F \stackrel{\alpha}{\to} F'} \; E = \mathbf{0}$$

Draw the transition graphs of the following agents:

(a) $(a.\mathbf{0}|b.\overline{a}.\mathbf{0});(a.\mathbf{0}|\overline{a}.b.\mathbf{0})$

(b) $(a.\mathbf{0}+b.\mathbf{0}+c.\mathbf{0});(d.\mathbf{0}|e.\mathbf{0})$

(c) By comparing the transition graphs of the agents $(P;Q)|R$ and $(P|R);(Q|R)$, or otherwise, argue that they have a different observable behaviour.

4. Consider CCS extended with a new binary combinator f which is defined by the following transition rules.

$$\frac{E \stackrel{\alpha}{\to} E'}{f(E,F) \stackrel{\alpha}{\to} f(E',F)} \qquad \frac{F \stackrel{\alpha}{\to} F'}{f(E,F) \stackrel{\alpha}{\to} F'}$$

(a) Explain the behaviour of $f(E, F)$ in terms of the behaviours of E and F. Draw the transition graph for $f(a.b.c.\mathbf{0} + d.e.\mathbf{0}, o.o.\mathbf{0})$.

(b) Give an example of a system whose behaviour can be described using combinator f as well as other CCS combinators, but cannot be easily described using only the standard CCS combinators.

5. Fill in the gaps on the right-hand sides of the following transitions, and construct inference trees for them, where $A \stackrel{def}{=} a.\mathbf{0} + \overline{b}.c.\mathbf{0}$.

$$((a.P + b.Q)[e/b] \mid (\overline{c}.R)[e/c]) \setminus e \stackrel{\tau}{\rightarrow} (???|???) \setminus e$$

$$(A \mid (a.b.\mathbf{0} + c.d.\mathbf{0})[d/a][b/d]) \setminus b \stackrel{\tau}{\rightarrow} (???|???[d/a][b/d]) \setminus b$$

6. Fill in the gaps (denoted by ???) on the right-hand side of the following transition, and construct the inference tree for the transition, assuming $P \stackrel{def}{=} e.b.\mathbf{0} + \overline{c}.\mathbf{0}$.

$$(P \setminus \{c\} \mid (a.\mathbf{0} + d.\mathbf{0})[\overline{e}/a,\ b/d]) \setminus \{e, b\} \stackrel{???}{\rightarrow} (???) \setminus \{e, b\}$$

7. Draw the transition graph for $(P|Q|R) \setminus \{d, e, f, g\}$, where

$$\begin{array}{lcl} P & \stackrel{def}{=} & a.(\overline{d}.f.P + \overline{e}.g.P) \\ Q & \stackrel{def}{=} & d.b.\overline{e}.Q + f.b.\overline{g}.Q \\ R & \stackrel{def}{=} & e.c.\overline{f}.R \end{array}$$

8. Draw the transition graphs for A and $(A|B)\setminus b$, where

$$\begin{array}{l} A \stackrel{def}{=} a.A_1, \quad A_1 \stackrel{def}{=} a.\overline{b}.A_1 + \overline{b}.a.A_1 \\ B \stackrel{def}{=} b.B + a.B \end{array}$$

9. Assuming that $a \neq b$, derive the following transitions by the means of inference trees using the transition rules for the CCS combinators.

$$\begin{array}{l} (a.E + b.G)|\overline{a}.F \stackrel{a}{\rightarrow} E|\overline{a}.F \\ (a.E + b.G)|\overline{a}.F \stackrel{\overline{a}}{\rightarrow} (a.E + b.G)|F \\ (a.E + b.G)|\overline{a}.F \stackrel{b}{\rightarrow} G|\overline{a}.F \\ ((a.E + b.G)|\overline{a}.F)\setminus a \stackrel{b}{\rightarrow} (G|\overline{a}.F)\setminus a \\ ((a.E + b.G)|\overline{a}.F)\setminus a \stackrel{\tau}{\rightarrow} (E|F)\setminus a \end{array}$$

10. **Mutual Exclusion**. Imagine a system which consists of printer P,

$$P \stackrel{def}{=} trans(x).\overline{print}(x).P$$

and two users $U1$ and $U2$ who send jobs to printer:

$$U1 \stackrel{def}{=} \overline{trans}(Happy).\overline{trans}(Birthday).\mathbf{0}$$

and

$$U2 \stackrel{def}{=} \overline{trans}(Merry).\overline{trans}(Christmas).\mathbf{0}$$

Draw the transition graph for $(P|U1|U2)\backslash trans$. How many different printouts can we get? Show how a semaphore can be used with the above system to ensure that jobs of each user are printed in full, and are not interrupted by jobs of other users. Hint: keep the printer as it is but change the expressions for the users so they can interact with the semaphore.

11. **Expressiveness of CCS**. Imagine a simple system which consists of a television set, TV, and a remote control unit, RC. TV can be switched on and switched off by RC at any time. Channel *on* is private to TV and RC. When on and only when on, TV repeatedly plays frames and sound, which are simply denoted by the action *play*. Define the system in CCS. Draw the transition graphs for the system's components and for the system itself. Do you see any problems with your solution? Explain your answer.

12. **A railway level crossing controller**. Consider a railway level crossing between a single track railway and a road. A gate at each side of the railway controls road traffic. Assume that there are only two states 'open' and 'closed' for each gate. When both gates are open road traffic can pass the crossing, and when both are closed no road traffic can pass the crossing. Train traffic on the railway is controlled by a signal on the side of the tracks along which the trains approach. The train signal shows: 'stop' or 'go'. The behaviour of the controller is informally described as follows.

    - It starts from a state in which no trains are approaching or passing, and the gates are open.
    - It repeatedly receives information about the train traffic on the railway.
    - When a train approaches, the gates should close.
    - When the gates have closed, the train signal sets 'go' to allow the train to pass through the crossing.
    - When no trains are approaching or passing, the train signal sets 'stop', and the gates open.

    (a) Using agent Constants, Prefix and Summation only in the full calculus, define an agent RCC, with

    $$\mathcal{L}(RCC) = \{in, open_1, close_1, open_2, close_2, go, stop\}$$

    which specifies the behaviour of the railway level crossing controlling system.

    (b) The railway level crossing controlling system is usually implemented by using a Sensor, a Controller (which is often a computer), two electronic Gates, and a traffic Light. The sensor keeps checking through a port the information about the train traffic on the railway and sending the signals through a port to the controller, the controller then sends acting information to the gates and the light through different ports. The gates and the light will then act according to the signals they receive – for example a gate will close if it receives a signal 'closegate' from the controller; the light will show 'go' if it receives the signal 'setgo', etc.

    Define the component agents *Sensor*, *Controller*, *Gate* and *Light* and write the Restricted Composition of the Relabelled form of these components that satisfies the requirement of the railway level crossing controller.

(c) Draw a flow graph for the composite agent in your solution to part (b). Explain why you think that the system in part (b) satisfied the specification as expressed by your answer to part (a).

13. **Temperature controller.** The temperature of a plant is controlled through a thermostat, which keeps measuring the temperature and turns a heater on and off.

(a) Using the value-passing CCS, write a specification of the temperature controller as an agent *Contspec* which satisfies the following requirements:

i. It has the sort $\{in, \overline{heateron}, \overline{heateroff}\}$.

ii. It starts from a state in which the heater is off.

iii. It repeatedly receives information about the temperature of the plant at port *in*, and then behaves, according to the information received, as follows:

- if the temperature is too low, say lower than or equal to *cold*, when the heater is off, then the heater should go on by performing an action $\overline{heateron}$; if the heater is already on, it should be left on.
- if the temperature is high enough, say higher than or equal to *warm*, when the heater is on, then the heater should go off by performing an action $\overline{haeteroff}$; if the heater is already off, it should be left off.

(b) The controller is usually implemented by several components including a sensor, a computer, and an electronic heater. The sensor keeps sensing the temperature of the plant and keeps sending the sensed information to the computer. According to the information received from the sensor, the computer sends instruction signals, such as *on* or *off*, to the heater. The heater then performs the functions as instructed by the computer, for example, the heater must be on (or off) after it receives a message *on* (or *off* respectively).

Define the component agents, *Sensor*, *Computer* and $Heater$ such that when composed appropriately they implement *Contspec*. Write out the definition of the implementation. Argue informally, or otherwise, that your implementation is correct.