Introduction to CCS and π-calculus

Pierre-Louis Curien

May 20, 2003

Abstract

These notes in evolving state started as a complement to [7, 8], on which my graduate course "Calculs de Processus" (DEA Programmation, Paris) builds. I hope to bring them to a reasonably self-contained status.

1 CCS

Concurrency comes in when several machines are connected in some network, performing computations that may be independent, but may also require some communication (for example through access to a shared memory). Process calculi borrow concepts both from automata theory and from λ-calculus.

1.1 From automata to CCS

We remove final states because we are not interested in termination, and initial states since we assimilate processes with states, and hence any state is "initial" relative to the process it is iddentified with. Such an automaton deprived from initial and final states is called a *labelled transition system*, or LTS for short. Hence an automaton is just given by a finite set of states, or *processes*, call them $P, Q, \ldots$, a finite alphabet whose members are called *actions*, and transitions between them, written $P \stackrel{a}{\rightarrow} Q$. A finite automaton together with one of its states, that is, a process, can be described by the following syntax:

$$P ::= \Sigma_{i \in I} a_i \cdot P_i \mid let\ \vec{K} = \vec{P}\ in\ K_j \mid K$$

where I is a finite set, $\vec{K} = K_1, \ldots, K_n$, $\vec{P} = P_1, \ldots, P_n$, and $j \in \{1, \ldots, n\}$. We call $\Sigma_{i \in I} a_i \cdot P_i$ a *sum*, $let \vec{K} = \vec{P} in K_j$ a (mutually) *recursive* definition,$\vec{K} = \vec{P}$ a *system of equations*, and K a *process variable*. We in fact call process a term of the syntax where each process variable is in the scope of a recursive definition. The empty sum ($I = \emptyset$) is denoted by 0.

Without the *let in* construct, we have automata without cycles, that is, we have trees whose nodes are labelled by processes and whose edges are labelled by actions: we call them (finite branching) *synchronization trees*. If we allow infinite sums (i.e., I may be infinite), we get synchronization trees for short. CCS is obtained by allowing infinite sums and by adding two constructs: *parallel composition*, and *restriction*:

$$P ::= \Sigma_{i \in I} \mu_i \cdot P_i \mid let\ \vec{K} = \vec{P}\ in\ K_j \mid K \mid (P \mid Q) \mid (\nu a)P$$

The set *Act* of actions is now composed of *labels* a, their co-labels $\overline{a}$, and a silent action τ that records a synchronisation. Thus, $Act = L \cup \overline{L} \cup \{\tau\}$ (disjoint union), where L is the set of labels, also called *names*, or *channels*. We let μ range over Act and α range over $L \cup \overline{L}$, with the convention that $\overline{\overline{\alpha}} = \alpha$. If we remove the *let in* construct from CCS and restore the finiteness of sums, we get *finitary CCS*. If we just restore the finiteness of sums, we get CCS with finite sums. In summary, we have:

CCS
$P ::= \Sigma_{i \in I} \mu_i \cdot P_i \mid let\ \vec{K} = \vec{P}\ in\ K_j \mid K \mid (P \mid Q) \mid (\nu a)P$

Synchronization Trees
$P ::= \Sigma_{i \in I} \mu_i \cdot P_i$

CCS with finite sums
$P ::= \Sigma_{i \in I} \mu_i \cdot P_i \mid let\ \vec{K} = \vec{P}\ in\ K_j \mid K \mid (P \mid Q) \mid (\nu a)P \quad (I \text{ finite})$

Finitary CCS
$P ::= \Sigma_{i \in I} \mu_i \cdot P_i \mid (P \mid Q) \mid (\nu a)P \quad (I \text{ finite})$

1.2 Labelled operational semantics

We define a labelled transition system on processes by the following rules:

$$\frac{}{\Sigma_{i\in I}\mu_i \cdot P_i \stackrel{\mu_i}{\to} P_i} \qquad \frac{P \stackrel{\mu}{\to} P' \quad (\mu \neq a, \overline{a})}{(\nu a)P \stackrel{\mu}{\to} (\nu a)P'}$$

$$\frac{P \stackrel{\mu}{\to} P'}{P \mid Q \stackrel{\mu}{\to} P' \mid Q} \qquad \frac{Q \stackrel{\mu}{\to} Q'}{P \mid Q \stackrel{\mu}{\to} P \mid Q'} \qquad \text{COMM} \ \frac{P \stackrel{\alpha}{\to} P' \quad Q \stackrel{\overline{\alpha}}{\to} Q'}{P \mid Q \stackrel{\tau}{\to} P' \mid Q'}$$

$$\frac{P_j[\vec{K} \leftarrow (\mathit{let}\ \vec{K} = \vec{P}\ \mathit{in}\ \vec{K})] \stackrel{\mu}{\to} P'}{\mathit{let}\ \vec{K} = \vec{P}\ \mathit{in}\ K_j \stackrel{\mu}{\to} P'}$$

The most interesting rule is the one named COMM (for "communication").

1.3 Reduction operational semantics

In the labelled semantics, τ-transitions, also called *silent transitions*, correspond to internal evolutions, while the α-transitions describe interactions with the environment. In λ-calculus, one considers only a notion of internal reduction: the β-reduction. It is also possible to formulate internal reduction in CCS without reference to the environment, which amounts to specifying silent transitions without appeal to non-silent ones as in the rule COMM. The price to pay is that we need to introduce a notion of equivalence, called structural equivalence (which can be easily shown to be included in strong bisimilarity, to be defined shortly), and denoted by $\equiv$. There are variations on what people put in the structural equivalence. Here, we put the minimum needed to make the definition work (i.e., to prove Proposition 1.1).

$$\begin{array}{l}
\Sigma_{i\in I}\mu_i \cdot P_i \equiv \Sigma_{i\in I}\mu_{f(i)} \cdot P_{f(i)} \quad (f \text{ permutation}) \\
P \mid Q \equiv Q \mid P \\
P \mid (Q \mid R) \equiv (P \mid Q) \mid R \\
((\nu a)P) \mid Q \equiv (\nu a)\ (P \mid Q) \quad (a \text{ not free in } Q) \\
\mathit{let}\ \vec{K} = \vec{P}\ \mathit{in}\ K_j \equiv Pj[\vec{K} \leftarrow (\mathit{let}\ \vec{K} = \vec{P}\ \mathit{in}\ \vec{K})]
\end{array}$$

The last equation, from left to right, is called scope extension. We can now specify the reduction relation $\to$ between processes.

$$\text{REACTION}\quad \frac{}{P_1 + a\cdot P \mid \overline{a}\cdot Q + Q_1 \to P \mid Q} \qquad \frac{}{P_1 + \tau\cdot P \to P}$$

$$\frac{P_1 \to P_1'}{P_1 \mid P_2 \to P_1' \mid P_2} \qquad \frac{P \to P'}{(\nu a)P \to (\nu a)P'}$$

$$\frac{P_1 \equiv P_2 \to P_2' \equiv P_1'}{P_1 \to P_1'}$$

Proposition 1.1 *The relations $\to$ and $\xrightarrow{\tau}\equiv$ coincide.*

PROOF (INDICATION). One establishes the following claims:

1. If $P \xrightarrow{\mu} P'$ and $P \equiv Q$, then there exists Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \equiv Q'$.
2. If $P \xrightarrow{\alpha} P'$, then $P \equiv (\nu\vec{a})\,(\alpha\cdot Q + P_1 \mid P_2)$ and $P' \equiv (\nu\vec{a})\,(P_1 \mid P_2)$, for some $\vec{a}, P_1, P_2, Q$.

The inclusion $\to \subseteq \xrightarrow{\tau}\equiv$ can be proved with the help of Claim (1). The converse inclusion is proved with the help of Claim (2). □

There is a way to escape structural congruence, without going into labelled operational semantics. An elegant such formulation, called *chemical abstract machine*, was proposed in [1]. The key idea is to "dissolve" processes as multisets. The rewriting relation is thus now between multisets M. The machine has reversible rules $\leftrightharpoons$, called *heating rules*, which can be applied both ways, and a single reaction rule $\to$. The key heating rule is $(M, P \mid Q) \leftrightharpoons (M, P, Q)$. We fully describe a particularly simple chemical abstract machine in section 4.8.

1.4 Bisimulation

This is the key notion of equivalence in concurrency theory. It makes sense on any LTS. If $\mathcal{R}$ is a binary relation, then we set $\mathcal{R}^{-1} = \{(Q,P) \mid P\,\mathcal{R}\,Q\}$.

Definition 1.2 *A* simulation *is a relation* $\mathcal{R}$ *such that for all* P, Q*, if* $P \mathcal{R} Q$ *then for all* μ, P' *such that* $P \xrightarrow{\mu} P'$ *there exists* Q' *such that* $Q \xrightarrow{\mu} Q'$ *and* $P' \mathcal{R} Q'$*. A* bisimulation *is a relation* $\mathcal{R}$ *such that* $\mathcal{R}$ *and* $\mathcal{R}^{-1}$ *are simulations. We say that* P, Q *are* bisimilar *if there exists a bisimulation* $\mathcal{R}$ *such that* $P \mathcal{R} Q$*.*

We take *strong bisimulation* to be the instance of this definition on the LTS of section 1.2. There are variations, for example, one may speak of an LTS simulating another LTS: but by considering the two LTS's as a single LTS (the juxtaposition of the two LTS's), the above definition applies. If $\mathcal{R}, \mathcal{S}$ are bisimulations, then so is their composition $RS = \{(P, R) \mid \exists\, Q \;\; P \mathcal{R} Q \text{ and } Q \mathcal{S} R\}$. In particular, $\sim\sim \subseteq \sim$, i.e., bisimilarity is transitive.

Clearly, if P and Q are bisimilar, then one simulates the other and vice-versa. The converse is not clear, since one has to use the same relation for the two sides of the bisimulation. And indeed it does not hold. Consider

$$\begin{array}{ll} P_1 = a \cdot P_2 + a \cdot P_4 & Q_1 = a \cdot Q_2 \\ P_2 = b \cdot P_3 & Q_2 = b \cdot Q_3 \end{array}$$

Suppose that $\mathcal{T}$ is a simulation containing (P_1, Q_1). We have:

$$P_1 \, T \, Q_1 \text{ et } P_1 \xrightarrow{a} P_4 \quad \Rightarrow \quad P_4 \, T \, Q_2$$

This implies that $\mathcal{T}^{-1}$ cannot be a simulation, since action b cannot be pas performed from P_4. Hence P_1 et Q_1 are not bisimilar. However we have the following simulations in each direction:

$$\begin{array}{llll} P_1 \, \mathcal{T} \, Q_1 & P_4 \, \mathcal{T} \, Q_2 & P_2 \, \mathcal{T} \, Q_2 & P_3 \, \mathcal{T} \, Q_3 \\ Q_1 \, \mathcal{S} \, P_1 & Q_2 \, \mathcal{S} \, P_2 & Q_3 \, \mathcal{S} \, P_3 \, . & \end{array}$$

Intuitively, considering $a = 1$ euro and b as a drink, we do not want to consider the two processes as equivalent: P_1 takes your coin and decides on its own whether it will grant you a drink, while Q_1 guarantees the drink.

2 Induction and coinduction

We consider monotone operators from $\mathcal{P}(X)$ to $\mathcal{P}(X)$ (the set of subsets of X), ordered by inclusion (more generally, one can take a complete lattice instead of $\mathcal{P}(X)$). Recall that a partial order $(D, \leq)$ is a set equipped with a reflexive, transitive and anti-symmetric relation $\leq$.

2.1 Least and greatest fixpoints

Definition 2.1 *A function $f : D \to E$, where D, E are partial orders, is called monotone if $x \leq y$ implies $f(x) \leq f(y)$ for all x, y.*

Definition 2.2 *Given a (monotone) function $f : D \to D$ where D is a partial order, a* prefixpoint *(resp. a* postfixpoint*, a* fixpoint*) of f is a point x such that $f(x) \leq x$ (resp. $x \leq f(x)$, $x = f(x)$).*

Theorem 2.3 (Tarski) *Any monotone function $G : \mathcal{P}(X) \to \mathcal{P}(X)$ has a least prefixpoint, which is moreover a fixpoint, and a greatest postfixpoint, which is moreover a fixpoint. They are respectively:*

$$
\begin{aligned}
\mathit{lfp}(G) &= \bigcap\{X \mid G(X) \subseteq X\} \\
\mathit{gfp}(G) &= \bigcup\{X \mid X \subseteq G(X)\}
\end{aligned}
$$

Proof. It is enough to prove one half of the theorem, since a monotone function from $(\mathcal{P}(X), \subseteq)$ to $(\mathcal{P}(X), \subseteq)$ is also monotone from $(\mathcal{P}(X), \supseteq)$ to $(\mathcal{P}(X), \supseteq)$. The claimed definition of $\mathit{lfp}(G)$ says exactly that is is included in any prefixpoint. We are done if we show that it is actually a fixpoint. So let $Y = \bigcap\{X \mid G(X) \subseteq X\}$, and let Z be an arbitrary prefixpoint of G. We have $Y \subseteq Z$, hence $G(Y) \subseteq G(Z) \subseteq Z$, using monotonicity. It follows that Y is a prefixpoint. To show that Y is in fact a fixpoint, it is enough to show that $G(Y)$ is a prefixpoint, since this will entain $Y \subseteq G(Y)$. We have, indeed, $G(G(Y) \subseteq G(Y)$, by monotonicity. $\square$

This theorem gives us two reasoning tools. The *induction principle* asserts that to show $\mathit{lfp}(\mu) \subseteq R$ is is enough to show $\mu(R) \subseteq R$ (note that it is not necessary!). The *coinduction principle* asserts that to show $R \subseteq \mathit{gfp}(\mu)$ it is enough to show $\mu(R) \subseteq R$. In practice, the induction principle is often used for a subset of $\mathit{lfp}(\mu)$, and then serves to show that $R = \mathit{lfp}(\mu)$. In practice, the principle of coinduction is used to show that some element x is in $\mathit{gfp}(\mu)$, and for this it is enough to find a postfixpoint R such that $x \in R$.

Often, the set X is itself defined by induction (or coinduction, see section 2.4). This always requires some kind of "bootstrapping", i.e. that we have somehow already a more general picture in which the set already exists, so that we content ourselves with characterizing it. The prototypical example is the set of natural numbers, which is the least set of numbers closed under the following rules: 0 is a natural number, and if n is a natural number then $n + 1$ is a natural number. To illustrate the idea of "bootstrapping", let us take a larger set X', e.g., the set of rationals, that we suppose to be already built (any set of numbers containing the natural numbers would do for this informal argument). Then the natural numbers are by specification the least fixed point of $G : \mathcal{P}(X') \to \mathcal{P}(X')$ defined by $G(Y) = \{0\} \cup \{n+1 \mid n \in Y\}$. And the associated principle of induction is the usual mathematical induction. To show that a predicate P (identified to the subset of X where it holds) is true for all n ($lfp(G_K) \subseteq P$), it is enough to show that $G_K(P) \subseteq P$, which spells out as $P(0)$ and $P(n) \Rightarrow P(n+1)$ (closure forwards by the rules).

2.2 Continuity and cocontinuity

We consider monotone operators from $\mathcal{P}(X)$ to $\mathcal{P}(X)$ (the set of subsets of X) defined via a set K of rules, each of the form (Y, x), with $Y \subseteq X$ and $x \in X$, or, graphically (for $Y = \{x_1, \ldots, x_n\}$ finite):

$$\frac{x_1, \ldots, x_n}{x}$$

We set $G_K(R) = \{x \in X \mid \exists\, (Y, x) \in K \;\; Y \subseteq R\}$. Here are some examples.

- The operator G defining natural numbers (cf. section 2.1) is G_K for the following K:

$$\frac{}{0} \qquad \frac{n}{n+1}$$

- The transitive closure of a relation can be defined inductively by the following rules:

$$\frac{}{(a, a)} \qquad\qquad a \to a' \;\; \frac{}{(a, a')}$$

$$\frac{(a_1, a_2) \quad (a_2, a_3)}{(a_1, a_3)}$$

  Call $\to^*$ the least fixpoint of the associated operator. Then one can show that $\to^* = \{(a, a') \mid \exists\, n \geq 0 \;\; a = a_0 \to a_1 \to \ldots \to a_n = a'\}$ For $\subseteq$ one uses the induction principle relative to the set of rules just introduced (this is called *rule induction*). For the other inclusion, one uses mathematical induction.

Back to our main theme, bisimulation is defined by a set of rules. Namely, we take K to be the set of all $(Y_{P,Q,f,g}, (P,Q))$, where f (resp. g) is any function mapping each pair μ, P' such that $P \stackrel{\mu}{\to} P'$ to a process $f(\mu, P')$ such that $Q \stackrel{\mu}{\to} f(\mu, P')$ (resp. each pair μ, Q' such that $Q \stackrel{\mu}{\to} Q'$ to a process $g(\mu, Q')$ such that $P \stackrel{\mu}{\to} g(\mu, Q')$) , and where

$$Y_{P,Q,f,g} = \{(P', f(\mu, P')) \mid P \stackrel{\mu}{\to} P'\} \cup \{(g(\mu, Q'), Q') \mid Q \stackrel{\mu}{\to} Q'\} .$$

Then it is easily checked that $\mathcal{R} \subseteq G_K(\mathcal{R})$ means exactly that $\mathcal{R}$ is a bisimulation.

What do we gain by knowing that $\sim$, first defined as the union of all bisimulations, is actually the largest fixpoint of some operator? First, that $\sim$ itself is a bisimulation, second that it is a prefixpoint, not only a postfixpoint.

Definition 2.4 *A function $G : \mathcal{P}(X) \to \mathcal{P}(X)$ is called* continuous *if it preserves increasing chains, i.e. for any sequence $(X_n)_{n\in\omega}$ such that $X_n \subseteq X_{n+1}$ for all n we have $G(\bigcup_{n\in\omega} X_n) = \bigcup_{n\in\omega} G(X_n)$. G is called* anti-continuous *is it preserves decreasing chains.*

Proposition 2.5 *If all the Y's in rules of K are finite, then G_K is continuous.*

Proof. We have to prove $G(\bigcup_{n\in\omega} X_n) \subseteq \bigcup_{n\in\omega} G(X_n)$, since the other inclusion is obvious by monotonicity. Let $x \in G(\bigcup_{n\in\omega} X_n)$, and hence let $Y \subseteq \bigcup_{n\in\omega} X_n$ be such that $(Y, x) \in K$. Since by assumption Y is finite, then for n large enough we have $Y \subseteq X_n$, and hence $x \in G(X_n) \subseteq \bigcup_{n\in\omega} G(X_n)$. $\square$

Proposition 2.6 *If, for all x, $\{(Y \mid (Y, x) \in K\}$ is finite, then G_K is anti-continuous.*

Proof. We have to prove $\bigcap_{n\in\omega} G(X_n) \subseteq G(\bigcap_{n\in\omega} X_n)$. Let $x \in \bigcap_{n\in\omega} G(X_n)$ and hence let$Y_n \subseteq X_n$ be such that $(Y_n, x) \in K$, for all n. It follows from the assumption that there exists Y such that $Y = Y_n$ for infinitely many Y_n's. Hence $Y \subseteq X_n$ for infinitely many n's, from which it follows that Y is included in all X_n's, and hence $x \in G(\bigcap_{n\in\omega} X_n)$. In full, the argument goes as follows. We have $\forall m \;\; \exists n \geq m \;\; Y = Y_n$, from which we get $\forall m \;\; \exists n \geq m \;\; Y \subseteq X_n \subseteq X_m$, which boils down to $\forall m \;\; Y \subseteq X_m$. $\square$

The benefit of continuity and anti-continuity is that they allow for a more tractable characterization of the least and greatest fixpoint, respectively.

Proposition 2.7 *If G is continuous, then $lfp(G) = \bigcup_{n\in\omega} G^n(\emptyset)$. If G is anti-continuous, then $gfp(G) = \bigcap_{n\in\omega} G^n(X)$.*

PROOF. Noticing that G is anti-continuous if and only if it is continuous with respect to $(\mathcal{P}(X), \supseteq)$, it is enough to prove the first part of the statement. We first notice that $\emptyset \subseteq G(\emptyset)$ and then $G(\emptyset) \subseteq G(G(\emptyset))$ by monotonicity, etc..., so that the sequence $(G^n(\emptyset))_{n\in\omega}$ is increasing That $\bigcup_{n\in\omega} G^n(\emptyset)$ is a fixpoint is an immediate consequence of the continuity of G. We show that $G^n(\emptyset) \subseteq Z$ for all n and Z such that Z is prefixpoint of G, by induction on n. Clearly, $G^0(\emptyset) = \emptyset \subseteq Z$. And if $G^n(\emptyset) \subseteq Z$, then $G^{n+1}(Z) \subseteq G(Z) \subseteq Z$. □

Proposition 2.8 *In CCS with finite sums, the bisimulation operator G_K is anti-continuous.*

PROOF. The key properties that we use is that for fixed μ the relation $\stackrel{\mu}{\to}$ is *image-finite*, i.e., for all P there are only finitely many P''s such that $P \stackrel{\mu}{\to} P'$, and that for fixed P there are only finitely many actions that P can perform, i.e., $\{\mu \mid \exists P' \;\; P \stackrel{\mu}{\to} P'\}$ is finite. Both properties are easily proved by induction on the definition of the labelled operational semantics, using the finite sum assumption for the sum axiom. Hence, for fixed P, Q, the domain and codomain of f, g in all $Y_{P,Q,f,g}$'s are included in finite sets depending on P, Q only, hence there are finitely many such f, g. Thus, the condition in Proposition 2.6 holds. □

Note that in [5] sums are taken over arbitrary index sets, for the sake of encoding value-passing.

2.3 Convergence and divergence in the λ-calculus

We denote by Λ_0 the set of closed λ-terms. Convergence $e \Downarrow \lambda x.e'$ in the λ-calculus is defined as the lfp of the following operator acting on subsets of $\Lambda_0 \times \Lambda_0$:

$$\begin{array}{rcl} F(X) & = & \{(\lambda x.e, \lambda x.e) \mid \lambda x.e \in \Lambda_0\} \cup \\ & & \{(e_1 e_2, e') \mid (e_1, \lambda x.e_0) \in X \text{ and } (e_0[x \leftarrow e_2], e') \in X\} \end{array}$$

Divergence $e \Uparrow$ is defined as the gfp of the following operator acting on subsets of Λ_0:

$$G(Y) = \{e_1 e_2 \mid e_1 \in Y \text{ or } (e_1 \Downarrow \lambda x.e_0 \text{ and } e_0[x \leftarrow e_2] \in Y)\}$$

Let us prove formally that $e \Uparrow$ iff $\nexists e' \; e \Downarrow e'$. We prove $\nexists e' \; e \Downarrow e' \Rightarrow e \Uparrow$ by coinduction. Let $Y = \{e \mid \nexists e' \; e \Downarrow e'\}$. We have to prove $Y \subseteq G(Y)$. If $e \in Y$, then e must be an application $e_1 e_2$ (as otherwise it would be an abstraction, and abstractions converge, as $(\lambda x.e, \lambda x.e) \in F(\Downarrow) = \Downarrow$). We have to show that $e_1 \in Y$ or ($e_1 \Downarrow \lambda x.e_0$ and $e_0[x \leftarrow e_2] \in Y$). If not, then $e_1 \Downarrow \lambda x.e_0$ and $e_0[x \leftarrow e_2] \Downarrow e'$ for some e'. Hence $(e_1 e_2, e') \in F(\Downarrow) = \Downarrow$, contradicting our assumption that $e_1 e_2 \in Y$.

Conversely, we show $(\exists e' \;\; e \Downarrow e') \Rightarrow e \not\Uparrow$ by induction on the size of the proof that $e \Downarrow e'$. The base case is $\lambda x.e \Downarrow \lambda x.e$. We cannot have $\lambda x.e \in \Uparrow = G(\Uparrow)$ since $G(\Uparrow)$ contains only applications. The induction case is

$$\frac{e_1 \Downarrow \lambda x.e_0 \quad e_0[x \leftarrow e_2] \Downarrow e'}{e_1 e_2 \Downarrow e'}$$

By induction, we know that $e_1 \not\Uparrow$ and that $e_0[x \leftarrow e_2] \not\Uparrow$. This forbids $e_1 e_2 \in \Uparrow = G(\Uparrow)$ by definition of G.

What about the lfp of G? It is the empty set, as $G(\emptyset) = \emptyset$.

As we have seen, the lfp is actually the least prefixpoint, i.e., $F(X) \subseteq X$: if $(e_1, \lambda x.e_0) \in X$ and $(e_0[x \leftarrow e_2], e') \in X$, then also $(e_1 e_2, e') \in X$, i.e. a prefixpoint is *closed forwards* by the rules. Similarly, the gfp is actually the greatest postfixpoint, i.e., $Y \subseteq G(Y)$: if $e \in Y$, then $e = e_1 e_2$, and either $e_1 \in Y$ or $e_1 \Downarrow \lambda x.e_0$ and $e_0[x \leftarrow e_2] \in Y$, i.e. a postfixpoint is *closed backwards* by the rules, which are:

$$\frac{e_1 \Uparrow}{e_1 e_2 \Uparrow} \qquad \frac{e_1 \Downarrow \lambda x.e_0 \quad e_0[x \leftarrow e_2] \Uparrow}{e_1 e_2 \Uparrow}$$

2.4 Finite and infinite lists

We take as universe the set T^∞ of all finite and infinite terms written over an infinite signature (e.g. all chains of characters are allowed as function symbols). Define

$$F(X) = \{nil\} \cup \{cons(a, l) \mid l \in X\}$$

or, equivalently, we can present $F = G_K$ with the following set K of rules:

$$\frac{}{nil} \qquad \frac{l}{cons(a, l)}$$

It should be clear that F is anti-continuous, by Proposition 2.6. The set of finite lists is the lfp of $F(X)$. Let us show that the gfp is the set L^∞ of finite or infinite lists. By coinduction, to prove $L^\infty \subseteq gfp(F)$, it is enough to prove $L^\infty \subseteq F(L^\infty)$. So let l be a finite or infinite list. It is either nil, in which case $l \in F(L^\infty)$, or it has the form $cons(a, l')$, where l' is also a finite or infinite list, which proves the claim. For the converse direction, we use the anti-continuity of F. It is easy to prove by induction on n that $F^n(T^\infty)$ is the union of the set L^n of all finite lists of length $< n$ and of the set $T^{\geq n}$ of terms of the form $cons(a_1, cons(a_2, \ldots cons(a_n, t') \ldots))$, where $t' \in T^\infty$. Now, suppose that $t \in gfp(F)$ and that t is not a finite list. Then we must have $t \in T^{\geq n}$ for all n, which means that t is an infinite list: it is an infinite term, which can be checked to have arbitrary large truncations which are lists, so this term does not have any internal node other than $cons$, since any internal node of an infinite tree is at finite depth.

Consider now

$$G(X) = \{cons(a, l) \mid l \in X\}$$

Then $gfp(G)$ is the set L^ω of infinite lists. The arguments above are adapted as follows. By coinduction, to prove $L^\omega \subseteq gfp(G)$, it is enough to prove $L^\omega \subseteq F(L^\omega)$. So let l be an infinite list. It must have the form $cons(a, l')$, where l' is also an infinite list, which proves the claim. For the converse direction $gfp(F) = \bigcap_{n \in \omega} G^n(T^\infty) \subseteq L^\infty$, it is easy to prove by induction on n that $G^n(T^\infty)$ is the set of terms of the form $cons(a_1, cons(a_2, \ldots cons(a_n, t') \ldots))$, where $t' \in T^\infty$. Hence, it t is in all the $G^n(T^\infty)$'s, then t is an infinite list.

3 Strong and weak bisimulations

3.1 Weak bisimulation

An equivalent formulation of strong bisimulation is by words of actions rather than by individual actions. That is, one replaces $\stackrel{\mu}{\rightarrow}$ by $\stackrel{s}{\rightarrow}$, where $s \in Act^+$ (recall that $Act = L \cup \overline{L} \cup \{\tau\}$). Using * to speak about this new notion of strong bisimulation, let us show that $\sim^* = \sim$. We show first that $\sim$ is a *-bisimulation. This follows from the following picture:

$$\begin{array}{ccc} P & \sim & Q \\ \downarrow & & \downarrow \\ & \sim & \\ \vdots & & \vdots \\ & \sim & \\ \downarrow & & \downarrow \\ P' & \sim & Q' \end{array}$$

where each square is completed thanks to $\sim$ being a bisimulation. Conversely, we prove that $\sim^*$ is a bisimulation. Let $P \sim^* Q$ and $P \xrightarrow{\mu} P'$. Then, μ being a particular s, we have by definition of $\sim^*$ that $Q \xrightarrow{\mu} Q'$ and $P' \sim^* Q'$ for some Q'.

The notion of weak bisimulation is best introduced from this level of granularity. Given a sequence s, we do not care about its τ-actions, thus we define $\hat{s}$ as follows:

$$\hat{\epsilon} = \epsilon \quad \hat{\tau} = \epsilon \quad \hat{\alpha} = \alpha \quad \hat{s\mu} = \hat{s}\hat{\alpha}$$

Then we define $P \stackrel{t}{\Rightarrow} Q$ if $P \xrightarrow{s} Q$ and $\hat{s} = t$ for some s. Weak bisimulation is defined via this new notion of reduction. It coincides with its one step version, where one step is either $\stackrel{\epsilon}{\Rightarrow}$ or $\stackrel{\alpha}{\Rightarrow}$. In full, a weak bisimulation is a relation $\mathcal{R}$ such that if $P \,\mathcal{R}\, Q$ and $P \stackrel{\rho}{\Rightarrow} P'$, then there exists Q' such that $QP \stackrel{\rho}{\Rightarrow} Q'$ and $P' \,\mathcal{R}\, Q'$, and conversely. Here, ρ ranges over $L \cup \underline{L} \cup \{\epsilon\}$. The proof is as above.

The following is yet another presentation of weak bisimulation, which is the standard presentation. It is less symmetric, but easier to check. A weak bisimulation is a relation $\mathcal{R}$ such that if $P \mathcal{R} Q$ and $P \xrightarrow{\mu} P'$, then there exists Q' such that $QP \stackrel{\hat{\mu}}{\Rightarrow} Q'$ and $P' \,\mathcal{R}\, Q'$, and conversely. We use ^{s} for denoting the version given just before, and $\approx$ for the standard version. We prove that $\approx$ and $\approx^s$ coincide. That $\approx$ is a ^{s}-bisimulation follows from the following pictures:

$$
\begin{array}{ccc}
P & \approx & Q \\
\downarrow_{\tau} & & \downarrow_{\tau^*} \\
 & \approx & \\
\vdots & & \vdots \\
 & \approx & \\
\downarrow_{\tau} & & \downarrow_{\tau^*} \\
P' & \approx & Q'
\end{array}
\qquad\qquad
\begin{array}{ccc}
P & \approx & Q \\
\downarrow_{\tau^*} & & \downarrow_{\tau^*} \\
 & \approx & \\
\downarrow_{\alpha} & & \Downarrow^{\alpha} \\
 & \approx & \\
\downarrow_{\tau^*} & & \downarrow_{\tau^*} \\
P' & \approx & Q'
\end{array}
$$

where the left column of the right diagram expands $\stackrel{\alpha}{\Rightarrow}$ while the right column folds into $\stackrel{\alpha}{\Rightarrow}$. Conversely, that $\approx^s$ is a bisimulation follows rightaway from the fact that $\stackrel{\mu}{\to} \subseteq \stackrel{\hat{\mu}}{\Rightarrow}$.

3.2 Some bisimulation equivalences

- $P \approx \tau \cdot P$. This follows from the fact if $P \stackrel{\mu}{\to} P'$ then $\tau \cdot P \stackrel{\tau}{\to} P \stackrel{\mu}{\to} P'$, while conversely the only action $\tau \cdot P \stackrel{\tau}{\to} P$ is matched by $P \stackrel{\epsilon}{\Rightarrow} P$.
- $P + \tau \cdot P \approx P$. This follows from the following observations: if $P \stackrel{\mu}{\to} P'$, then $P + \tau \cdot P \stackrel{\mu}{\to} P'$, and conversely we have to cope also with the reduction $P + \tau \cdot P \stackrel{\tau}{\to} P$ which is matched by $P \stackrel{\hat{\tau}}{\Rightarrow} P$.
- $let\ \vec{K} = \vec{P}\ in\ K_i \approx P_i[\vec{K} \leftarrow let\ \vec{K} = \vec{P}\ in\ \vec{K}]$. This follows immediately from the fact that the two sides have exactly the same transitions to the same terms.

Bisimulations are often used to prove that a process meets a specification. Both the process and the specification are processes, and we prove them bisimilar. We give two examples.

- Consider the following processes:

$$
\begin{array}{lll}
H = g \cdot H' & J = in \cdot S & K = in \cdot D \\
H' = p \cdot H & S = \overline{g} \cdot U & D = \overline{out} \cdot K \\
 & U = \overline{p} \cdot F & \\
 & F = \overline{out} \cdot J &
\end{array}
$$

  Here (cf. Milner-CC[Chapters 1 and 5]), $H, H', J, S, U, F, K, D, g, p$ are abbreviations for “hammer”, “busyhammer”, “jobber”, “start”, "usehammer", "finish", "strongjobber", "doing", "gethammer", and "puthammer", respectively. We show $(\nu g,h)(J \mid J \mid H) \approx K \mid K$. The right process stands for a specification (two jobbers, as viewed by the outside) while the first is an "implementation" (we come to know a bit more on how the work can actually be implemented). Their first actions are exactly the same, so we readily include the following pairs in our candidate weak bisimulation $\mathcal{R}$:

$$\begin{array}{ll} ((\nu g,h)(J \mid J \mid H), K \mid K) & ((\nu g,h)(S \mid J \mid H), D \mid K) \\ ((\nu g,h)(J \mid S \mid H), K \mid D) & ((\nu g,h)(S \mid S \mid H), D \mid D) \end{array}$$

Now the key observation is that the only possible sequence of actions out of, say, $(\nu g,h)(S \mid S \mid H)$ is:

$$(\nu g,h)(S \mid S \mid H) \xrightarrow{\tau} (\nu g,h)(S \mid U \mid H') \xrightarrow{\tau} (\nu g,h)(S \mid U \mid H') \xrightarrow{\overline{out}} (S \mid J \mid H)$$

From this it follows that $\mathcal{R}$ completed by all the pairs

$$\begin{array}{ll} ((\nu g,h)(S \mid U \mid H'), D \mid D) & ((\nu g,h)(S \mid F \mid H), D \mid D) \\ ((\nu g,h)(J \mid U \mid H'), K \mid D) & ((\nu g,h)(J \mid F \mid H), K \mid D) \\ ((\nu g,h)(U \mid J \mid H'), D \mid K) & ((\nu g,h)(F \mid J \mid H), D \mid K) \end{array}$$

is a weak bisimulation.

- The following process *SPEC* is a specification of a simple slot machine:

  $$\mathit{SPEC}\langle x\rangle = s \cdot (\tau \cdot \bar{l} \cdot \mathit{SPEC}\langle x+1\rangle + \Sigma_{1\le y\le x+1}\overline{w} \cdot \mathit{SPEC}\langle x+1-y\rangle)\ .$$

  Let IO, B, D be given as follows:

  $$\begin{array}{l} \mathit{IO} = s \cdot \bar{b} \cdot (L \cdot \bar{l} \cdot \mathit{IO} + R(y) \cdot \overline{w}\langle y\rangle \cdot \mathit{IO}) \\ B\langle x\rangle = b \cdot \overline{\mu}\langle x+1\rangle \cdot \lambda(y) \cdot B\langle y\rangle \\ D = \mu(z) \cdot (\overline{L} \cdot \overline{\lambda}\langle z\rangle \cdot D + \Sigma_{1\le y\le z}\overline{R}\langle y\rangle \cdot \overline{\lambda}\langle z-y\rangle \cdot D) \end{array}$$

  Our objective is to prove $\mathit{SPEC}\langle n\rangle \approx \mathit{SM}\langle n\rangle$, where

  $$\mathit{SM}\langle n\rangle = (\nu b,\mu,\lambda,L,R)(\mathit{IO} \mid B\langle n\rangle \mid D)$$

  (Think of s, b, μ, λ, R, l, L, and w as abbreviations for "slot", "bank", "max", "left" "release", "loss", "lost", and "win", respectively.) By algebraic laws, we have (we give indexes to spell out how the τ actions arise):

$$SM\langle n\rangle$$

$$\approx$$

$$s \cdot ((\nu b, \mu, \lambda, L, R)((\overline{b} \cdot (L \cdot \overline{l} \cdot IO + R(y) \cdot \overline{w}\langle y\rangle \cdot IO)) \mid B\langle n\rangle \mid D))$$

$$\approx^b$$

$$s \cdot \tau \cdot ((\nu b, \mu, \lambda, L, R) \begin{array}{l} (L \cdot \overline{l} \cdot IO + R(y) \cdot \overline{w}\langle y\rangle \cdot IO \\ \mid \overline{\mu}\langle n+1\rangle \cdot \lambda(y) \cdot B\langle y\rangle \mid D)) \end{array}$$

$$\approx^\mu$$

$$s \cdot \tau \cdot \tau \cdot P'$$

where

$$P' = (\nu b, \mu, \lambda, L, R) \begin{array}{l} (L \cdot \overline{l} \cdot IO + R(y) \cdot \overline{w}\langle y\rangle \cdot IO \\ \mid \lambda(y) \cdot B\langle y\rangle \\ \mid \overline{L} \cdot \overline{\lambda}\langle n+1\rangle \cdot D + \Sigma_{1\le y\le n+1}\overline{R}\langle y\rangle \cdot \overline{\lambda}\langle n+1-y\rangle \cdot D) \end{array}$$

By the equations proved above, we have $SM\langle n\rangle \approx s \cdot P'$. Now let us define:

$$P'_0 = (\nu b, \mu, \lambda, L, R) \begin{array}{l} (\overline{l} \cdot IO \\ \mid \lambda(y) \cdot B\langle y\rangle \\ \mid \overline{\lambda}\langle n+1\rangle \cdot D) \end{array}$$

and for $1 \le y \le n+1$:

$$P'_y = (\nu b, \mu, \lambda, L, R) \begin{array}{l} (\overline{w}\langle y\rangle \cdot IO \\ \mid \lambda(y) \cdot B\langle y\rangle \\ \mid \overline{\lambda}\langle n+1-y\rangle \cdot D) \end{array}$$

We have $P' \approx \tau.P'_0 + \Sigma_{1\le y\le n+1}\tau \cdot P'_y$ by the expansion law. Next we have, by expansion again, $P'_0 \approx \overline{l} \cdot (\tau \cdot SM\langle n+1\rangle + s \cdot \tau \cdot P'') + \tau \cdot \overline{l} \cdot SM\langle n+1\rangle$, where P'' is such that $SM\langle n+1\rangle \approx s \cdot P''$. So we can write:

$$P'_0 \approx \overline{l} \cdot (\tau \cdot s \cdot P'' + s \cdot \tau \cdot P'') + \tau \cdot \overline{l} \cdot s \cdot P'' \; .$$

Now we use the following bisimilarity equations:

$$\tau \cdot Q \approx Q \qquad Q + \tau \cdot Q \approx Q$$

and get the following simpler equivalence:

$$\begin{array}{rcl}P'_0 & \approx & \overline{l} \cdot (\tau \cdot s \cdot P'' + s \cdot P'') + \tau \cdot \overline{l} \cdot s \cdot P'' \\ & \approx & \overline{l} \cdot s \cdot P'' + \tau \cdot \overline{l} \cdot s \cdot P'' \\ & \approx & \overline{l} \cdot s \cdot P'' \\ & \approx & \overline{l} \cdot SM\langle n+1\rangle\end{array}$$

Similarly, we obtain $P'_y \approx \overline{w} \cdot SM\langle n+1-y\rangle$. We conclude by Proposition 3.6.

3.3 Up-to bisimulation

It may ease the work to relax a bit the notion of bisimulation, while still being able to use the resulting notion to prove that some processes are bisimmilar.

Definition 3.1 *A simulation up to $\sim$ is a relation $\mathcal{R}$ such that for all P, Q, if $P \mathcal{R}\, Q$ then for all μ, P' such that $P \xrightarrow{\mu} P'$ there exists Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \sim \mathcal{R} \sim Q'$. A* bisimulation *up to $\sim$ is a relation $\mathcal{R}$ such that $\mathcal{R}$ and $\mathcal{R}^{-1}$ are simulations up to $\sim$.*

Proposition 3.2 *If R is strong bisimulation up to $\sim$, then $R \subseteq \sim$.*

Proof. Since $R \subseteq (\sim R \sim)$, it is enough to show that $\sim R \sim$ is a bisumulation, which is easily seen from the following picture:

$$\begin{array}{ccccccccc} P & \sim & P_1 & & R & & Q_1 & \sim & Q \\ \downarrow_\mu & & \downarrow_\mu & & & & \downarrow_\mu & & \downarrow_\mu \\ P' & \sim & P'_1 & & \sim R \sim & & Q'_1 & \sim & Q' \end{array}$$

$\square$

3.4 Bisimulation is a congruence

We define $\sim^*$ inductively by the following rules:

$$\frac{P \sim Q}{P \sim^* Q} \quad \frac{P \sim^* Q \;\; Q \sim^* R}{P \sim^* R}$$

$$\frac{\forall\, i \in I \;\; P_i \sim^* Q_i}{\Sigma_{i\in I}\alpha_i \cdot P_i \sim^* \Sigma_{i\in I}\alpha_i \cdot Q_i} \quad \frac{P_1 \sim^* Q_1 \;\; P_2 \sim^* Q_2}{P_1 \mid P_2 \sim^* Q_1 \mid Q_2} \quad \frac{P \sim^* Q}{(\nu a)P \sim^* (\nu a)Q}$$

We have that $\sim^*$ is reflexive (because $\sim$ is reflexive). It is a congruence by construction, and it contains $\sim$ by construction. We show that $\sim^*$ is a bisimulation, which will entail that $\sim^* \subseteq \sim$ and hence that $\sim^* = \sim$. We prove this by rule induction. If $P \sim Q$ and $P \xrightarrow{a} P'$, then $Q \xrightarrow{a} Q'$ for some Q' such that $P' \sim Q'$ and hence a fortiori $P' \sim^* Q'$. Let $\Sigma_{i\in I}\alpha_i \cdot P_i \xrightarrow{\alpha_i} P_i$. Then $\Sigma_{i\in I}\alpha_i \cdot Q_i \xrightarrow{\alpha_i} Q_i$, and we have $P_i \sim^* Q_i$ by the assumption of the rule. If $(\nu a)P \xrightarrow{\beta} (\nu a)P'$, then $P \xrightarrow{\beta} P'$, and by induction we have $Q \xrightarrow{\beta} Q'$ for some Q' such that $P' \sim^* Q'$. Hence $(\nu a)Q \xrightarrow{\beta} (\nu a)Q'$ and $(\nu a)P' \sim^* (\nu a)Q'$. For parallel composition, one proceeds in the same way. We only consider the synchronisation case. If $P_1 \xrightarrow{a} P_1'$ and $P_2 \xrightarrow{\overline{a}} P_2'$, then by induction there exists Q_1' such that $Q_1 \xrightarrow{a} Q_1'$ and $P_1' \sim^* Q_1'$. Similarly, there exists Q_2' such that $Q_2 \xrightarrow{\overline{a}} Q_2'$ and $P_2' \sim^* Q_2'$. Hence we have $Q_1 \mid Q_2 \xrightarrow{\tau} Q_1' \mid Q_2'$ and $P_1' \mid P_2' \sim^* Q_1' \mid Q_2'$.

The same proof goes through if we split prefix from sum. The rules are then

$$\frac{P_i \xrightarrow{\alpha} P_i'}{\Sigma_{i\in I}P_i \xrightarrow{\alpha} P_i'} \qquad \frac{}{\alpha \cdot P \xrightarrow{\alpha} P}$$

(Decorating for an instant with ' the system obtained by substituting $\Sigma_{i\in I}\mu_i \cdot P_i \xrightarrow{\mu_i} P_i$ with these two rules, it is easily checked that $\rightarrow$ and $\rightarrow'$ coincide on the sublanguage where summands have to be prefixes. Hence bisimulation on the restricted language is the restriction of bisimulation on the larger one.) Now consider weak bisimulation on the extended language. The proof so far goes through provided we can show:

$$\frac{P \stackrel{\hat{\alpha}}{\Rightarrow} P'}{(\nu a)P \stackrel{\hat{\alpha}}{\Rightarrow} (\nu a)Q'} \qquad \frac{Q_1 \stackrel{\hat{\alpha}}{\Rightarrow} Q_1'}{Q_1 \mid Q_2 \stackrel{\hat{\alpha}}{\Rightarrow} Q_1' \mid Q_2} \qquad \frac{Q_1 \stackrel{\hat{a}}{\Rightarrow} Q_1' \qquad Q_2 \stackrel{\hat{\bar{a}}}{\Rightarrow} Q_2'}{Q_1 \mid Q_2 \stackrel{\hat{\tau}}{\Rightarrow} Q_1' \mid Q_2'}$$

and indeed it works smoothly. But if we had unguarded sums, we would also have to show

$$\frac{Q_1 \stackrel{\hat{\alpha}}{\Rightarrow} Q_1'}{Q_1 + Q_2 \stackrel{\hat{\alpha}}{\Rightarrow} Q_1'}$$

which does *not* hold (take $\alpha = \tau$ and $Q_1' = Q_1$). This is only a heuristic argument, but the reader may check that the following is indeed a counter-example: $\tau \cdot a \cdot 0 + b \cdot 0 \not\approx a \cdot 0 + b \cdot 0$.

Summarizing, weak bisimulation is a congruence for guarded sums, parallel composition, and restriction. It is not a congruence for sums. The counter-example has nothing to do with the fact that the langage with general sums allows new expressions such as $P + (Q \mid R)$ since the counter-example above is taken in the sublanguage. The point is that sum really refuses the congruence, while guarded sum as an atomic operation accepts it. As for recursion, we have the following results ([5][4.5,7.3]).

Definition 3.3 *A process variable K is weakly guarded in P if each occurrence of K is within some subterm of the form $\mu \cdot P'$ of P.*

Proposition 3.4 *If $\vec{K} = \vec{P}$ is a system of equations where all K's are weakly guarded in all P's, and if $\vec{Q}$ and $\vec{R}$ are solutions of the system in the sense that $\vec{Q} \sim P[\vec{K} \leftarrow \vec{Q}]$ and $\vec{R} \sim P[\vec{K} \leftarrow \vec{R}]$, then $\vec{P} \sim \vec{R}$.*

PROOF (INDICATION). One shows that the set of all pairs

$$(S[\vec{K} \leftarrow \vec{Q}], (S[\vec{K} \leftarrow \vec{R}]) \ ,$$

where S is arbitrary, is a bisimulation. The key to the proof is the observation that when K is weakly guarded in some process P, and if $P[K \leftarrow Q] \stackrel{\mu}{\rightarrow} T$ for some Q and T, then T has the form $P'[K \leftarrow Q]$ for some P' such that $P \stackrel{\mu}{\rightarrow} P'$ (and hence $P[K \leftarrow Q'] \stackrel{\mu}{\rightarrow} P'[K \leftarrow S']$ for any other Q'). $\square$

Definition 3.5 *A process variable K is guarded in P if each occurrence of K is within some subterm of the form $\alpha \cdot P'$ of P. A process variable K is guarded in P if no occurrence of K is within some subterm of P which is not a sum.*

Proposition 3.6 *If $\vec{K} = \vec{P}$ is a system of equations where all K's are guarded and sequential in all P's, and if $\vec{Q}$ and $\vec{R}$ are solutions of the system in the sense that $\vec{Q} \approx P[\vec{K} \leftarrow \vec{Q}]$ and $\vec{R} \approx P[\vec{K} \leftarrow \vec{R}]$, then $\vec{P} \approx \vec{R}$.*

PROOF (INDICATION). One shows that the set of all pairs

$$(S[\vec{K} \leftarrow \vec{Q}], (S[\vec{K} \leftarrow \vec{R}]) ,$$

where S is any process in which all the K's are sequential, is a bisimulation. $\square$

3.5 Logical characterization of bisimulation

The set of formulas is defined by

$$A := T \mid A \wedge B \mid \neg A \mid \langle a \rangle .$$

Let L_n the subset of formulas with depth of at most n of modal operators $\langle a \rangle$, and $L_n(P) = \{A \in L_n \mid P \models L_n\}$. Notice that there are no propositional variables, hence every formula of L_0 can be rewritten to T or $\neg T$ by the rules $T \wedge T \to T$, $\neg\neg T \to T$ etc. It is then easy to check that any two formulas A and B that are proved equal by these equations are such that $P \models A$ iff $P \models B$ for any P. Hence, for any formula $A \in L_0$, we have, for any P, Q, $P \models A$ iff $Q \models A$. Indeed, if A rewrites to T, then both $P \models A$ and $Q \models A$ are true, and if A rewrites to $\neg T$, then both $P \models A$ and $Q \models A$ are false. This proves that $P \sim_0 Q$ iff $L_0(P) = L_0(Q)$. Now we show that $P \not\sim_{n+1} Q$ implies $L_{n+1}(P) \neq L_{n+1}(Q)$ by induction on n. Since $P \not\sim_{n+1} Q$, there exists a, P' such that for all $Q'_1, \ldots Q'_n$ (notice that we are using image-finiteness here) such that $Q \xrightarrow{a} Q'$ we have $P' \neg \sim_n Q'_i$ for all i. Now, we have by induction $L_n(P') \neq L_n(Q'_i)$. Hence there exists A_i in $L_n(P')$ not in $L_n(Q'_i)$ or there exists B in $L_n(Q'_i)$ not in $L_n(P')$. But in the latter case we can take $\neg B$, hence we may assume that there exists A_i in $L_n(P')$ not in $L_n(Q'_i)$. Let $A = A_1 \wedge \ldots \wedge A_n$. Then $P' \models A$, and since $Q'_i \not\models A_i$ we have a fortiori $Q'_i \not\models A$ for all i. From there it follows that $P \models \langle a \rangle A$ and that $Q \not\models \langle i \rangle A$.

We now prove conversely that if $P \sim_{n+1} Q$ then $L_{n+1}(P) = L_{n+1}(Q)$. Let $A \in L_{n+1}(P)$. We proceed by structural induction on A. The only non-trivial case is $A = \langle a \rangle B$. Since $P \models A$, there exist a, P' such that $P \xrightarrow{a} P'$ and $P‘ \models B$. Since $P \sim_{n+1} Q$, there exists Q' such that $Q \xrightarrow{a} Q'$ and $P' \sim_n Q'$. By induction we get $Q' \models B$ and hence $Q \models A$.

How should we adapt this to the situation of infinite sums of processes, which lead to non-image finiteness? We have to go to infinite conjunctions, and to ordinals. Note that ordinals are needed on both sides of the equivalence $P \sim_\kappa Q$ iff $L_\kappa(P) = L_\kappa(Q)$. On the left side, this is because the non image-finiteness entails non-continuity of the operator of which $\sim$ is a fixpoint, while on the right side the depth of a formula may go beyond ω because of infinite branching, as the depth of a sum is the sup of the depths. In this way we may reach depth $\omega = \sup\{1, \ldots, n, \ldots\}$.

3.6 Finitary bisimulation

Proposition 3.7 *In finitary CCS, strong bisimulation is the following equational theory:*

$$
\begin{array}{ll}
(1) & \Sigma_{i\in I}\mu_i \cdot P_i = \Sigma_{i\in I}\mu_{f(i)} \cdot P_{f(i)} \quad (f \textit{ permutation}) \\
(2) & \Sigma_{i\in I}\mu_i \cdot P_i + P_j = \Sigma_{i\in I}\mu_i \cdot P_i \quad (j \in I) \\
(3) & P \mid Q = \Sigma\{\mu \cdot (P' \mid Q) \mid P \xrightarrow{\mu} P'\} + \Sigma\{\mu \cdot (P \mid Q') \mid Q \xrightarrow{\mu} Q'\} \\
 & \qquad\qquad +\Sigma\{\tau \cdot (P' \mid Q' \mid P \xrightarrow{\alpha} P' \textit{ and } Q \xrightarrow{\overline{\alpha}} Q'\} \\
(4) & (\nu a)\,(\Sigma_{i\in I}\mu_i \cdot P_i) = \Sigma_{\{}j \in I \mid \mu_j \neq a, \overline{a}\}\mu_j \cdot ((\nu a).P_j)
\end{array}
$$

Proof. Given a process P, we systematically and repeatedly apply to it the rules (3) and (4) considered as rewrite rules, until none of the two rules can be applied. The process R reached cannot contain a parallel composition, otherwise rule (3) would apply. Suppose that R contains a restriction $(\nu a)Q$, take it of minimal size (the size of a term is the number of symbols in the term, or, in a more structured way, it is the number of nodes in the tree representation of the term). Then Q cannot be a parallel composition (since there are no more) and cannot be a restriction either, by minimality. Hence it is a sum but then rule (4) applies, contradition.

We now show that this procedure of applying rules (3) and (4) (in any order) always terminates. We use the multi-set ordering, which is well-founded, i.e. it has no infinit strictly decreasing chain. Recall that a multi-set is a set with repetitions, for example $\{3, 3, 5, 2\}$. We otains a smaller multi-set by replacing one of its elements by a finite multi-set all of whose elements are smaller (in the sense of the order between numbers). For example. taking one of the 3's and replacing it by $\{2,2,1\}$, we get $\{2,2,1,3,5,2\} < \{3,3,5,2\}$. We associate with a process P the multi-set of the sizes of all its sub-terms $(\nu a)Q$ and $Q_1 \mid Q_2$. This multi-set decreases at each application of rules (3)-(4).

Thus, we have proved that each process is provably equal to a synchronization tree. Let $\Sigma_{i=1\ldots m}\alpha_i \cdot P_i \sim \Sigma_{j=m+1\ldots n}\alpha_j \cdot P_j$, then let $\leftrightharpoons$ be the equivalence relation on $\{1,\ldots n\}$ defined by $i \leftrightharpoons j$ iff $\alpha_i = \alpha_j$ and $P_i \sim P_j$. Bisimulation guarantees that each $\leftrightharpoons$ equivalence class contains at least one element of $[1,m]$ and at least one element of $[m+1,n]$. Now for each of the equivalence classes we pick one representative in $[1,m]$ and one in $[m+1,n]$. Call them $\alpha_{i_1},\ldots,\alpha_{i_k}$ and $\alpha_{j_1},\ldots,\alpha_{j_k}$. Then we have

$$\Sigma_{i=1\ldots m}\alpha_i.P \sim \Sigma_{l=1\ldots k}\alpha_{i_l} \cdot P_{i_l} \sim \Sigma_{l=1\ldots k}\alpha_{j_l} \cdot P_{j_l} \sim \Sigma_{j=m+1\ldots n}\alpha_j \cdot P_j$$

with $P_{i_l} \sim P_{j_l}$ for all l, so we can apply induction. (Notre that the finiteness of sums is crucial: we could not eliminate an infinite number of copies of one summand.) $\square$

The law (3) is called the *expansion law.*

Proposition 3.8 *In finitary CCS, weak bisimulation is the equational theory consisting of the equations for strong bisimulation plus the following equations: We add the following rules involving* τ*:*

$$\begin{array}{ll}(\tau_1) & P = \tau \cdot P \\ (\tau_2) & P + \tau \cdot P\ (+R) = \tau \cdot P\ (+R) \\ (\tau_3) & \alpha \cdot (\tau \cdot P + Q)\ (+R) = \alpha \cdot (\tau \cdot P + Q) + \alpha \cdot P\ (+R)\end{array}$$

(The parentheses around R *mean that the summand* R *is optional. Notice the absence of* $(+R)$ *in rule* (τ_1)*.)*

Proof. Since strong bisimulation is included in weak bisimulation, we can safely concentrate on synchronisation trees. So, throughout the proof, all processes will be synchronisation trees.

We have recalled the idempotence rule, which is the only equation for strong simulation which is relevant for synchronisation trees. Notice that the structural equivalence $\equiv$ on synchronisation trees means syntactical identity up to the order of the branches.

Lemma 3.9 *If $Q \stackrel{\hat{\mu}}{\Rightarrow} P$ and $P \not\equiv Q$, then $\vdash Q\,(+R) = Q + \mu \cdot P\,(+R)$.*

Proof. Let $Q \stackrel{s}{\rightarrow} P$ with $\hat{s} = \hat{\mu}$. Notice that the assumption $P \not\equiv Q$ amounts to $s \neq \epsilon$. We proceed by induction on the length of s. We omit the $+R$ which raises no difficulty.

1. $s = \tau s'$. Then $Q \equiv \tau \cdot Q_1\,(+Q_2)$, hence $\vdash Q = Q + Q_1$ by (τ_2). We can apply induction to Q_1 with s' where $s = \tau s'$ (and hence $\hat{s} = \hat{s'}$):

$$\vdash Q = Q + Q_1 = Q + (Q_1 + \mu \cdot P) = (Q + Q_1) + \mu \cdot P = Q + \mu \cdot P$$

2. $s = \alpha$. Then $Q \equiv \alpha \cdot P\,(+Q_2)$, and we conclude by idempotence.

3. $s = \alpha\tau s'$ (and hence $\mu = \alpha$).

   (a) $Q \equiv \alpha \cdot \tau^p \cdot P\,(+Q_2)$. Then $\vdash Q = \alpha.P\,(+Q_2) = Q + \alpha.P$ by (τ_1) and by idempotence.

   (b) $Q \equiv \alpha \cdot \tau^p \cdot (\tau \cdot Q_1 + Q_2)\,(+Q_3)$, by (τ_1) and (τ_3). Then $\vdash Q = \alpha \cdot (\tau \cdot Q_1 + Q_2)\,(+Q_3) = Q + \alpha \cdot Q_1$. We can apply induction to $\alpha \cdot Q_1$ with $s' = \alpha s''$ where $s = \alpha\tau^{p+1}s''$. $\square$

Lemma 3.10 *If P, S, and R are such that $R \stackrel{\epsilon}{\Rightarrow} S \stackrel{\epsilon}{\Rightarrow} P$, and if R' (resp. S') is the term obtained by replacing P by R in R (resp. in S), then $\vdash S' = R'$.*

Proof. One proves actually the following claim:

$$\forall T \ \ (R \stackrel{\epsilon}{\Rightarrow} T \stackrel{\epsilon}{\Rightarrow} P \Rightarrow \ vdash R = T')\,.$$

Hence in particular $\vdash S' = R$ and $\vdash R' = R$, whence the result. The claim is proved by a repeated application of lemma 3.9. We content ourselves with an example. Let $R = \tau \cdot S + R'$ and $S = \tau \cdot (P + \mu_1 \cdot R_1) + \mu_2 \cdot R_2$, hence $S' = \tau \cdot (R + \mu_1 \cdot R_1) + \mu_2 \cdot R_2$. Then $R \stackrel{\hat{\mu}_1}{\Rightarrow} R_1$ and $R \stackrel{\hat{\mu}_2}{\Rightarrow} R_2$, with $R \not\equiv R_1$ and $R \not\equiv R_2$. It follows that $\vdash R + \mu_1 \cdot R_1 = R$, so that $\vdash S' = \tau \cdot R + \mu_2 \cdot R_2$, and that $\vdash \tau \cdot R + \mu_2 \cdot R_2 = \tau \cdot R = R$, since a fortiori $\tau \cdot R \stackrel{\hat{\mu}_2}{\Rightarrow} R_2$. $\square$

We now prove $P \approx Q \ \Rightarrow \vdash P = Q$ by induction on the sum of the sizes of P and Q. We first examine the case where $P \equiv \tau \cdot P_1 + P'$ and $P_1 \approx Q$. Then by induction $\vdash P_1 = Q$. We show $\vdash P = P_1$, which will entail $\vdash P = Q$. Let $P' = \mu \cdot P_2 + P''$. Since $P \approx P_1$, there exists P'^1 such that $P_1 \stackrel{\hat{\mu}}{\Rightarrow} P'^1$ with $P_2 \approx P'^1$. Since P_2, P'^1 are subterms of P', P_1, respectively, the sum of their sizes is less than the size of P, so we can apply induction and we have $\vdash P_2 = P'^1$. We have:

$$\vdash \tau \cdot P_1 + P' \equiv \tau \cdot P_1 + \mu \cdot P_2 + P'' = \tau \cdot P_1 + \mu \cdot P'^1 + P'' = \tau \cdot P_1 + P''$$

where the last equality follows from lemma 3.9, noticing that with $P_1 \stackrel{\hat{\mu}}{\Rightarrow} P'^1$ implies $\tau \cdot P_1 \stackrel{\hat{\mu}}{\Rightarrow} P'^1$, with $P'^1 \not\equiv \tau \cdot P_1$. In this way, we absorb successively all the summands of P'. So we have $\vdash P \equiv \tau \cdot P_1 + P' = \tau \cdot P_1 = P_1$.

Hence, from now on, we may safely assume that there are functions f and g between the summands of $P = \Sigma_i \mu_i \cdot P_i$ and $Q = \Sigma_j \mu_j \cdot Q_j$ such that P_i is weak bisimilar to a subterm Q'_i of $Q_{f(i)}$, such that $Q \stackrel{\hat{\mu_i}}{\Rightarrow} Q'_i$, for all i, and Q_j is weak bisimilar to a subterm P'_j of $P_{g(j)}$, such that $P \stackrel{\hat{\mu_j}}{\Rightarrow} P'_j$, for all j. By induction we have $\vdash P_i = Q'_i$ and $\vdash Q_j = P'_j$, for all i, j. Let us fix P_i, so let us write without loss of generality $P \equiv \mu_1 \cdot P_1 + P'$. Let us consider $P'_{g(f(1))}$, and suppose that $g(f(1)) \neq 1$. Then $P'_{(f(1)}$ is a subterm of P' and $P' \stackrel{\hat{\mu}_{f(1)}}{\Rightarrow} P'_{f(1)}$. Moreover $P' \not\equiv P'_{f(1)}$ since the reduction starts as $P' \stackrel{\mu_{g(f(1))}}{\longrightarrow} P_{g(f(1))}$. Let P'' be the term obtained by replacing this subterm $P'_{f(1)}$ by $Q_{f(1)}$, so that we have $\vdash P'' = P'$ and $P'' \stackrel{\hat{\mu}_{f(1)}}{\Rightarrow} Q_{f(1)}$, with $Q_{f(1)} \not\equiv P''$. We show that $P'' \stackrel{\hat{\mu}_1}{\Rightarrow} Q'_1$. We know that $Q \stackrel{\hat{\mu}_1}{\Rightarrow} Q'_1$ and that Q'_1 is a subterm of $Q_{f(1)}$, and this can happen only in one of the following two cases.

1. $\mu_{f(1)} = \mu_1 \neq \tau$. Then $Q_{f(1)} \stackrel{\epsilon}{\Rightarrow} Q'_1$, and $P'' \stackrel{\hat{\mu}_{f(1)}}{\Rightarrow} Q_{f(1)}$ can be rephrased as $P'' \stackrel{\hat{\mu}_1}{\Rightarrow} Q_{f(1)}$, and hence $P'' \stackrel{\hat{\mu}_1}{\Rightarrow} Q'_1$.

2. $\mu_{f(1)} = \tau$. In that case $Q_{f(1)} \stackrel{\hat{\mu}_1}{\Rightarrow} Q'_1$, and $P'' \stackrel{\hat{\mu}_{f(1)}}{\Rightarrow} Q_{f(1)}$ reads as $P'' \stackrel{\epsilon}{\Rightarrow} Q_{f(1)}$, and hence $P'' \stackrel{\hat{\mu}_1}{\Rightarrow} Q'_1$.

Notice that in both cases the reduction from P'' to Q'_1 goes through $Q_{f(1)}$, hence a fortiori $Q'_1 \not\equiv P''$. Let now P''' be obtained by replacing Q'_1 by P_1 in P''. We have $\vdash P'' = P'''$, and $P''' \stackrel{\hat{\mu}_1}{\Rightarrow} P_1$, with $P_1 \not\equiv P'''$. By lemma 3.9 we have $\mu_1 \cdot P_1 + P''' = P'''$, hence $\vdash P = P'$, and since P' has a smaller size than P, we have $\vdash P' = Q$ by induction, and hence $\vdash P = Q$.

Thus, from now on, we can assume that f and g are inverse bijections. Note that this forces $\mu_{f(i)} = \mu_i$ for all i since $\mu_{f(i)} = \alpha \Rightarrow \mu_i = \alpha$ and

$\mu_i = \mu_{g(f(i))} = \alpha \Rightarrow \mu_{f(i)} = \alpha$. And it also follows that $Q_{f(i)} \stackrel{\epsilon}{\Rightarrow} Q'_i$ and that $P_i = P_{g(f(i))} \stackrel{\epsilon}{\Rightarrow} P'_{f(i)}$. Suppose, say, that $P'_{f(1)}$ is a strict subterm of P_1. Call R the term obtained by replacing $P'_{f(1)}$ by $Q_{f(1)}$ in P_1. By applying lemma 3.10 to $R, Q_{f(1)}, Q'_1$, we get $\vdash R' = S'$, where R' (resp. S') is R (resp. $Q_{f(1)}$) where Q'_1 has been replaced by P_1. Hence $\vdash P_1 = R = R' = S' = Q_{f(1)} = P'_{f(1)}$, and $\vdash P \equiv \mu_1 \cdot P_1 + P' = \mu_1 \cdot P'_{f(1)} + P'$, and we conclude by induction since $\mu_1 \cdot P'_{f(1)} + P'$ has smaller size than P.

Thus we can assume that $P'_{f(i)} \equiv P_i$ for all i, so that $P_i \approx Q_{f(i)}$ for all i. By induction we have $\vdash P_i = Q_{f(i)}$ for all i, and hence

$$\vdash P \equiv \Sigma_i \mu_i \cdot P_i = \Sigma_i \mu_i \cdot Q_{f(i)} = \Sigma_i \mu_{f(i)} \cdot Q_{f(i)} = \Sigma_j \mu_j \cdot Q_j \equiv Q\ .$$

□

4 π-calculus

4.1 Examples of reduction

Creation of an infinite chain of cells (simplified version of [9][1.2.3]). We set

$$P\langle h_0\rangle = I\langle h_0\rangle \mid G \mid H\langle h_0\rangle$$

where

$$\begin{array}{l} I\langle h\rangle = (\nu i)\ (\bar{i}h\ |!i(z) \cdot n(a) \cdot \overline{z}a \cdot \bar{i}a) \\ G = !(\nu p)\ \overline{n}p \cdot p(z) \cdot !\overline{p}z \\ H\langle h\rangle = h(z) \cdot !\overline{h}z \end{array}$$

H is an initial cell, G is a cell generator, and I controls the building of the chain. First, we have (setting $I' = !i(z) \cdot n(a) \cdot \overline{z}a \cdot \bar{i}a$):

$$I\langle h_0\rangle \rightarrow^i I_1 = (\nu i)\ (n(a) \cdot \overline{h_0}a \cdot \bar{i}a \mid I')$$

and hence $I\langle h_0\rangle \mid G \rightarrow I_1 \mid G$. Since $G \equiv ((\nu h_1)\ \overline{n}h_1 \cdot h_1(z) \cdot !\overline{h_1}z) \mid G$, we have:

$$\begin{array}{rcl} I_1 \mid G & \equiv & (\nu i, h_1)\ (n(a) \cdot \overline{h_0}a \cdot \bar{i}a \mid \overline{n}h_1 \cdot h_1(z) \cdot !\overline{h_1}z \mid I') \mid G \\ & \rightarrow^n & (\nu i, h_1)\ (\overline{h_0}h_1 \cdot \bar{i}h_1 \mid h_1(z) \cdot !\overline{h_1}z \mid I') \mid G \end{array}$$

and hence $I\langle h_0\rangle \mid G \mid H\langle h_0\rangle \rightarrow^* I_2 \mid G$, where $I_2 = (\nu i, h_1)\ (\overline{h_0}h_1 \cdot \bar{i}h_1 \mid h_1(z)\cdot !\overline{h_1}z \mid I') \mid H\langle h_0\rangle$. Then we have:

$$\begin{array}{lll} I_2 & \equiv & (\nu i, h_1)\ (\overline{h_0}h_1 \cdot \bar{i}h_1 \mid H\langle h_0\rangle \mid h_1(z)\cdot !\overline{h_1}z \mid I') \\ & \rightarrow^{h_0} & (\nu i, h_1)\ (\bar{i}h_1 \mid !\overline{h_0}h_1 \mid h_1(z)\cdot !\overline{h_1}z \mid I') \\ & = & (\nu i, h_1)\ (\bar{i}h_1 \mid !\overline{h_0}h_1 \mid H\langle h_1\rangle \mid I') \\ & \equiv & (\nu h_1)\ (I\langle h_1\rangle \mid H\langle h_1\rangle \mid !\overline{h_0}h_1) \end{array}$$

and hence, setting $C_0 = !\overline{h_0}h_1$:

$$P\langle h_0\rangle = I\langle h_0\rangle \mid G \mid H\langle h_0\rangle \rightarrow^* (\nu h_1)(I\langle h_1\rangle \mid G \mid H\langle h_1\rangle \mid C_0) = (\nu_1)(P\langle h_1\rangle \mid C_0)$$

In this way, setting $C_i = !\overline{h_i}h_{i+1}$ (with all the h_i different), we get:

$$P\langle h_0\rangle \rightarrow^* (\nu h_1, \ldots, h_{i+1})\ (P\langle h_i\rangle \mid C_i \mid C_{i-1} \mid \ldots \mid C_0)$$

So that, for example:

$$\begin{array}{l} P\langle h_0\rangle \mid h_0(x_1)\cdot x_1(x_2)\cdot \ldots \cdot x_i(x_{i+1}) \\ \quad \rightarrow^* (\nu h_j, \ldots, h_{i+1})(P\langle h_i\rangle \mid C_i \mid C_{i-1} \mid \ldots \mid C_j \mid x_j(x_{j+1})\cdot \ldots \cdot x_i(x_{i+1}) \end{array}$$

4.2 reduction versus τ-transitions

The following example illustrates the need and use of the OPEN and CLOSE rules. We suppose b not free in Q.

$$\begin{array}{lll} ((\nu b)\ \overline{a}b \cdot \overline{b}c) \mid a(x)\cdot x(y)\cdot Q & \equiv & (\nu b)\ (\overline{a}b \cdot \overline{b}c \mid a(x)\cdot x(y)\cdot Q) \\ & \rightarrow & (\nu b)\ (\overline{b}c \mid b(y)\cdot Q[x \leftarrow b]) \end{array}$$

We cannot write $((\nu b)\overline{a}b \cdot \overline{b}c) \xrightarrow{\overline{a}b} (\nu b)\overline{b}c$ since b is restricted. And indeed we do not want $((\nu b)\ \overline{a}b \cdot \overline{b}c) \mid a(x)\cdot x(y)\cdot Q \xrightarrow{\tau} ((\nu b)\ \overline{b}c) \mid b(y)\cdot Q[x \leftarrow b]$. In CCS by value, this situation does not occur since values belong to another world and hence are not restricted. The correct trransition is derived as follows:

$$\text{CLOSE}\ \dfrac{\text{OPEN}\ \dfrac{\overline{a}b \cdot \overline{b}c \xrightarrow{\overline{a}b} \overline{b}c}{(\nu b)\ \overline{a}b \cdot \overline{b}c \xrightarrow{\overline{a}(b)} \overline{b}c} \quad a(x)\cdot x(y)\cdot Q \xrightarrow{a(b)} b(y)\cdot Q[x \leftarrow b]}{((\nu b)\ \overline{a}b \cdot \overline{b}c) \mid a(x)\cdot x(y)\cdot Q \xrightarrow{\tau} (\nu b)\ (\overline{b}c \mid b(y)\cdot Q[x \leftarrow b])}$$

Or, using concretion and abstraction:

$$\text{GENCOMM}\ \frac{\dfrac{\overline{a}b \cdot \overline{b}c \xrightarrow{\overline{a}} \langle b \rangle \cdot \overline{b}c}{(\nu b)\, \overline{a}b \cdot \overline{b}c \xrightarrow{\overline{a}} (\nu b)\, \langle b \rangle \cdot \overline{b}c} \quad a(x) \cdot x(y) \cdot Q \xrightarrow{a} (x) \cdot x(y) \cdot Q}{((\nu b)\, \overline{a}b \cdot \overline{b}c) \mid a(x) \cdot x(y) \cdot Q \xrightarrow{\tau} (\nu b)\, (\overline{b}c \mid b(y) \cdot Q[x \leftarrow b])}$$

where GENCOMM is the following generalized communication rule:

$$\frac{P_1 \xrightarrow{a} P_1' \quad P_2 \xrightarrow{\overline{a}} P_2'}{P_1 \mid P_2 \xrightarrow{\tau} P_1' @ P_2'}$$

where e.g. $((\nu b)\, \overline{b}c) @ ((x) \cdot x(y) \cdot Q) = (\nu b)(\overline{b}c \mid b(y) \cdot Q[x \leftarrow b])$. Thanks to concretions and abstractions, there is not OPEN rule anymore, and there is only one (generalized) communication rule.

4.3 Sorting

We want to rule out

$$\overline{s}\langle x, w \rangle \mid w(v) \cdot \overline{v}\langle a, b, c \rangle \mid s(t, u) \cdot \overline{u}\langle t \rangle \cdot t(y, z) \cdot P$$

because this process reduces to $w(v) \cdot \overline{v}\langle a, b, c \rangle \mid \overline{w}\langle x \rangle \cdot x(y, z) \cdot P$ and then to $\overline{x}\langle a, b, c \rangle \mid x(y, z) \cdot P$, where there is an arity mismatch.

Proof of preservation of well-sortedness under reduction. It just resides on the remark that substituting names with others that have the same sort does not affect well-sortedness.

Let us prove that the process above is ruled out. We first notice that t and x (resp. w and u) should have the same sort. Now, since w occurs in $w(v)$ and $\overline{u}$ occurs in $\overline{u}\langle t \rangle$, we infer that v and t should have the same sort. But they occur with different arities.

4.4 From polyadic to monadic

We take this example to illustrate the notions of correct encoding, and of complete encoding. We first try the following encoding, which we decorate by a subscript $_?$ as an anticipation of its rejection. To simplify, we consider here that $x() \cdot P$ stands in both calculi for $x(z) \cdot P$ where z is not free in P, and we define the encoding only for $n \geq 1$, as follows.

$$\begin{aligned} [\![x(z_1, \ldots, z_n) \cdot P]\!]_? &= x(z_1) \cdot \ldots \cdot x(z_n) \cdot [\![P]\!]_? \\ [\![\overline{x}\langle w_1, \ldots, w_n\rangle \cdot P]\!]_? &= \overline{x}w_1 \cdot \ldots \cdot \overline{x}w_n \cdot [\![P]\!]_? \end{aligned}$$

Now consider

$$P = x(y_1, y_2) \cdot (y_1() \mid \overline{y_2}\langle\rangle) \mid x(y_3, y_4) \mid \overline{x}\langle w_1, w_2\rangle \mid \overline{x}\langle w_3, w_4\rangle$$

Applying the expansion law, we get that $[\![P]\!] \approx Q$, where Q has the form $C[w_i() \mid \overline{w_j}\langle\rangle]_{i,j}$ with $C[]$ a multihole context generated by the following syntax:

$$C ::= [\,]_{i,j} \mid \tau \cdot C \mid \Sigma_k C_k$$

Since Q does not contain subjects with arity ≥ 2, we have $Q = [\![Q]\!]$. Now, we take, say, $w_1 = w_3, w_1 \neq w_2, w_3 \neq w_4$, and we show that then $P \not\approx Q$ Indeed, Q can become 0 via $w_1() \mid \overline{w_3}\langle\rangle$ after some τ-actions, while it is easy to check that for every P' such that $P \rightarrow^* P'$ there exists P'' such that $P' \stackrel{\overline{w_2}\langle\rangle}{\Longrightarrow} P''$ or $P' \stackrel{\overline{w_4}\langle\rangle}{\Longrightarrow} P''$, and hence $P' \not\approx 0$. The point of the counter-example is that the encoding allows mixing the emitted values, contrarily to the discipline of the polyadic calculus.

We now sketch the proof that the following more careful encoding is correct:

$$\begin{aligned} [\![x(y_1, \ldots, y_n) \cdot P]\!] &= x(w) \cdot w(y_1) \cdot \ldots \cdot w(y_n) \cdot [\![P]\!] && (w \text{ not free in } P) \\ [\![\overline{x}\langle w_1, \ldots, w_n\rangle \cdot P]\!] &= (\nu w)\, \overline{x}w \cdot \overline{w}w_1 \cdot \ldots \cdot \overline{w}w_n \cdot [\![P]\!] && (w \text{ not free in } P) \end{aligned}$$

First, remark that this encoding takes also the nullary case into account, by replacing $x()$ by $x(w)$ where w is dummy. We show that $[\![P]\!] \approx [\![Q]\!]$ implies $P \approx Q$. We show that $\{(P, Q) \mid [\![P]\!] \approx [\![Q]\!]\}$ is a bisimulation. If $P \stackrel{xw_1 \ldots w_n}{\longrightarrow} P'$, then $[\![P]\!] \stackrel{xw\, wy_1 \ldots wy_n}{\longrightarrow} [\![P']\!]$, hence $[\![Q]\!] \stackrel{xw\, wy_1 \ldots wy_n}{\longrightarrow} R$ for some $R \approx [\![P']\!]$. It is here that we formalize the fact that unlike the encoding $[\![\,]\!]_?$, the encoding $[\![\,]\!]$ does not allow for an interference between different emissions or receptions. Hence, performing if needed some τ steps from R in order to complete the simulation of some τ-steps from Q, we can infer that there exists Q' such that $Q \stackrel{xw_1 \ldots w_n}{\Longrightarrow} Q'$ and $R \stackrel{\epsilon}{\Rightarrow} [\![Q']\!]$. More precisely, these τ-steps are all performed using the rule CLOSE (it is here that the privacy of the name w in the compilation is used), and it follows that $R \approx [\![Q']\!]$, hence $P' \approx Q'$, which completes our sketch of proof.

Finally, we show that the encoding is not complete. We have that $\overline{x}y \cdot \overline{x}y \approx^c \overline{x}y \mid \overline{x}y$. We have $[\![P]\!] \equiv (\nu w, w')\, \overline{x}w \cdot \overline{w}y \cdot \overline{x}w' \cdot \overline{w'}y$ and $[\![Q]\!] = (\nu w)\, \overline{x}w \cdot \overline{w}y \mid (\nu w')\, \overline{x}w' \cdot \overline{w'}y$. We now observe that $[\![Q]\!]$ can perform the sequence of actions $\overline{x}(w)\overline{x}(w')$, which $[\![P]\!]$ cannot perform. Hence $[\![P]\!] \not\approx [\![Q]\!]$. The point is that this sequence of actions results from an interaction with a process of the form $x(z) \cdot x(z') \cdot R$ which does not respect the protocol of the compilation.

4.5 Encoding data types

(Taken from [9][pp. 152-153].) We want to compile binary trees whose internal nodes and leaves are decorated with, say, numbers (that we suppose to have encoded elsewhere). Such trees can be declared by the following syntax (where e is an expression denoting a number):

$$t ::= \text{leaf } e \mid \text{node } e\, t\, t$$

or more briefly $t ::= e \mid e(t, t')$

$$\begin{array}{l} \text{LEAF}\langle x, v\rangle = !x(l, n) \cdot \overline{l}\langle v\rangle \\ \text{NODE}\langle x, v, y, z\rangle = !x(l, n) \cdot \overline{n}\langle v, y, z\rangle \end{array}$$

$$\begin{array}{l} [\![\text{leaf } e]\!]\langle x\rangle = (\nu v)\, (\text{LEAF}\langle x, v\rangle \mid [\![e]\!]\langle v\rangle) \\ [\![\text{node } e\, t\, t']\!]\langle x\rangle = (\nu v, y, z)\, (\text{NODE}\langle x, v, y, z\rangle \mid [\![e]\!]\langle v\rangle \mid [\![t]\!]\langle y\rangle \mid [\![t']\!]\langle z\rangle) \end{array}$$

For example, we get

$$\begin{array}{rcl} [\![2(1, 4(3, 5))]\!]\langle a\rangle & = & (\nu v_1, v_2, v_3, v_4, v_5, b, l_1, l_2, l_3) \\ & & \quad \text{NODE}\langle a, v_2, l_1, b\rangle \mid [\![2]\!]\langle v_2\rangle \\ & & \quad \mid \text{LEAF}\langle l_1, v_1\rangle \mid [\![1]\!]\langle v_1\rangle \\ & & \quad \mid \text{NODE}\langle b, v_4, l_2, l_3\rangle \mid \text{LEAF}\langle l_2, v_3\rangle \\ & & \quad \mid [\![3]\!]\langle v_3\rangle \mid \text{LEAF}\langle l_3, v_5\rangle \mid [\![5]\!]\langle v_5\rangle \end{array}$$

This amounts to giving a name to all the edges in the tree representation (including an edge going into the root):

$$[\![t]\!]\langle a\rangle = a \quad \text{node} \begin{cases} v_2 & \text{two} \\ l_1 & \text{leaf} \left\{ \begin{array}{ll} v_1 & \text{one} \end{array} \right. \\ b & \text{node} \begin{cases} v_4 & \text{four} \\ l_2 & \text{leaf} \left\{ \begin{array}{ll} v_3 & \text{three} \end{array} \right. \\ l_3 & \text{leaf} \left\{ \begin{array}{ll} v_5 & \text{five} \end{array} \right. \end{cases} \end{cases}$$

We now define a process which when interacting with a tree will produce its mirror image:

$$\mathrm{MIR}\langle a, a'\rangle = (\nu g, h)\, M\langle a, a', g, h\rangle \mid G\langle g\rangle \mid H\langle h\rangle$$

where

$$\begin{aligned} M\langle a, a', g, h\rangle = {} & (\nu l, n)\, \overline{a}\langle l, n\rangle \cdot (l(v) \cdot \overline{g}\langle a', v\rangle \\ & +n(v, y, z) \cdot (\nu y', z')\, \overline{h}\langle a', v, y', z'\rangle \cdot (M\langle z, y', g, h\rangle \mid M\langle y, z', g, h\rangle)) \end{aligned}$$

$$\begin{aligned} G\langle g\rangle &= !g(x, v) \cdot \mathrm{LEAF}\langle x, v\rangle \\ H\langle h\rangle &= !h(x, v, y, z) \cdot \mathrm{NODE}\langle x, v, y, z\rangle \end{aligned}$$

Here, G is a leaf generator, H is a node generator, while M is the main program that coordinates the tasks. The execution of $(\nu a)\,(\mathrm{MIR}\langle a, a'\rangle \mid [\![t]\!]\langle a\rangle)$ goes as follows. First $M\langle a, a', g, h\rangle$ synchronizes with $\mathrm{NODE}\langle a, v_2, l_1, b\rangle$ on a. Then there is a synchronization on n, and the corresponding branch of M is chosen. So, v_2, l_1, b are sent, and we now have (α-converting y', z' as b', l_1') $\overline{h}\langle a', v_2, b', l_1'\rangle \cdot (M\langle b, b', g, h\rangle \mid M\langle l_1, l_1', g, h\rangle)$. Then there is a synchronization with $H\langle h\rangle$ on h, which creates a node $\mathrm{NODE}\langle a', v_2, b', l_1'\rangle$, so we have started the construction of the mirror:

$$a' \quad \text{node} \begin{cases} v_2 & \text{two} \\ b' & \cdots \\ l_1' & \cdots \end{cases}$$

Continuing in this way, we get

$$(\nu a)\,(\mathrm{MIR}\langle a, a'\rangle \mid [\![2(1, 4(3, 5))]\!]\langle a\rangle) \to^* [\![2(4(5, 3), 1)]\!]\langle a'\rangle\ .$$

4.6 Encoding the λ-calculus

Call-by-value CPS transformation. The transformation is parameterized by a variable k (the *continuation*).

$$
\begin{aligned}
&[\![x]\!]_p = px\\
&[\![\lambda x.M]\!]_p = p(\lambda x, k.[\![M]\!]_k)\\
&[\![MN]\!]_p = (\lambda q.[\![M]\!]_q)(\lambda v.(\lambda r[\![N]\!]_r)(\lambda w.vwp))
\end{aligned}
$$

Up to a β-expansion, the target of this translation lies in a subset of the λ-calculus defined by the following syntax:

$$M ::= x\vec{y} \mid (\lambda x.M_1)(\lambda \vec{y}.M_2)$$

which is more readable with a let notation: $M ::= x\vec{y} \mid \mathit{let}\ x(\vec{y}) = M_2\ \mathit{in}\ M_1$. (This calculus has been first defined and studied by Thielecke [10].) With this notation, the above translation is easier to read too:

$$
\begin{aligned}
&[\![x]\!]_p = px\\
&[\![\lambda x.M]\!]_p = (\mathit{let}\ y(x,k) = [\![M]\!]_k\ \mathit{in}\ py)\\
&[\![MN]\!]_p = (\mathit{let}\ q(v) = (\mathit{let}\ r(w) = vwp\ \mathit{in}\ [\![N]\!]_r)\ \mathit{in}\ [\![M]\!]_q)
\end{aligned}
$$

What is this mapping from λ-calculus to λ-calculus for? It encodes within the λ-calculus the call-by-value evaluation strategy. For example, the compilation of MN reads off as: Evaluate M, pass its value V to the continuation q, which triggers the evaluation of N whose value W is passed to the continuation r, which triggers the evaluation of VW, with continuation p. In call-by-value λ-calculus, the values are the abstractions or the variables (which are bound to abstractions). Hence VW stands for a redex.
We are now left with compiling the restricted λ-calculus to the π-calculus:

$$
\begin{aligned}
&[\![x\vec{y}]\!] = \overline{x}\langle\vec{y}\rangle\\
&[\![\mathit{let}\ x(\vec{y}) = M_1\ \mathit{in}\ M_2]\!] = (\nu x)\ ([\![M_2]\!]\ |!x(\vec{y}) \cdot [\![M_1]\!])
\end{aligned}
$$

We can dispense with the ! if we know that x appears only once in M_1. By composing the two translations, we get:

$$
\begin{aligned}
&[\![x]\!]_p = \overline{p}\langle x\rangle\\
&[\![\lambda x.M]\!]_p = (\nu y)\ (\overline{p}\langle y\rangle\ |!y(x,k) \cdot [\![M]\!]_p)\\
&[\![MN]\!]_p = (\nu q)\ ([\![M]\!]_q \mid q(v) \cdot ((\nu r)\ [\![N]\!]_r \mid r(w) \cdot \overline{v}\langle w,p\rangle))
\end{aligned}
$$

In the translation, we have taken into account the fact that the parameters p, k, q, r in the CPS translation appear only once.

4.7 Asynchronous π-calculus

The asynchronous π-calculus is the sublanguage of the π-calculus in which all outputs are followed by 0. The intuition is that emission is decoupled from synchronization: in $\overline{a}x \cdot P$, the process P is blocked until a receiver on a accepts x, while in $\overline{a}x \mid P$ the process P can start to be active before x is is accepted.

There is no loss in expressive power, in the sense that there is a sound translation from π-calculus to asynchronous π-calculus, which is easy to formulate in the polyadic versions of both calculi:

$$\begin{array}{l}
[\![\overline{x}\langle\vec{y}\rangle \cdot P]\!] = (\nu a)\ (\overline{x}\langle\vec{y}, a\rangle \mid a() \cdot [\![P]\!]) \\
[\![x(\vec{z}) \cdot P]\!] = x(\vec{z}, a) \cdot (\overline{a}\langle\rangle \mid [\![Q]\!])
\end{array}$$

We see that

$$\begin{array}{rcl}
[\![x(z) \cdot Q \mid \overline{z}x \cdot P]\!] & \to & (\nu a)\ (\overline{a}\langle\rangle \mid [\![Q]\!][z \leftarrow x] \mid a() \cdot P) \\
& \to & (\nu a)\ ([\![Q]\!][z \leftarrow x] \mid P) \\
& \equiv & [\![Q[z \leftarrow x]]\!] \mid P
\end{array}$$

Note that P can become unguarded only after the reception of z.

4.8 Core join calculus

The join calculus was motivated by two aims: to merge functional programming with concurrency, and to have a more realistic computational model. The syntax of (core) join calculus is remarkably economical: replication, restriction and input are merged into a single construct:

$$P ::= x\langle\vec{u}\rangle \mid (P \mid P) \mid (\mathit{def}\ x(\vec{u}) \rhd P\ \mathit{in}\ P) \mid (\mathit{def}\ x(\vec{u}) \mid y(\vec{v}) \rhd P\ \mathit{in}\ P)$$

where the names is $\vec{u}$ and $\vec{v}$ are all distinct. (We invite the reader to compare this with the target of the CPS translation in section 4.6.) A correct encoding into the asymchronous π-calculus is given as follows:

$$\begin{array}{l}
[\![P \mid Q]\!] = [\![P]\!] \mid [\![Q]\!] \\
[\![x\langle\vec{v}\rangle]\!] = \overline{x}\langle\vec{v}\rangle \\
[\![\mathit{def}\ x(\vec{u}) \rhd P\ \mathit{in}\ Q]\!] = (\nu x)\ ((!x(\vec{u}) \cdot [\![P]\!]) \mid [\![Q]\!]) \\
[\![\mathit{def}\ x(\vec{u}) \mid y(\vec{v}) \rhd P\ \mathit{in}\ Q]\!] = (\nu x, y)\ ((!x(\vec{u}) \cdot y(\vec{v}) \cdot [\![P]\!]) \mid [\![Q]\!])
\end{array}$$

The operational semantics of the join calculus is best expressed as a chemical abstract machine. The states of the machine have the form $D \vdash M$ where D

is a multiset of definitions and M is a multiset of processes. One starts with $\vdash P$.

$$
\begin{array}{l}
(D \vdash P \mid Q, M) \leftrightharpoons (D \vdash P, Q, M) \\
(D \vdash \mathit{def}\ x(\vec{u}) \mid y(\vec{v}) \triangleright P\ \mathit{in}\ Q, M) \leftrightharpoons (D, (x(\vec{u}) \mid y(\vec{v}) \triangleright P) \vdash Q, M) \quad (x, y \text{ fresh}) \\
(D, (x(\vec{u}) \triangleright P) \vdash x\langle\vec{a}\rangle, M) \rightarrow (D, (x(\vec{u}) \triangleright P) \vdash P[\vec{u} \leftarrow \vec{a}], M) \\
(D, (x(\vec{u}) \mid y(\vec{v}) \triangleright P) \vdash (x\langle\vec{a}\rangle \mid y\langle\vec{b}\rangle), M) \rightarrow (D, (x(\vec{u}) \mid y(\vec{v}) \triangleright P) \vdash P[\vec{u}, \vec{v} \leftarrow \vec{a}, \vec{b}], M)
\end{array}
$$

Finally, here is a correct encoding of the asynchronous π-calculus without sums into the join calculus. We suppose that replications are used only in front of inputs. Each name x is mapped to two names x_i and x_o, dealing with input and output, respectively. We give the translation for monadic processes, for simplicity.

$$
\begin{array}{l}
[\![P \mid Q]\!] = [\![P]\!] \mid [\![Q]\!] \\
[\![(\nu x)P]\!] = \mathit{def}\ x_0(v_o, v_i) \mid x_i(\kappa) \triangleright \kappa\langle v_o, v_i\rangle\ \mathit{in}\ [\![P]\!] \\
[\![\overline{x}v]\!] = x_0\langle v_0, v_i\rangle \\
[\![x(v) \cdot P]\!] = \mathit{def}\ \kappa(v_o, v_i) \triangleright [\![P]\!]\ \mathit{in}\ x_i\langle\kappa\rangle \\
[\![!x(v) \cdot P]\!] = \mathit{def}\ \kappa(v_o, v_i) \triangleright (x_i\langle\kappa\rangle \mid [\![P]\!])\ \mathit{in}\ x_i\langle\kappa\rangle
\end{array}
$$

Hence an input is translated as the emission of a defined continuation, and a local name is implemented by joining the emission of values and of (the name of) a continuation that together trigger the continuation with these values.

4.9 Typing

$T ::= B \mid \#T$. The key typing rules are:

$$
\frac{\Gamma \vdash a : \#T \quad \Gamma, x : T \vdash P}{\Gamma \vdash a(x) \cdot P}
\qquad
\frac{\Gamma \vdash \overline{a} : \#T \quad \Gamma \vdash b : T \quad \Gamma \vdash P}{\Gamma \vdash \overline{a}b \cdot P}
$$

One may want to add constants, such a booleans or integers ($B ::= \mathit{Bool} \mid \mathit{Int}$, $\vdash n : \mathit{Int}$, etc...). If we have made such an addition, and if we separate the types into value types B and link types $L ::= \#B$ (hence no nesting of $\#$ allowed), now the typing rules above force x and b to be of value types, and the link a to be of link type. No mixture between links and values is possible, links cannot be sent. This is just CCS with value passing!

4.10 i/o types

$T ::= B \mid \#T \mid oT \mid iT$. The subtyping rules are:

$$\frac{}{\#T \leq iT} \quad \frac{}{\#T \leq oT} \quad \frac{S \leq T}{iS \leq iT} \quad \frac{S \leq T}{oT \leq oS}$$

The key typing rules are:

$$\frac{\Gamma \vdash a : iT \quad \Gamma, x : T \vdash P}{\Gamma \vdash a(x) \cdot P} \qquad \frac{\Gamma \vdash \overline{a} : oT \quad \Gamma \vdash b : T \quad \Gamma \vdash P}{\Gamma \vdash \overline{a}b \cdot P}$$

$$\frac{\Gamma \vdash P : S \quad S \leq T}{\Gamma \vdash P : T}$$

The last rule is called the *subsumption* rule. The *contravariance* (i.e., the property that it reverses the subtyping relation) of o is justified as follows. Let $a(x) \cdot P \mid \overline{a}n$. And suppose that $a : \#S$ and $T \leq S$. To type $a(x) \cdot P$, we can only use $a : \#S \leq iS$, hence the occurrences of x in P have type T. On the other hand, we can take profit of the subtyping $\#S \leq oS \leq oT$ to emit $n : T$ on $\overline{a}$. This will not affect subject reduction, as in order to type $P[x \leftarrow v]$ we only have to use subsumption at every place where n is replaced.

The *covariance* (or monotonicity) of i is justified as follows. Let $a(x) \cdot P \mid \overline{a}n$. And suppose that $a : \#S$ and $S \leq T$. To type $a(x) \cdot P$, we can take profit of the subtyping $\#S \leq iS \leq iT$ and allow the occurrences of x in P to have type T. To type $\overline{a}n$, we can only use $\#S \leq oS$, hence $n : S$. This will not affect subject reduction, as in order to type $P[x \leftarrow v]$ we only have to use subsumption at every place where n is replaced.

Let us see now why $\#$ has to be invariant, by exhibiting a concrete counter-example, with $\mathit{int} \leq \mathit{real}$. Consider $a(x) \cdot \overline{b}(\mathit{succ}\, z) \mid \overline{a}\, 3.5$. This process should not be typable, as it evolves to $\overline{b}(\mathit{succ}\, 3.5)$. But, if we accept the covariant rule for $\#$, then $\#\mathit{int} \leq \#\mathit{real}$ and $a : \#\mathit{int} \vdash a(x) \cdot \overline{b}(\mathit{succ}\, z) \mid \overline{a}\, 3.5$, while if we accept the contravariant rule then $\#\mathit{real} \leq \#\mathit{int}$ and $a : \#\mathit{real} \vdash a(x) \cdot \overline{b}(\mathit{succ}\, z) \mid \overline{a}\, 3.5$.

Example of the use of i/o types: the printer with mobile ownership. A printer is modelled as a process of the form $P = !p(j) \cdot P'$. It is initially accessible to, say, two clients, one of which has print jobs to deliver, which is formalized by $C_1 = \overline{p}j_1 \cdot \overline{p}j_2$, while the other wishes to communicate the private name p to other future clients: $C_2 = \overline{b}p$. But the intention is that other clients will not do harm to C_1 (other than possibly delaying its printing jobs). Consider a client $C_3 = b(p)\cdot !p(j)$. Then the following execution, whch throws away the jobs of C_1, is possible:

$$\begin{array}{rcl} ((\nu p)\,(P \mid C_1 \mid C_2)) \mid C_3 & \equiv & (\nu p)\,(P \mid C_1 \mid C_2 \mid C_3) \\ & \rightarrow & (\nu p)\,(P \mid \overline{p}j_1 \cdot \overline{p}j_2 \mid !p(j)) \\ & \rightarrow^* & (\nu p)\,(P \mid !p(j)) \end{array}$$

This example can be prevented by imposing that the initial clients can only export the output capability of p. For example, we can impose the type $\#oT$ to b, where p has type $\#T$ in the typing context for $P \mid C_1 \mid C_2$.

References

[1] G. Berry and G. Boudol, The chemical abstract machine, Theoretical Computer Science 96, 217-248 (1992)).

[2] C. Fournet and G. Gonthier, The reflexive chemical abstract machine and the join calculus, Proc. Principles of Programming Languages '96.

[3] C. Fournet and G. Gonthier, The join calculus: a language for distributed mobile programming, to appear in the Proceedings of APPSEM Summer School, Caminha, September 2000.

[4] R. Milner, A calculus of communicating systems, Springer Lecture Notes in Computer Science 92 (1980).

[5] R. Milner, Communication and concurrency, Prentice Hall (1989).

[6] R. Milner, Communicating and mobile systems: the π-calculus, Cambridge University Press (1999).

[7] D. Sangiorgi, An introduction to the semantics of concurrent languages and systems, cours de DEA Programmation (Nov. 2000).

[8] D. Sangiorgi, An introduction to process calculi, slides for the BASICS'01 Summer School, Beijing (Oct. 2001).

[9] D. Sangiorgi and D. Walker, Communication and mobility: the π-calculus, Cambridge University Press (2001).

[10] Thielecke, H., Categorical structure of continuation passing style, PhD Thesis, Univ. of Edinburgh (1997).