Dipartimento di Informatica, Università di Pisa

Notes on

Models of Computation

Introduction, Preliminaries, Operational Semantics of IMP, Induction, Recursion, Partial Orders, Denotational Semantics of IMP, Operational Semantics of HOFL, Domain Theory, Denotational Semantics of HOFL, CCS, Temporal Logic, μ-calculus, π-calculus Markov Chains with Actions and Nondeterminism, PEPA

Roberto Bruni Lorenzo Galeotti Ugo Montanari*

May 16, 2015

*Also based on material by Andrea Cimino, Lorenzo Muti, Gianmarco Saba, Marco Stronati

Contents

7. Domain Theory

As done for IMP we would like to introduce the denotational semantics of HOFL, for which we need to develop a proper domain theory.

In order to define the denotational semantics of IMP we have shown that the semantic domain of commands, for which we need to apply fixpoint theorem, has the required properties. The situation is more complicated for HOFL, because HOFL provides constructors for infinitely many term types, so there are infinitely many domains to be considered. We will handle this problem by showing by structural induction that the type constructors of HOFL correspond to domains which are equipped with adequate $CPO_\perp$ structures.

7.1. The Domain $\mathbb{N}_\perp$

We define the $\text{CPO}_\perp$ $\mathbb{N}_\perp = (\mathbb{N} \cup \{\perp_{\mathbb{N}_\perp}\}, \sqsubseteq)$ as follows:

- $\mathbb{N}$ is the set of integer numbers
- $\forall x \in \mathbb{N} \cup \{\perp_{\mathbb{N}_\perp}\}.\ \perp_{\mathbb{N}_\perp} \sqsubseteq x$ and $x \sqsubseteq x$

obviously $\mathbb{N}_\perp$ is a CPO with bottom, indeed $\perp_{\mathbb{N}_\perp}$ is the bottom element and each chain has a LUB (note that chains are all of length 1 or 2).

7.2. Cartesian Product of Two Domains

We start with two $CPO_\perp$:

$$\begin{aligned} \mathscr{D} &= (D, \sqsubseteq_D) \\ \mathscr{E} &= (E, \sqsubseteq_E) \end{aligned}$$

Now we construct the Cartesian product $\mathscr{D} \times \mathscr{E} = (D \times E, \sqsubseteq_{D\times E})$ which has as elements the pairs of elements of D and E. Let us define the order as follows (note that the order is different from the lexicographic one):

- $\forall d_0, d_1 \in D\ \forall e_0, e_1 \in E.\ (d_0, e_0) \sqsubseteq_{D\times E} (d_1, e_1) \Leftrightarrow d_0 \sqsubseteq_D d_1 \wedge e_0 \sqsubseteq_E e_1$

Let us show that $\sqsubseteq_{D\times E}$ is a partial order:

- reflexivity: since $\sqsubseteq_D$ and $\sqsubseteq_E$ are reflexive we have $\forall e \in E\ e \sqsubseteq_E e$ and $\forall d \in D\ d \sqsubseteq_D d$ so by definition of $\sqsubseteq_{D\times E}$ we have $\forall d \in D\ \forall e \in E.\ (d, e) \sqsubseteq_{D\times E} (d, e)$.
- antisymmetry: let us assume $(d_0, e_0) \sqsubseteq_{D\times E} (d_1, e_1)$ and $(d_1, e_1) \sqsubseteq_{D\times E} (d_0, e_0)$ so by definition of $\sqsubseteq_{D\times E}$ we have $d_0 \sqsubseteq_D d_1$ (using the first relation) and $d_1 \sqsubseteq_D d_0$ (by using the second relation) so it must be $d_0 = d_1$ and similarly $e_0 = e_1$, hence $(d_0, e_0) = (d_1, e_1)$.
- transitivity: let us assume $(d_0, e_0) \sqsubseteq_{D\times E} (d_1, e_1)$ and $(d_1, e_1) \sqsubseteq_{D\times E} (d_2, e_2)$. By definition of $\sqsubseteq_{D\times E}$ we have $d_0 \sqsubseteq_D d_1$, $d_1 \sqsubseteq_D d_2$, $e_0 \sqsubseteq_E e_1$ and $e_1 \sqsubseteq_E e_2$. By transitivity of $\sqsubseteq_D$ and $\sqsubseteq_E$ we have $d_0 \sqsubseteq_D d_2$ and $e_0 \sqsubseteq_E e_2$. By definition of $\sqsubseteq_{D\times E}$ we obtain $(d_0, e_0) \sqsubseteq_{D\times E} (d_2, e_2)$.

Now we show that the PO has a bottom element $\perp_{D\times E} = (\perp_D, \perp_E)$. In fact $\forall d \in D, e \in E.\ \perp_D \sqsubseteq d \wedge \perp_E \sqsubseteq e$, thus $(\perp_D, \perp_E) \sqsubseteq_{D\times E} (d, e)$. It remains to show the completeness of $\mathscr{D} \times \mathscr{E}$.

Theorem 7.1 (Completeness of $\mathscr{D} \times \mathscr{E}$)
The PO $\mathscr{D} \times \mathscr{E}$ defined above is complete.

Proof. We will prove that for each chain $(d_i, e_i)_{i\in\omega}$ it holds:

$$\bigsqcup_{i\in\omega}(d_i, e_i) = \left(\bigsqcup_{i\in\omega} d_i, \bigsqcup_{i\in\omega} e_i\right)$$

Obviously $(\bigsqcup_{i\in\omega} d_i, \bigsqcup_{i\in\omega} e_i)$ is an upper bound, indeed for each $j \in \omega$ we have $d_j \sqsubseteq_D \bigsqcup_{i\in\omega} d_i$ and $e_j \sqsubseteq_E \bigsqcup_{i\in\omega} e_i$ so by definition of $\sqsubseteq_{D\times E}$ it holds $(d_j, e_j) \sqsubseteq_{D\times E} (\bigsqcup_{i\in\omega} d_i, \bigsqcup_{i\in\omega} e_i)$.
Moreover $(\bigsqcup_{i\in\omega} d_i, \bigsqcup_{i\in\omega} e_i)$ is also the least upper bound. Indeed, let $(\bar{d}, \bar{e})$ be an upper bound of $\{(d_i, e_i)\}_{i\in\omega}$, since $\bigsqcup_{i\in\omega} d_i$ is the LUB of $\{d_i\}_{i\in\omega}$ we have $\bigsqcup_{i\in\omega} d_i \sqsubseteq_D \bar{d}$, furthermore we have that $\bigsqcup_{i\in\omega} e_i$ is the LUB of $\{e_i\}_{i\in\omega}$ then $\bigsqcup_{i\in\omega} e_i \sqsubseteq_E \bar{e}$. So by definition of $\sqsubseteq_{D\times E}$ we have $(\bigsqcup_{i\in\omega} d_i, \bigsqcup_{i\in\omega} e_i) \sqsubseteq_{D\times E} (\bar{d}, \bar{e})$. Thus $(\bigsqcup_{i\in\omega} d_i, \bigsqcup_{i\in\omega} e_i)$ is the least upper bound. □

Let us define the projection operators of $\mathscr{D} \times \mathscr{E}$.

Definition 7.2 (Projection operators π_1 and π_2)
Let $(d, e) \in D \times E$ be a pair, we define the left and right projection functions $\pi_1 : D \times E \to D$ and $\pi_2 : D \times E \to E$ as follows.

- $\pi_1((d, e)) = d$
- $\pi_2((d, e)) = e$

Recall that in order to use a function in domain theory we have to show that it is continuous, this ensures that the function respects the domain structure (i.e. the function does not change the order and preserves limits) and so we can calculate its fixpoints.
So we have to prove that each function which we use on $\mathscr{D} \times \mathscr{E}$ is continuous. The proof that projections are monotonic is immediate and left as an exercise.

Theorem 7.3 (Continuity of π_1 and π_2)
Let π_1 and π_2 be the projection functions of the previous definition and let $\{(d_i, e_i)\}_{i\in\omega}$ be a chain of elements in $\mathscr{D} \times \mathscr{E}$, then:

$$\pi_1\left(\bigsqcup_{i\in\omega}(d_i, e_i)\right) = \bigsqcup_{i\in\omega} \pi_1((d_i, e_i))$$

and

$$\pi_2\left(\bigsqcup_{i\in\omega}(d_i, e_i)\right) = \bigsqcup_{i\in\omega} \pi_2((d_i, e_i))$$

Proof. Let us prove the first statement:

$$\pi_1\left(\bigsqcup_{i\in\omega}(d_i, e_i)\right) = \pi_1\left(\left(\bigsqcup_{i\in\omega} d_i, \bigsqcup_{i\in\omega} e_i\right)\right) = \bigsqcup_{i\in\omega} d_i = \bigsqcup_{i\in\omega} \pi_1((d_i, e_i)).$$

For π_2 the proof is analogous. □

7.3. Functional Domains

As for Cartesian product, we start from two domains and we define the order on the set $[D \to E]$ of the continuous functions in $\{f \mid f : D \to E\}$. Note that as usual we require the continuity of the functions to

preserve the applicability of fixpoint theory.
Let us consider the $\text{CPO}_\perp$s:

$$\begin{aligned}\mathscr{D} &= (D, \sqsubseteq_D)\\ \mathscr{E} &= (E, \sqsubseteq_E)\end{aligned}$$

Now we define an order on set of continuous functions $[D \to E]$:

$$[\mathscr{D} \to \mathscr{E}] = ([D \to E], \sqsubseteq_{[D \to E]})$$

where:

- $[D \to E] = \{\, f \mid f : D \to E, f \text{ is continuous} \,\}$
- $f \sqsubseteq_{[D \to E]} g \Leftrightarrow \forall d \in D. f(d) \sqsubseteq_E g(d)$

We leave as an exercise the proof that $[\mathscr{D} \to \mathscr{E}]$ is a PO with bottom, namely the order is reflexive, anti-symmetric, transitive and that the function $\perp_{[D \to E]}$ defined by letting $\perp_{[D \to E]}(d) = \perp_E$ for any $d \in D$ is a continuous function and also the bottom element of $[\mathscr{D} \to \mathscr{E}]$.
Let us show that the PO is complete. In order to simplify the completeness proof we introduce the following lemmas:

Lemma 7.4 (Switch Lemma)
Let $(E, \sqsubseteq_E)$ be a CPO whose elements are of the form $e_{n,m}$ with $n, m \in \omega$. If $\sqsubseteq_E$ is such that:

$$e_{n,m} \sqsubseteq_E e_{n',m'} \text{ if } n \leq n' \text{ and } m \leq m'$$

then it holds:

$$\bigsqcup_{n,m \in \omega} e_{n,m} = \bigsqcup_{n \in \omega}(\bigsqcup_{m \in \omega} e_{n,m}) = \bigsqcup_{m \in \omega}(\bigsqcup_{n \in \omega} e_{n,m}) = \bigsqcup_{k \in \omega} e_{k,k}$$

Proof. Our order can be summarized as follows:

$$\begin{array}{ccccccccc}
\vdots & & \vdots & & \vdots & & \iddots & & \vdots \\
\sqcup & & \sqcup & & \sqcup & & & & \sqcup \\
e_{20} & \sqsubseteq & e_{21} & \sqsubseteq & e_{22} & \sqsubseteq & \cdots & \bigsqcup_{i \in \omega} e_{2i} & = e_2 \\
\sqcup & & \sqcup & & \sqcup & & & \sqcup & \\
e_{10} & \sqsubseteq & e_{11} & \sqsubseteq & e_{12} & \sqsubseteq & \cdots & \bigsqcup_{i \in \omega} e_{1i} & = e_1 \\
\sqcup & & \sqcup & & \sqcup & & & \sqcup & \\
e_{00} & \sqsubseteq & e_{01} & \sqsubseteq & e_{02} & \sqsubseteq & \cdots & \bigsqcup_{i \in \omega} e_{0i} & = e_0
\end{array}$$

We show that all the following sets have the same upper bounds:

$$\{e_{n,m}\}_{n,m \in \omega} \qquad \{\bigsqcup_{m \in \omega} e_{n,m}\}_{n \in \omega} \qquad \{\bigsqcup_{n \in \omega} e_{n,m}\}_{m \in \omega} \qquad \{e_{k,k}\}_{k \in \omega}$$

Let us consider the first two sets. Let e be an upper bound of $\{\bigsqcup_{m \in \omega} e_{n,m}\}_{n \in \omega}$ and take any $e_{n',m'}$ for some n', m'. Then

$$e_{n',m'} \sqsubseteq \bigsqcup_{m \in \omega} e_{n',m} \sqsubseteq e$$

Thus e is an upper bound for $\{e_{n,m}\}_{n,m \in \omega}$.

Vice versa, let e be an upper bound of $\{e_{n,m}\}_{n,m \in \omega}$ and consider $\bigsqcup_{m \in \omega} e_{n',m}$ for some n'. Since $\{e_{n',m}\}_{m \in \omega} \subseteq \{e_{n,m}\}_{n,m \in \omega}$, obviously e is an upper bound for $\{e_{n',m}\}_{m \in \omega}$ and therefore $\bigsqcup_{m \in \omega} e_{n',m} \sqsubseteq e$.

The correspondence between the sets of upper bounds of $\{e_{n,m}\}_{n,m \in \omega}$ and $\{\bigsqcup_{n \in \omega} e_{n,m}\}_{m \in \omega}$ can be proved analogously.

Now each element $e_{n,m}$ is smaller than $e_{k,k}$ with $k = max\{n, m\}$ thus an upper bound of $\{e_{k,k}\}_{k \in \omega}$ is also an upper bound of $\{e_{n,m}\}_{n,m \in \omega}$. Moreover $\{e_{k,k}\}_{k \in \omega}$ is a subset of $\{e_{n,m}\}_{n,m \in \omega}$ so an upper bounds of $\{e_{n,m}\}_{n,m \in \omega}$ is also an upper bound of $\{e_{k,k}\}_{k \in \omega}$.

The set of upper bounds $\{\bigsqcup_{m\in\omega} e_{n,m}\}_{n\in\omega}$ has a least element. In fact, $n_1 \leq n_2$ implies $\bigsqcup_{m\in\omega} e_{n_1,m} \sqsubseteq \bigsqcup_{m\in\omega} e_{n_2,m}$, since every upper bound of $\bigsqcup_{m\in\omega} e_{n_2,m}$ is an upper bound of $\bigsqcup_{m\in\omega} e_{n_1,m}$. Therefore $\{\bigsqcup_{m\in\omega} e_{n,m}\}_{n\in\omega}$ is a chain, and thus it has a LUB since E is a CPO. □

Lemma 7.5
Let $\{f_n\}_{n\in\omega}$ be a chain of functions (not necessarily continuous) in $\mathscr{D} \to \mathscr{E}$ the LUB $\bigsqcup_{n\in\omega} f_n$ exists and is defined as:

$$\left(\bigsqcup_{n\in\omega} f_n\right)(d) = \bigsqcup_{n\in\omega}(f_n(d))$$

Proof. Function $\lambda d.\bigsqcup_{n\in\omega}(f_n(d))$ is clearly an upper bound for $\{f_n\}_{n\in\omega}$ since for every k and d we have $f_k(d) \sqsubseteq_E \bigsqcup_{n\in\omega} f_n(d)$. Function $\lambda d.\bigsqcup_{n\in\omega}(f_n(d))$ is also the LUB of $\{f_n\}_{n\in\omega}$ since taken g such that $f_n \sqsubseteq_{D\to E} g$ for any n, we have for any d that $f_n(d) \sqsubseteq_E g(d)$ and therefore $\bigsqcup_{n\in\omega}(f_n(d)) \sqsubseteq_E g(d)$. □

Lemma 7.6
Let $\{f_n\}_{n\in\omega}$ be a chain of functions in $[\mathscr{D} \to \mathscr{E}]$ and let $\{d_n\}_{n\in\omega}$ be a chain on $\mathscr{D}$ then the function

$$h \stackrel{\text{def}}{=} \lambda d.\bigsqcup_{n\in\omega}(f_n(d))$$

is continuous, namely

$$h(\bigsqcup_{m\in\omega} d_m) = \bigsqcup_{n\in\omega}\left(f_n(\bigsqcup_{m\in\omega} d_m)\right) = \bigsqcup_{m\in\omega}\left(\bigsqcup_{n\in\omega} f_n(d_m)\right) = \bigsqcup_{m\in\omega} h(d_m)$$

Furthermore, $\lambda d.\bigsqcup_{n\in\omega}(f_n(d))$ is the LUB of $\{f_n\}_{n\in\omega}$ not only in $\mathscr{D} \to \mathscr{E}$ as stated by lemma 7.5, but also in $[\mathscr{D} \to \mathscr{E}]$.

Proof.

$$\begin{aligned}
\bigsqcup_{n\in\omega}\left(f_n\left(\bigsqcup_{m\in\omega} d_m\right)\right) &= \bigsqcup_{n\in\omega}\left(\bigsqcup_{m\in\omega}(f_n(d_m))\right) && \text{by continuity}\\
&= \bigsqcup_{m\in\omega}\left(\bigsqcup_{n\in\omega}(f_n(d_m))\right) && \text{by lemma 7.4 (switch lemma)}
\end{aligned}$$

The upper bounds of $\{f_n\}_{n\in\omega}$ in $\mathscr{D} \to \mathscr{E}$ are a larger set then those in $[\mathscr{D} \to \mathscr{E}]$, thus if $\lambda d.\bigsqcup_{n\in\omega}(f_n(d))$ is the LUB in $\mathscr{D} \to \mathscr{E}$, it is also the LUB in $[\mathscr{D} \to \mathscr{E}]$. □

Theorem 7.7 (Completeness of the functional space)
The PO $[\mathscr{D} \to \mathscr{E}]$ is a $CPO_\perp$

Proof. The statement follows immediately from the previous lemmas. □

7.4. Lifting

In IMP we introduced a lifting operator (Chapter 5.2.3) on memories Σ to obtain a CPO $\Sigma_\perp$. In the semantics of HOFL we need the same operator in a more general fashion: we need to apply the operator to any domain.

Definition 7.8
Let $\mathscr{D} = (D, \sqsubseteq_D)$ be a CPO and let $\perp$ be an element not in D, so we define the lifted $CPO_\perp$ $\mathscr{D}_\perp$ as follows:

- $D_\perp = \{(0, \perp)\} \cup \{1\} \times D$
- $\perp_{D_\perp} = (0, \perp)$
- $d_1 \sqsubseteq_D d_2 \Rightarrow (1, d_1) \sqsubseteq_{D_\perp} (1, d_2)$
- $\perp_{D_\perp} \sqsubseteq_{D_\perp} \perp_{D_\perp}$ *and* $\forall d \in D\ \perp_{D_\perp} \sqsubseteq_{D_\perp} (1, d)$

Now we define a lifting function $\lfloor - \rfloor : D \longrightarrow D_\perp$ as follows:

- $\lfloor d \rfloor = (1, d)\ \forall d \in D$

We leave it as an exercise to show that $\mathscr{D}_\perp$ is a $\mathrm{CPO}_\perp$.

As it was the case for Σ in the IMP semantics, when we add a bottom element to a domain $\mathscr{D}$ we would like to extend the continuous functions in $[D \to E]$ to continuous functions in $[D_\perp \to E]$. The function defining the extension should itself be continuous.

Definition 7.9
Let $\mathscr{D}$ be a CPO and let $\mathscr{E}$ be a $CPO_\perp$. We define a lifting operator $\_^* : [D \to E] \to [D_\perp \to E]$ for functions in $[D \to E]$ as follows:

$$\forall f \in [D \to E]\ \ f^*(x) = \begin{cases} \perp_E & \text{if } x = \perp_{D_\perp} \\ f(d) & \text{if } x = \lfloor d \rfloor \end{cases}$$

Theorem 7.10
i) If f is continuous in $[D \to E]$, then f^* is continuous in $[D_\perp \to E]$.

ii) The operator $\_^*$ is continuous.

Proof.

i) Let $\{d_i\}_{i \in \omega}$ be a chain in $\mathscr{D}_\perp$. We have to prove $f^*(\bigsqcup_{n\in\omega} d_n) = \bigsqcup_{n\in\omega} f^*(d_n)$.

If $\forall n.\ d_n = \perp_{D_\perp}$, then this is obvious.

Otherwise for some k and for all $m \geq k$ we have $d_m = \lfloor d'_m \rfloor$ and also $\bigsqcup_{n\in\omega} d_n = \lfloor \bigsqcup_{n\in\omega} d'_{n+k} \rfloor$. Then:

$$\begin{aligned}
f^*(\bigsqcup_{n\in\omega} d_n) &= f^*(\lfloor \bigsqcup_{n\in\omega} d'_{n+k} \rfloor) && \text{by the above assumption} \\
&= f(\bigsqcup_{n\in\omega} d'_{n+k}) && \text{by definition of lifting} \\
&= \bigsqcup_{n\in\omega} f(d'_{n+k}) && \text{by continuity of } f \\
&= \bigsqcup_{n\in\omega} f^*(d_{n+k}) && \text{by definition of lifting} \\
&= \bigsqcup_{n\in\omega} f^*(d_n) && \text{by prefix independence of the limit}
\end{aligned}$$

ii) We leave the proof that $\_^*$ is monotone as an exercise.

Let $\{f_i\}_{i\in\omega}$ be a chain of functions in $[\mathscr{D} \to \mathscr{E}]$. We will prove that for all $x \in D_\perp$:

$$\left(\bigsqcup_{i\in\omega} f_i\right)^* (x) = \left(\bigsqcup_{i\in\omega} f_i^*\right)(x)$$

if $x = \perp_{D_\perp}$ both sides of the equation simplify to $\perp_E$. So let us assume $x = \lfloor d \rfloor$ for some $d \in D$ we have:

$$\begin{aligned}
(\bigsqcup_{i\in\omega} f_i)^*(\lfloor d \rfloor) &= (\bigsqcup_{i\in\omega} f_i)(d) && \text{by definition of lifting}\\
&= \bigsqcup_{i\in\omega}(f_i(d)) && \text{by definition of LUB in a functional domain}\\
&= \bigsqcup_{i\in\omega}(f_i^*(\lfloor d \rfloor)) && \text{by definition of lifting}\\
&= (\bigsqcup_{i\in\omega} f_i^*)(\lfloor d \rfloor) && \text{by definition of LUB in a functional domain}
\end{aligned}$$

□

7.5. Function's Continuity Theorems

In this section we show some theorems which allow to prove the continuity of the functions which we will define over our CPOs. We start proving that the composition of two continuous functions is continuous.

Theorem 7.11
Let $f : [D \to E]$ and $g : [E \to F]$ be two continuous functions on $CPO_\perp$'s. The composition

$$f; g = g \circ f = \lambda d.\, g(f(d)) : D \to F$$

is a continuous function, namely for any chain $\{d_i\}_{i\in\omega}$ in D we have

$$g(f(\bigsqcup_{n\in\omega} d_n)) = \bigsqcup_{n\in\omega} g(f(d_n))$$

Proof. Immediate:

$$\begin{aligned}
g(f(\bigsqcup_{n\in\omega} d_n)) &= g(\bigsqcup_{n\in\omega} f(d_n)) && \text{by the continuity of } f\\
&= \bigsqcup_{n\in\omega} g(f(d_n)) && \text{by the continuity of } g
\end{aligned}$$

□

Now we consider a function whose outcome is a pair of values. So the function has as domain a CPO but the result is on a product of CPOs.

$$f : S \to D \times E$$

For this type of functions we introduce a theorem which allows to prove the continuity of f in a convenient way. We will consider f as the pairing of two simpler functions $g_1 : S \longrightarrow D$ and $g_2 : S \longrightarrow E$, then we can prove the continuity of f from the continuity of g_1 and g_2.

Theorem 7.12
Let $f : S \longrightarrow D \times E$ be a function over CPOs and let $g_1 : S \longrightarrow D$ and $g_2 : S \longrightarrow E$ be two functions defined as follows:

- $g_1 = f; \pi_1$

- $g_2 = f;\pi_2$

where $f;\pi_1 = \lambda s.\pi_1(f(s))$ is the composition of f and π_1. Notice that we have

$$\forall s \in S.\ f(s) = (g_1(s), g_2(s))$$

Then we have: f is continuous if and only if g_1 and g_2 are continuous.

Proof.

$\Rightarrow$) Immediate by Theorem 7.11 (continuity of composition) and Theorem7.3 (continuity of projections), since g_1 and g_2 are compositions of continuous functions.

$\Leftarrow$) We assume the continuity of g_1 and g_2. We prove:

$$f(\bigsqcup_{i\in\omega} s_i) = \bigsqcup_{i\in\omega} f(s_i)$$

So we have:

$$\begin{aligned}
f(\bigsqcup_{i\in\omega} s_i) &= (g_1(\bigsqcup_{i\in\omega} s_i), g_2(\bigsqcup_{i\in\omega} s_i)) && \text{by definition}\\
&= (\bigsqcup_{i\in\omega} g_1(s_i), \bigsqcup_{i\in\omega} g_2(s_i)) && \text{by continuity of } g_1 \text{ and } g_2\\
&= \bigsqcup_{i\in\omega}(g_1(s_i), g_2(s_i)) && \text{definition of LUB of pairs}\\
&= \bigsqcup_{i\in\omega} f(s_i) && \text{by definition}
\end{aligned}$$

□

Note that in our construction we defined only ordered pairs of elements, this means that if we want to consider sequences (i.e. with finitely many elements) we have to use the pairing repeatedly. So for example (a, b, c) is defined as $((a, b), c)$.

Now let us consider the case of a function $f : D_1 \times D_2 \longrightarrow E$ over CPOs which takes two arguments and returns an element of E. The following theorem allows us to study the continuity of f by analysing each parameter separately.

Theorem 7.13
Let $f : D_1 \times D_2 \to E$ be a function over CPOs. Then f is continuous iff all the functions in the following two classes are continuous.

$$\forall d_1 \in D_1.\ f_{d_1} : D_2 \to E \text{ defined as } f_{d_1} \stackrel{\text{def}}{=} \lambda d.f(d_1, d)$$

$$\forall d_2 \in D_2.\ f_{d_2} : D_1 \to E \text{ defined as } f_{d_2} \stackrel{\text{def}}{=} \lambda d.f(d, d_2)$$

Proof.

$\Rightarrow$) If f is continuous then $\forall d_1, d_2.\ f_{d_1}$ and f_{d_2} are continuous, since we are considering only certain chains (where one element of the pair is fixed). For example, fix d_1 and consider a chain $\{d_i\}_{i\in\omega}$ in D_2. Then we prove that f_{d_1} is continuous as follows:

$$\begin{aligned}
f_{d_1}(\bigsqcup_{i\in\omega} d_i) &= f(d_1, \bigsqcup_{i\in\omega} d_i) && \text{by definition of } f_{d_1}\\
&= f(\bigsqcup_{i\in\omega}(d_1, d_i)) && \text{by definition of LUB}\\
&= \bigsqcup_{i\in\omega} f(d_1, d_i) && \text{by continuity of } f\\
&= \bigsqcup_{i\in\omega} f_{d_1}(d_i) && \text{by definition of } f_{d_1}
\end{aligned}$$

⇐) On the other hand we have:

$$\begin{aligned}
f(\bigsqcup_{k\in\omega}(x_k, y_k)) &= f(\bigsqcup_{i\in\omega} x_i, \bigsqcup_{j\in\omega} y_j) && \text{by definition of LUB on pairs}\\
&= f_{\bigsqcup_{j\in\omega} y_j}(\bigsqcup_{i\in\omega} x_i) && \text{by definition of } f_{\bigsqcup_{j\in\omega} y_j}\\
&= \bigsqcup_{i\in\omega} f_{\bigsqcup_{j\in\omega} y_j}(x_i) && \text{by continuity of } f_{\bigsqcup_{j\in\omega} y_j}\\
&= \bigsqcup_{i\in\omega} f(x_i, \bigsqcup_{j\in\omega} y_j) && \text{by definition of } f_{\bigsqcup_{j\in\omega} y_j}\\
&= \bigsqcup_{i\in\omega} f_{x_i}(\bigsqcup_{j\in\omega} y_j) && \text{by definition of } f_{x_i}\\
&= \bigsqcup_{i\in\omega}\bigsqcup_{j\in\omega} f_{x_i}(y_j) && \text{by continuity of } f_{x_i}\\
&= \bigsqcup_{i\in\omega}\bigsqcup_{j\in\omega} f(x_i, y_j) && \text{by definition of } f_{x_i}\\
&= \bigsqcup_{k\in\omega} f(x_k, y_k) && \text{by Lemma 7.4 (switch lemma)}
\end{aligned}$$

□

7.6. Useful Functions

As done for IMP we will use the λ-notation as meta-language for the denotational semantics of HOFL. In previous sections we already defined two new functions for our meta-language: π_1 and π_2. We also showed that π_1 and π_2 are continuous. In this section we introduce some functions that will form the kernel of our meta-language.

Definition 7.14 (Apply)
We define a function apply : $[D \to E] \times D \to E$ *over domains as follows:*

$$\text{apply}(f, d) \stackrel{\text{def}}{=} f(d)$$

The function apply represents the application of a function in our meta-language. We leave it as an exercise to prove that apply is monotone. In order to use apply we prove that it is continuous.

Theorem 7.15 (Continuity of apply)
Let apply : $[D \to E] \times D \to E$ *be the function defined above and let* $\{(f_i, d_i)\}_{i\in\omega}$ *be a chain on the CPO*$_\perp$ $[\mathscr{D} \to \mathscr{E}] \times \mathscr{D}$ *then:*

$$\text{apply}(\bigsqcup_{i\in\omega}(f_i, d_i)) = \bigsqcup_{i\in\omega} \text{apply}(f_i, d_i)$$

Proof. By using the Theorem 7.13 we can test the continuity on each parameter separately.
Let us fix $d \in D$, we have:

$$\begin{aligned}
\text{apply}\left(\bigsqcup_{n\in\omega} f_n, d\right) &= \left(\bigsqcup_{n\in\omega} f_n\right)(d) && \text{by definition}\\
&= \bigsqcup_{n\in\omega}(f_n(d)) && \text{by definition of LUB of functions}\\
&= \bigsqcup_{n\in\omega} \text{apply}(f_n, d) && \text{by definition}
\end{aligned}$$

Now we fix $f \in [D \to E]$:

$$\begin{aligned}
\text{apply}\left(f, \bigsqcup_{m\in\omega} d_m\right) &= f\left(\bigsqcup_{m\in\omega} d_m\right) && \text{by definition}\\
&= \bigsqcup_{m\in\omega} f(d_m) && \text{by continuity of } f\\
&= \bigsqcup_{m\in\omega} \text{apply}(f, d_m) && \text{by definition}
\end{aligned}$$

So apply is a continuous function. □

Definition 7.16 (Curry and un-curry)
We define the function curry : $(D \times F \to E) \to (D \to (F \to E))$ *as:*

$$\text{curry } g\ d\ f \stackrel{\text{def}}{=} g(d, f) \text{ where } g : D \times F \to E, d : D \text{ and } f : F$$

And we define the un-curry : $(D \to F \to E) \to (D \times F \to E)$ *as:*

$$\text{un-curry } g\ (d, f) \stackrel{\text{def}}{=} g\ d\ f \text{ where } g : D \to F \to E, d : D \text{ and } f : F$$

Theorem 7.17 (Continuity of the curry of a function)
Let curry : $(D \times F \to E) \to (D \to (F \to E))$ *be the function defined above let* $\{d_i\}_{i\in\omega}$ *be a chain on* $\mathscr{D}$ *and let* $g : D \times F \to E$ *a continuous function then* (curry g) *is a continuous function, namely*

$$(\text{curry } g)(\bigsqcup_{i\in\omega} d_i) = \bigsqcup_{i\in\omega} (\text{curry } g)(d_i)$$

Proof. Let us note that since g is continuous, by Theorem 7.13 g is continuous separately on each parameter. Then let us take $f \in F$ we have:

$$\begin{aligned}
(\text{curry } g)(\bigsqcup_{i\in\omega} d_i)(f) &= g(\bigsqcup_{i\in\omega} d_i, f) && \text{by definition of curry } g\\
&= \bigsqcup_{i\in\omega} g(d_i, f) && \text{by continuity of } g\\
&= \bigsqcup_{i\in\omega} ((\text{curry } g)(d_i)(f)) && \text{by definition of curry } g
\end{aligned}$$

□

As shown in Chapter 4 in order to define the denotational semantics of recursive definitions we provide a fixpoint operator. So it seems quite natural to introduce the fixpoint in our meta-theory.

Definition 7.18 (Fix)
Let D be a $CPO_\perp$. *We define the function* fix : $([D \to D] \to D)$ *as:*

$$\text{fix} \stackrel{\text{def}}{=} \bigsqcup_{i\in\omega} \lambda f.f^i(\perp_D)$$

Notice that the LUB does exist since $\{\lambda f.\ f^i(\perp_D)\}$ *is a chain of functions and* $([D \to D] \to D)$ *is complete.*

Theorem 7.19 (Continuity of fix)
Function fix : $([D \to D] \to D)$ *is continuous, namely* fix : $[[D \to D] \to D]$.

Proof. We know that $[[D \to D] \to D]$ is complete, thus if for all $i \in \omega$ the function $\lambda f.\ f^i(\bot_D)$ is continuous, then $\bigsqcup_{i\in\omega} \lambda f.\ f^i(\bot_D) = \text{fix}$ is also continuous. We prove that $\forall i.\ \lambda f.\ f^i(\bot_D)$ is continuous by mathematical induction.

i=0) $\lambda f.\ f^0(\bot_D)$ is a constant function and thus it is continuous.

i=n+1) Let us assume that $h = \lambda f.\ f^n(\bot_D)$ is continuous, namely that

$$h(\bigsqcup_{i\in\omega} f_i) = \left(\bigsqcup_{i\in\omega} f_i\right)^n (\bot_D) = \bigsqcup_{i\in\omega} f_i^n(\bot_D) = \bigsqcup_{i\in\omega} h(f_i)$$

and let us prove that $\left(\bigsqcup_{i\in\omega} f_i\right)^{n+1} (\bot_D) = \bigsqcup_{i\in\omega} f_i^{n+1}(\bot_D)$. In fact we have:

$$\begin{aligned}
\left(\bigsqcup_{i\in\omega} f_i\right)^{n+1} (\bot_D) &= \left(\bigsqcup_{i\in\omega} f_i\right)\left(\left(\bigsqcup_{i\in\omega} f_i\right)^n (\bot_D)\right) && \text{by definition}\\
&= \left(\bigsqcup_{i\in\omega} f_i\right)\left(\bigsqcup_{i\in\omega} f_i^n(\bot_D)\right) && \text{by the inductive hypothesis}\\
&= \bigsqcup_{i\in\omega} f_i\left(\bigsqcup_{j\in\omega} f_j^n(\bot_D)\right) && \text{by the definition of LUB of functions}\\
&= \bigsqcup_{i\in\omega}\bigsqcup_{j\in\omega} f_i\left(f_j^n(\bot_D)\right) && \text{by continuity of } f_i\\
&= \bigsqcup_{k\in\omega} f_k\left(f_k^n(\bot_D)\right) && \text{by Lemma 7.4 (switch lemma)}\\
&= \bigsqcup_{k\in\omega} f_k^{n+1}(\bot_D) && \text{by definition}
\end{aligned}$$

□

Finally we introduce the "let" operator, whose role is that of binding a name to a de-lifted expression. Note that the continuity of the "let" operator directly follows from the continuity of the lifting operator.

Definition 7.20 (`Let` operator)
Let $\mathscr{E}$ be a $CPO_\bot$ and $\lambda x.e$ a function in $[D \to E]$. We define the let *operator as follows:*

$$\textbf{let}\ x \Leftarrow d_\bot.\ e \stackrel{\text{def}}{=} \underbrace{\underbrace{\underbrace{(\lambda x.\ e)^*}_{D\to E}}_{D_\bot \to E}\ \underbrace{(d_\bot)}_{D_\bot}}_{E} = \begin{cases} \bot_E & \text{if } d_\bot = \bot_{D_\bot} \\ e\left[{}^{d}/_{x}\right] & \text{if } d_\bot = \lfloor d \rfloor \end{cases}$$

8. HOFL Denotational Semantics

In order to define the denotational semantics of a computer language we have to define by structural recursion an evaluation function from each syntactic domain to a semantic domain.
Since HOFL has only one syntactic domain (i.e. the set of typed terms t) we have only one evaluation function $[\![t]\!]$. However, since the terms are typed, the evaluation function is parametrised by the types. We have

$$[\![t : \tau]\!] : Env \longrightarrow (V_\tau)_\perp$$

Here $\rho \in Env$ are environments, which contain the values to be assigned to variables, in practice the free variables of t

$$\rho : Env = Var \longrightarrow \bigcup_\tau (V_\tau)_\perp$$

with the condition $\rho(x : \tau) \in (V_\tau)_\perp$.

In our denotational semantics of HOFL we distinguish between V_τ, where we find the meanings of the terms of type τ with canonical forms, and $(V_\tau)_\perp$, where the additional $\perp_{(V_\tau)_\perp}$ is the meaning of all the terms of type τ without a canonical form.

The actual semantic domains V_τ (and $(V_\tau)_\perp$) are defined by structural recursion on the syntax of types:

$$\begin{array}{ll} V_{int} = \mathbb{N} & (V_{int})_\perp = \mathbb{N}_\perp \\ V_{\tau_1 * \tau_2} = (V_{\tau_1})_\perp \times (V_{\tau_2})_\perp & (V_{\tau_1 * \tau_2})_\perp = ((V_{\tau_1})_\perp \times (V_{\tau_2})_\perp)_\perp \\ V_{\tau_1 \to \tau_2} = [(V_{\tau_1})_\perp \to (V_{\tau_2})_\perp] & (V_{\tau_1 \to \tau_2})_\perp = [(V_{\tau_1})_\perp \to (V_{\tau_2})_\perp]_\perp \end{array}$$

Notice that the recursive definition takes advantage of the domain constructors we defined in Chapter 7. While the lifiting $\mathbb{N}_\perp$ of the integer numbers $\mathbb{N}$ is strictly necessary, liftings on cartesian pairs and on continuous functions are actually optional, since cartesian products and functional domains are already $CPO_\perp$. We will discuss the motivation of our choice at the end of the following chapter.

8.1. HOFL Evaluation Function

Now we are ready to define the evaluation function, by structural recursion. As usual we start from the constant terms, then we define compositionally the more complicated ones.

8.1.1. Constants

We define the meaning of a constant as the obvious value on the lifted domains:

$$[\![n]\!]\rho = \lfloor n \rfloor$$

8.1.2. Variables

The meaning of a variable is defined by its value in the given environment:

$$[\![x]\!]\rho = \rho x$$

8.1.3. Binary Operators

We give the generic semantics of a binary operator as:

$$[\![t_0\ op\ t_1]\!]\rho = [\![t_0]\!]\rho\ \underline{op}_\perp\ [\![t_1]\!]\rho$$

where

$$\_\underline{op}_\perp\_ : (\mathbb{N}_\perp \times \mathbb{N}_\perp) \longrightarrow \mathbb{N}_\perp$$

$$x_1 \underline{op}_\perp x_2 = \begin{cases} \lfloor n_1\ \underline{op}\ n_2 \rfloor & \text{if } x_1 = \lfloor n_1 \rfloor \text{ and } x_2 = \lfloor n_2 \rfloor \\ \perp_{\mathbb{N}_\perp} & \text{otherwise} \end{cases}$$

note that for any operator $op \in \{+, -, \times\}$ in the syntax we have the corresponding function $\underline{op} : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ on the integers $\mathbb{N}$ and also the binary function $\underline{op}_\perp$ on $\mathbb{N}_\perp$. Notice also that $\underline{op}_\perp$ yields $\perp_{\mathbb{N}_\perp}$ when at least one of the two arguments is $\perp_{\mathbb{N}_\perp}$.

Since $\underline{op}_\perp$ is monotone over a domain with only finite chains then it is also continuous.

8.1.4. Conditional

In order to define the semantics of the conditional expression we use the conditional operator of the meta-language:

$$[\![\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1]\!]\rho = Cond\,([\![t]\!]\rho, [\![t_0]\!]\rho, [\![t_1]\!]\rho)$$

where

$$Cond_\tau : \mathbb{N}_\perp \times (V_\tau)_\perp \times (V_\tau)_\perp \longrightarrow (V_\tau)_\perp$$

$$Cond_\tau(v, d_0, d_1) = \begin{cases} d_0 & \text{if } v = \lfloor 0 \rfloor \\ d_1 & \text{if } v = \lfloor n \rfloor \wedge n \neq 0 \\ \perp_{(V_\tau)_\perp} & \text{if } v = \perp_{\mathbb{N}_\perp} \end{cases}$$

8.1.5. Pairing

For the pairing operator we have:

$$[\![(t_0, t_1)]\!]\rho = \lfloor ([\![t_0]\!]\rho, [\![t_1]\!]\rho) \rfloor$$

8.1.6. Projections

We define the projections by using the lifted version of π_1 and π_2 functions of the meta-language:

$$\begin{aligned} [\![\textbf{fst}(t)]\!]\rho &= \textbf{let}\, d \Leftarrow [\![t]\!]\rho.\ \pi_1 d = (\lambda d.\pi_1 d)^*\ [\![t]\!]\rho \\ [\![\textbf{snd}(t)]\!]\rho &= \textbf{let}\, d \Leftarrow [\![t]\!]\rho.\ \pi_2 d = (\lambda d.\pi_2 d)^*\ [\![t]\!]\rho \end{aligned}$$

as we said in the previous chapter the "let" operator allows to *de-lift* $[\![t]\!]\rho$ in order to apply projections π_1 and π_2.

8.1.7. Lambda Abstraction

Obviously we use the lambda operator of the lambda calculus:

$$[\![\lambda x.t]\!]\rho = \left\lfloor \lambda d.\ [\![t]\!]\rho[{}^d/_x] \right\rfloor$$

8.1.8. Function Application

We simply apply the de-lifted version of the function to its argument:

$$[\![(t_1\ t_0)]\!]\rho = \textbf{let}\, \varphi \Leftarrow [\![t_1]\!]\rho.\ \varphi([\![t_0]\!]\rho) = (\lambda\varphi.\varphi\,([\![t_0]\!]\rho))^*\ [\![t_1]\!]\rho$$

8.1.9. Recursion

We simply apply the fix operator of the meta-language:

$$[\![\mathbf{rec}\ \ x.t]\!]\,\rho = \mathbf{fix}\,\lambda d.\,[\![t]\!]\,\rho[{}^{d}/_{x}]$$

8.2. Typing the Clauses

Now we show that the clauses of the structural recursion are typed correctly.

Constants

$$\underbrace{[\![n : int]\!]\rho}_{(V_{int})_\perp = \mathbb{N}_\perp} = \underbrace{\underbrace{\lfloor n \rfloor}_{\mathbb{N}}}_{\mathbb{N}_\perp}$$

Variables

$$\underbrace{[\![x : \tau]\!]\rho}_{(V_\tau)_\perp} = \underbrace{\rho x}_{(V_\tau)_\perp}$$

Binary operations

$$\underbrace{[\![(t_0 : int\ op\ t_1 : int) : int]\!]\rho}_{(V_{int})_\perp} = \underbrace{\underbrace{[\![t_0]\!]\,\rho}_{(V_{int})_\perp}\quad \underbrace{\underbrace{op}_{\perp}}_{(V_{int})_\perp \times (V_{int})_\perp \to (V_{int})_\perp}\quad \underbrace{[\![t_1]\!]\rho}_{(V_{int})_\perp}}_{(V_{int})_\perp}$$

Conditional

$$\underbrace{[\![\mathbf{if}\ \ t_0 : int\ \ \mathbf{then}\ \ t_1 : \tau\ \ \mathbf{else}\ \ t_2 : \tau]\!]\rho}_{(V_\tau)_\perp} = \underbrace{\underbrace{Cond_\tau}_{\mathbb{N}_\perp \times (V_\tau)_\perp \times (V_\tau)_\perp \longrightarrow (V_\tau)_\perp}\ (\underbrace{[\![t_0]\!]\,\rho}_{(V_{int})_\perp}, \underbrace{[\![t_1]\!]\,\rho}_{(V_\tau)_\perp}, \underbrace{[\![t_2]\!]\,\rho}_{(V_\tau)_\perp})}_{(V_\tau)_\perp}$$

Pairing

$$\underbrace{[\![(t_0 : \tau_0, t_1 : \tau_1)]\!]\rho}_{(V_{\tau_0 * \tau_1})_\perp} = \underbrace{\lfloor \underbrace{([\![t_0]\!]\rho}_{(V_{\tau_0})_\perp}, \underbrace{[\![t_1]\!]\rho)}_{(V_{\tau_1})_\perp} \rfloor}_{((V_{\tau_0})_\perp \times (V_{\tau_1})_\perp)_\perp}$$

Projections

$$\underbrace{[\![\mathbf{fst}(t : \tau_0 * \tau_1)]\!]\rho}_{(V_{\tau_0})_\perp} = \mathbf{let}\ \ \underbrace{d}_{(V_{\tau_0})_\perp \times (V_{\tau_1})_\perp} \Leftarrow \underbrace{[\![t]\!]\rho}_{(V_{\tau_0 * \tau_1})_\perp}\ .\ \underbrace{\underbrace{\pi_1}_{(V_{\tau_0})_\perp \times (V_{\tau_1})_\perp \to (V_{\tau_0})_\perp}\ d}_{(V_{\tau_0})_\perp}$$

Lambda abstraction

$$\underbrace{[\![\lambda x : \tau_0.t : \tau_1]\!]\rho}_{(V_{\tau_0 \to \tau_1})_\perp} = \underbrace{\lfloor \underbrace{\lambda\ \underbrace{d}_{(V_{\tau_0})_\perp}\ .\ \underbrace{[\![t]\!]\rho[{}^{d}/_{x}]}_{(V_{\tau_1})_\perp}}_{[(V_{\tau_0})_\perp \to (V_{\tau_1})_\perp]} \rfloor}_{[(V_{\tau_0})_\perp \to (V_{\tau_1})_\perp]_\perp}$$

Function application

$$\underbrace{[\![(t_1 : \tau_0 \longrightarrow \tau_1\ t_0 : \tau_0)]\!]\rho}_{(V_{\tau_1})_\perp} = \textbf{let}\ \underbrace{\varphi}_{[(V_{\tau_0})_\perp \to (V_{\tau_1})_\perp]} \Leftarrow \underbrace{[\![t_1]\!]\rho}_{(V_{\tau_0\to\tau_1})_\perp}\ .\underbrace{\varphi(\underbrace{[\![t_0]\!]\rho}_{(V_{\tau_0})_\perp})}_{(V_{\tau_1})_\perp}$$

Recursion

$$\underbrace{[\![\textbf{rec}\ x : \tau.t : \tau]\!]\rho}_{(V_\tau)_\perp} = \underbrace{\underbrace{fix}_{[(V_\tau)_\perp \to (V_\tau)_\perp] \to (V_\tau)_\perp}\ \underbrace{\underbrace{\lambda d}_{(V_\tau)_\perp}\ .\ \underbrace{[\![t]\!]\rho[^d/_x]}_{(V_\tau)_\perp}}_{[(V_\tau)_\perp \to (V_\tau)_\perp]}}_{(V_\tau)_\perp}$$

Example 8.1
Let us see some examples of evaluation of the denotational semantics. We consider three similar terms f, g, h such that f and h have the same denotational semantics while g has a different semantics because it requires a parameter x to be evaluated even if it is not used.

1. $f \stackrel{\text{def}}{=} \lambda x : int.3$
2. $g \stackrel{\text{def}}{=} \lambda x : int.\ \textbf{if}\ x\ \textbf{then}\ 3\ \textbf{else}\ 3$
3. $h \stackrel{\text{def}}{=} \textbf{rec}\ y : int \to int.\lambda x : int.3$

1. $[\![f]\!]\rho = [\![\lambda x : int.3]\!]\rho = \lfloor \lambda d.[\![3]\!]\rho[^d/_x] \rfloor = \lfloor \lambda d.\lfloor 3 \rfloor \rfloor$
2. $[\![g]\!]\rho = [\![\lambda x : int.\ \textbf{if}\ x\ \textbf{then}\ 3\ \textbf{else}\ 3]\!]\rho = \lfloor \lambda d.[\![\textbf{if}\ x\ \textbf{then}\ 3\ \textbf{else}\ 3]\!]\rho[^d/_x] \rfloor = \lfloor \lambda d.Cond(d, \lfloor 3 \rfloor, \lfloor 3 \rfloor) \rfloor = \lfloor \lambda d.\ \textbf{let}\ x \Leftarrow d.\lfloor 3 \rfloor \rfloor$
3. $[\![h]\!]\rho = [\![\textbf{rec}\ y : int \to int.\lambda x : int.3]\!]\rho = \textbf{fix}\ \lambda d.[\![\lambda x.3]\!]\rho[^d/_x] = \textbf{fix}\ \lambda d.\lfloor \lambda d'.\lfloor 3 \rfloor \rfloor$
   - $d_0 = \perp_{[\mathbb{N}_\perp \to \mathbb{N}_\perp]_\perp}$
   - $d_1 = (\lambda d.\lfloor \lambda d'.\lfloor 3 \rfloor \rfloor)\perp = \lfloor \lambda d'.\lfloor 3 \rfloor \rfloor$
   - $d_2 = (\lambda d.\lfloor \lambda d'.\lfloor 3 \rfloor \rfloor)\lfloor \lambda d'.\lfloor 3 \rfloor \rfloor = \lfloor \lambda d'.\lfloor 3 \rfloor \rfloor = d_1 = [\![h]\!]\rho = [\![f]\!]\rho$

8.3. Continuity of Meta-language's Functions

In order to show that the semantics is well defined we have to show that all the functions we employ in the definition are continuous.

Theorem 8.2
The following functions are continuous:

- $\underline{op}_\perp$
- $Cond$
- $(\_, \_)$
- $\pi_1,\ \pi_2$
- let
- $apply$
- fix

Proof.

- $\underline{op}_\perp$: we have already proved the continuity of $\underline{op}_\perp$ when it was introduced.
- *Cond*: By using the Theorem 7.13, we can prove the continuity on each parameter separately. Let us show the continuity on the first parameter. Since chains are finite, it is enough to prove monotonicity. We fix d_1 and d_2 and we prove the monotonicity of $Cond_\tau(x, d_1, d_2)$
  - for the case $\perp_{\mathbf{N}_\perp} \sqsubseteq n$ then obviously $\forall n \in \mathbf{N}_\perp\ Cond_\tau(\perp_{\mathbf{N}_\perp}, d_1, d_2) \sqsubseteq Cond_\tau(n, d_1, d_2)$, namely $\perp_{(V_\tau)_\perp} \sqsubseteq d_1$ or $\perp_{(V_\tau)_\perp} \sqsubseteq d_2$.
  - for the case $n \sqsubseteq n'$ with $n \neq \perp_{\mathbf{N}_\perp}$, since $\mathbf{N}_\perp$ is a flat domain we have $n = n'$ so obviously $Cond_\tau(n, d_1, d_2) \sqsubseteq Cond_\tau(n', d_1, d_2)$

  Now let us show the continuity on the second parameter, namely we fix v and d and prove that

  $$Cond_\tau(v, \bigsqcup_{i\in\omega} d_i, d) = \bigsqcup_{i\in\omega} Cond_\tau(v, d_i, d)$$

  - if $v = \perp_{\mathbf{N}_\perp}$, then $Cond_\tau$ is the constant function returning $\perp_{\mathbb{N}_\perp}$;
  - if $v = \lfloor 0 \rfloor$, then $Cond_\tau$ it is the identity function;
  - if $v = \lfloor n \rfloor$ with $n \neq 0$, then $Cond_\tau$ it is the constant function returning d.

  In all cases $Cond_\tau$ is continuous.

  Continuity on the third parameter is analogous.
- $(\_, \_)$ we can use the Theorem 7.13 which allows to show separately the continuity on each parameter. If we fix the first element we have $(d, \bigsqcup_{i\in\omega} d_i) = \bigsqcup_{i\in\omega}(d, d_i)$ by Theorem 7.1. The same holds for the second parameter.
- π_1 and π_2 are continuous as shown by Theorem 7.3.
- the **let** function is continuous since $(\_)^*$ is continuous by Theorem 7.10.
- apply is continuous as shown by Theorem 7.15
- fix is continuous as shown by Theorem 7.19.

□

In the previous theorem we have omitted the proofs for lambda abstraction and recursion, in the next theorem we fill these gaps.

Theorem 8.3
Let t be a well typed term of HOFL then the following holds:

- $(\lambda d : (V_{\tau_1})_\perp.[\![t : \tau_2]\!]\rho[^d/_x]) : (V_{\tau_1})_\perp \longrightarrow (V_{\tau_2})_\perp$ *is a continuous function.*
- fix $\lambda d.[\![t]\!]\rho[^d/_x]$ *is a continuous function.*

Proof. Let us show the first proposition by structural induction on t and for any number of arguments $\rho[^d/_x][^{d'}/_y]\ldots$.

- $t = x$: $\lambda d.[\![x]\!]\rho[^d/_x]$ is equal to the identity function $\lambda d.d$ which is obviously continuous.
- $t = t_1\ op\ t_2$: $\lambda d.[\![t_1\ op\ t_2]\!]\rho[^d/_x] = \lambda d.[\![t_1]\!]\rho[^d/_x]\ \underline{op}_\perp\ [\![t_2]\!]\rho[^d/_x]$ it is continuous since $\underline{op}_\perp$ is continuous and by the inductive hypothesis, and since the composition of continuous functions yields a continuous function by Theorem 7.11.
- $t = \lambda y.t'$: $\lambda d.[\![\lambda y.t']\!]\rho[^d/_x]$ is obviously continuous if $x = y$. Otherwise if $x \neq y$ we have by induction hypothesis that $\lambda(d, d').[\![t']\!]\rho[^d/_x, ^{d'}/_y]$ is continuous, then:

  $$\begin{array}{rcll} \text{curry}(\lambda(d, d').[\![t']\!]\rho[^d/_x, ^{d'}/_y]) & = & \lambda d.\lambda d'.[\![t']\!]\rho[^d/_x][^{d'}/_y] & \text{is continuous since curry is continuous} \\ & = & \lambda d.[\![\lambda y.t']\!]\rho[^d/_x] & \text{by definition} \end{array}$$

We leave the remaining cases as an exercise.
To prove the second proposition we note that, fix $\lambda d.[\![t]\!]\rho[^d/_x]$ is the application of a continuous function (i.e., the function fix, by Theorem 7.19) to a continuous argument (i.e., $\lambda d.[\![t]\!]\rho[^d/_x]$, continuous by the first part of this theorem) so it is continuous by Theorem 7.15. □

8.4. Substitution Lemma

We conclude this chapter by stating some useful theorems. The most important is the *Substitution Lemma* which states that the substitution operator commutes with the interpretation function.

Theorem 8.4 (Substitution Lemma)
Let t, t' be well typed terms: we have

$$\left[\!\left[t[^{t'}/_{x}] \right]\!\right] \rho = [\![t]\!] \rho[^{[\![t']\!]\rho}/_{x}]$$

Proof. By structural induction. □

The substitution lemma is an important result, as it implies the compositionality of denotational semantics:

$$[\![t_1]\!] \rho = [\![t_2]\!] \rho \Rightarrow \left[\!\left[t[^{t_1}/_{x}] \right]\!\right] \rho = \left[\!\left[t[^{t_2}/_{x}] \right]\!\right] \rho$$

In words, replacing a variable x with a term t' in a term t returns a term $t[^{t'}/_{x}]$ whose denotational semantics $\left[\!\left[t[^{t'}/_{x}] \right]\!\right] \rho = [\![t]\!] \rho[^{[\![t']\!]\rho}/_{x}]$ depends only on the denotational semantics $[\![t']\!] \rho$ of t' and *not* on t' itself.

Theorem 8.5
Let t be a well-defined term of HOFL. Let $\rho, \rho' \in Env$ such that $\forall\ x \in fv(t).\ \rho(x) = \rho'(x)$ then:

$$[\![t]\!] \rho = [\![t]\!] \rho'$$

Proof. By structural induction. □

Theorem 8.6
Let $c \in C_\tau$ be a closed term in canonical form of type τ. Then we have:

$$\forall\ \rho \in Env.\ [\![c]\!] \rho \neq \bot_{(V_\tau)_\bot}$$

Proof. Immediate, by inspection of the clauses for terms in canonical forms. □

9. Equivalence between HOFL denotational and operational semantics

As we have done for IMP, now we address the relation between the denotational and operational semantics of HOFL. We would like to prove a complete equivalence, as in the case of IMP:

$$t \longrightarrow c \overset{?}{\iff} \forall \rho.\ [\![t]\!]\rho = [\![c]\!]\rho$$

But, as we are going to show, the situation in the case of HOFL is more complex and the $\Longleftarrow$ case does not hold, i.e., the completeness holds but not the correctness:

$$t \longrightarrow c \Rightarrow \forall \rho.\ [\![t]\!]\rho = [\![c]\!]\rho \quad \text{but} \quad (\forall \rho.\ [\![t]\!]\rho = [\![c]\!]\rho) \not\Rightarrow t \longrightarrow c$$

Let us consider an example which shows the difference between the denotational and the operational semantics.

Example 9.1
Let $c = \lambda x.\ x$ and $t = \lambda x.\ x + 0$ be two HOFL terms, where $x : int$, then we have:

$$[\![t]\!]\rho = [\![c]\!]\rho$$

but

$$t \not\twoheadrightarrow c$$

In fact we have:

$$[\![t]\!]\rho = [\![\lambda x.\ x + 0]\!]\rho = \lfloor \lambda d.\ d \underline{+}_\perp \lfloor 0 \rfloor \rfloor = \lfloor \lambda d.d \rfloor = [\![\lambda x.\ x]\!]\rho = [\![c]\!]\rho$$

but for the operational semantics we have that both $\lambda x.x$ and $\lambda x.\ x + 0$ are already in canonical form and $t \neq c$.

The counterexample shows that at least for the functional type $int \to int$, there are different canonical forms with the same denotational semantics, namely terms which compute the same function $[\mathbb{N}_\perp \to \mathbb{N}_\perp]_\perp$. One could think that a refined version of our operational semantics (e.g. one which could apply an axiom like $x + 0 = 0$) would be able to identify exactly all the canonical forms which compute the same function. However this is not possible on computability grounds: since HOFL is able to compute all computable functions, the set of canonical terms which compute the same function cannot be recursively enumerable, while the set of theorems of every inference system is recursively enumerable.

9.1. Completeness

We are ready to show the completeness of the denotational semantics of HOFL w.r.t. the operational one.

Theorem 9.2 (Completeness)
Let $t : \tau$ be a HOFL closed term and let $c : \tau$ be a canonical form. Then we have:

$$t \to c \Longrightarrow \forall \rho \in Env\ [\![t]\!]\rho = [\![c]\!]\rho$$

Proof. As usual we proceed by rule induction. So we will prove $P(t \to c) \overset{\text{def}}{=} \forall \rho.\ [\![t]\!]\rho = [\![c]\!]\rho$ on each rule by assuming the premises.

canonical forms (integers, pairs, abstraction):

$$\frac{}{c \to c}$$

We have to prove $P(c \to c)$ that is by definition $\forall\rho.\ [\![c]\!]\rho = [\![c]\!]\rho$, which is obviously true.

mathematical operators:

$$\frac{t_1 \to n_1 \quad t_2 \to n_2}{t_1 \text{ op } t_2 \to n_1 \underline{op}\, n_2}$$

We have to prove $P(t_1 \text{ op } t_2 \to n_1 \underline{op}\, n_2) \stackrel{\text{def}}{=} \forall\rho.\ [\![t_1 \text{ op } t_2]\!]\rho = \left[\!\!\left[n_1 \underline{op}\, n_2\right]\!\!\right]\rho$.

We can assume the inductive hypotheses: $P(t_1 \to n_1) \stackrel{\text{def}}{=} \forall\rho.\ [\![t_1]\!]\rho = [\![n_1]\!]\rho = \lfloor n_1 \rfloor$ and $P(t_2 \to n_2) \stackrel{\text{def}}{=} \forall\rho.\ [\![t_2]\!]\rho = [\![n_2]\!]\rho = \lfloor n_2 \rfloor$.

We have $[\![t_1 \text{ op } t_2]\!]\rho = [\![t_1]\!]\rho\ \underline{op}_\perp\ [\![t_2]\!]\rho = \lfloor n_1 \rfloor\ \underline{op}_\perp \lfloor n_2 \rfloor = \lfloor n_1 \underline{op}\, n_2 \rfloor = \left[\!\!\left[n_1 \underline{op}\, n_2\right]\!\!\right]\rho$.

conditional:

$$\frac{t \to 0 \quad t_0 \to c_0}{\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \to c_0}$$

We want to prove $P(\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \to c_0) \stackrel{\text{def}}{=} \forall\rho.\ [\![\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1]\!]\rho = [\![c_0]\!]\rho$.

We can assume $P(t \to 0) \stackrel{\text{def}}{=} \forall\rho.\ [\![t]\!]\rho = [\![0]\!]\rho = \lfloor 0 \rfloor$ and $P(t_0 \to c_0) \stackrel{\text{def}}{=} \forall\rho.\ [\![t_0]\!]\rho = [\![c_0]\!]\rho$.

We have $[\![\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1]\!]\rho = Cond([\![t]\!]\rho, [\![t_0]\!]\rho, [\![t_1]\!]\rho) = Cond(\lfloor 0 \rfloor, [\![t_0]\!]\rho, [\![t_1]\!]\rho) = [\![t_0]\!]\rho = [\![c_0]\!]\rho$.

The same construction holds for the second rule of the conditional operator.

projections

$$\frac{t \to (t_0, t_1) \quad t_0 \to c_0}{\textbf{fst}(t) \to c_0}$$

We want to prove $P(\textbf{fst}(t) \to c_0) \stackrel{\text{def}}{=} \forall\rho.\ [\![\textbf{fst}(t)]\!]\rho = [\![c_0]\!]\rho$.

We can assume $P(t \to (t_0, t_1)) \stackrel{\text{def}}{=} \forall\rho.\ [\![t]\!]\rho = [\![(t_0, t_1)]\!]\rho$ and $P(t_0 \to c_0) \stackrel{\text{def}}{=} \forall\rho.\ [\![t_0]\!]\rho = [\![c_0]\!]\rho$.

We have:

$[\![\textbf{fst}(t)]\!]\rho$	=	$\textbf{let } v \Leftarrow [\![t]\!]\rho.\ \pi_1 v$	by definition
	=	$\textbf{let } v \Leftarrow [\![(t_0, t_1)]\!]\rho.\ \pi_1 v$	by the first inductive hypothesis
	=	$\textbf{let } v \Leftarrow \lfloor ([\![t_0]\!]\rho, [\![t_1]\!]\rho) \rfloor.\ \pi_1 v$	by definition
	=	$\pi_1([\![t_0]\!]\rho, [\![t_1]\!]\rho)$	by de-lifting
	=	$[\![t_0]\!]\rho$	by definition of projection
	=	$[\![c_0]\!]\rho$	by the second inductive hypothesis

The same holds for the **snd** operator.

application

$$\frac{t_1 \to \lambda x.t_1' \quad t_1'[^{t_0}/_x] \to c}{(t_1\ t_0) \to c}$$

We want to prove $P((t_1\ t_0) \to c) \stackrel{\text{def}}{=} \forall\rho.\ [\![(t_1\ t_0)]\!]\rho = [\![c]\!]\rho$.

We can assume $P(t_1 \to \lambda x.\ t_1') \stackrel{\text{def}}{=} \forall\rho.\ [\![t_1]\!]\rho = \left[\!\!\left[\lambda x.\ t_1'\right]\!\!\right]\rho$ and $P(t_1'[^{t_0}/_x] \to c) \stackrel{\text{def}}{=} \forall\rho.\ \left[\!\!\left[t_1'[^{t_0}/_x]\right]\!\!\right]\rho = [\![c]\!]\rho$.

We have:

$[\![(t_1\ t_0)]\!]\rho$	=	$\textbf{let } \varphi \Leftarrow [\![t_1]\!]\rho.\ \varphi([\![t_0]\!]\rho)$	by definition
	=	$\textbf{let } \varphi \Leftarrow \left[\!\!\left[\lambda x.t_1'\right]\!\!\right]\rho.\ \varphi([\![t_0]\!]\rho)$	by the first inductive hypothesis
	=	$\textbf{let } \varphi \Leftarrow \left\lfloor \lambda d.\ \left[\!\!\left[t_1'\right]\!\!\right]\rho[^d/_x] \right\rfloor.\ \varphi([\![t_0]\!]\rho)$	by definition
	=	$(\lambda d.\ \left[\!\!\left[t_1'\right]\!\!\right]\rho[^d/_x])\ ([\![t_0]\!]\rho)$	by de-lifting
	=	$\left[\!\!\left[t_1'\right]\!\!\right]\rho[^{[\![t_0]\!]\rho}/_x]$	by definition of functional application
	=	$\left[\!\!\left[t_1'[^{t_0}/_x]\right]\!\!\right]\rho$	by the substitution lemma
	=	$[\![c]\!]\rho$	by the second inductive hypothesis

recursion:

$$\frac{t[^{\mathbf{rec}\,x.t}/_x] \to c}{\mathbf{rec}\,x.t \to c}$$

We want to prove $P(\mathbf{rec}\,x.t \to c) \stackrel{\text{def}}{=} \forall\rho.\ [\![\mathbf{rec}\,x.t]\!]\rho = [\![c]\!]\rho$.

We can assume $P(t[^{\mathbf{rec}\,x.t}/_x] \to c) \stackrel{\text{def}}{=} \forall\rho.\ [\![t[^{\mathbf{rec}\,x.t}/_x]]\!]\rho = [\![c]\!]\rho$.

We have:

$$\begin{array}{rcll} [\![\mathbf{rec}\,x.t]\!]\rho & = & \text{fix}\ \lambda d.\,[\![t]\!]\rho[^d/_x] & \text{by definition} \\ & = & [\![t]\!]\rho[^{[\![\mathbf{rec}\,x.t]\!]\rho}/_x] & \text{by the fixpoint property} \\ & = & [\![t[^{\mathbf{rec}\,x.t}/_x]]\!]\rho & \text{by the substitution lemma} \\ & = & [\![c]\!]\rho & \text{by inductive hypothesis} \end{array}$$

□

9.2. Equivalence (on Convergence)

Now we define the concept of termination for the denotational and the operational semantics.

Definition 9.3 (Operational convergence)
Let $t : \tau$ be a closed term of HOFL, we define the following predicate:

$$t\downarrow \iff \exists c \in C_\tau.\ t \longrightarrow c.$$

If the predicate holds for t, then we say that t converges operationally.
We say that t diverges *and write $t\uparrow$ if t does not converge.*

Obviously, a term t converges operationally if the term can be evaluated to a canonical form c. For the denotational semantics we have that a term t converges if the evaluation function applied to t takes a value different from $\perp$.

Definition 9.4 (Denotational convergence)
Let t be a closed term of HOFL with type τ, we define the following predicate:

$$t\Downarrow \iff \exists v \in V_\tau.\ [\![t]\!]\rho = \lfloor v \rfloor.$$

If the predicate holds for t then we say that t converges denotationally.

We aim to prove that the two semantics agree at least on the notion of convergence.
As we will see, we can easily prove the implication:

$$t\downarrow \Longrightarrow t\Downarrow$$

For the opposite implication,

$$t\Downarrow \Longrightarrow t\downarrow$$

the property holds but the proof is not straightforward: We cannot simply rely on structural induction; instead it is necessary to introduce a particular order relation. We are not giving the full details of the proof, but we show that the standard structural induction does not help in proving the (left implication of) convergence agreement. Those who are interested in the full proof can refer to Winskel's book referenced in the introduction.

Theorem 9.5
Let t be a closed typable term of HOFL. Then we have:

$$t\downarrow \Longrightarrow t\Downarrow$$

Proof. If $t \longrightarrow c$, then $[\![t]\!]\rho = [\![c]\!]\rho$ for Theorem 9.2. But $[\![c]\!]\rho$ is a lifted value, (see Theorem 8.6) and thus it is different than $\perp$. □

While the converse implication $t \Downarrow \Longrightarrow t \downarrow$ also holds, we give some insight on the reason why the usual structural induction does not work for proving it. Let us consider function application $(t_1\ t_0)$. We assume by structural induction $t_1 \Downarrow \Rightarrow t_1 \downarrow$ and $t_0 \Downarrow \Rightarrow t_0 \downarrow$. Now we assume $(t_1\ t_0) \Downarrow$ and would like to prove that $(t_1\ t_0) \downarrow$, i.e., that $\exists c.\ (t_1\ t_0) \to c$. By definition of denotational semantics we have $t_1 \Downarrow$. In fact

$$[\![(t_1\ t_0)]\!]\rho = \mathbf{let}\ \varphi \Leftarrow [\![t_1]\!]\rho.\ \varphi([\![t_0]\!]\rho)$$

and therefore $[\![(t_1\ t_0)]\!]\rho \neq \perp$ requires $[\![t_1]\!]\rho \neq \perp$. By inductive hypothesis we then have $t_1 \downarrow$ and by definition of the operational semantics $t_1 \longrightarrow \lambda x.t_1'$ for some x and t_1'. By completeness we also have $[\![t_1]\!]\rho = \left[\!\left[\lambda x.t_1'\right]\!\right]\rho$. By denotational semantics definition we have:

$$\begin{array}{lcll}
[\![(t_1\ t_0)]\!]\rho & = & \mathbf{let}\ \varphi \Leftarrow \left\lfloor \lambda d.\left[\!\left[t_1'\right]\!\right]\rho[^{d}/_{x}]\right\rfloor.\ \varphi([\![t_0]\!]\rho) & \text{(see above)} \\
& = & (\lambda d.\left[\!\left[t_1'\right]\!\right]\rho[^{d}/_{x}])\ ([\![t_0]\!]\rho) & \text{by de-lifting} \\
& = & \left[\!\left[t_1'\right]\!\right]\rho[^{[\![t_0]\!]\rho}/_{x}] & \text{by functional application} \\
& = & \left[\!\left[t_1'[^{t_0}/_{x}]\right]\!\right]\rho & \text{by the substitution lemma}
\end{array}$$

So $(t_1\ t_0) \Downarrow$ if and only if $t_1'[^{t_0}/_{x}] \Downarrow$. We would like to conclude by structural induction that $t_1'[^{t_0}/_{x}] \downarrow$ and then prove the theorem by using the rule:

$$\frac{t_1 \to \lambda x.t_1' \quad t_1'[^{t_0}/_{x}] \to c}{(t_1\ t_0) \to c}$$

but this is incorrect since $t_1'[^{t_0}/_{x}]$ is not a sub-term of $(t_1\ t_0)$.

9.3. Operational and Denotational Equivalence

In this section we take a closer look at the relationship between the operational and denotational semantics of HOFL. In the introduction of this chapter we said that the denotational semantics is more abstract than the operational. In order to study this relationship we now introduce two equivalence relations between terms. Operationally two terms are equivalent if they both diverge or have the same canonical form.

Definition 9.6 (Operational equivalence)
Let t_0 and t_1 be two well-typed terms of HOFL then we define a binary relation:

$$t_0 \equiv_{op} t_1 \Longleftrightarrow (t_0 \uparrow\ \wedge\ t_1 \uparrow) \vee (t_0 \to c\ \wedge\ t_1 \to c)$$

And we say that t_0 is operationally equivalent to t_1.

Obviously we have the denotational counterpart of the definition.

Definition 9.7 (Denotational equivalence)
Let t_0 and t_1 be two well-typed terms of HOFL then we define a binary relation:

$$t_0 \equiv_{den} t_1 \Longleftrightarrow \forall \rho.\ [\![t_0]\!]\rho = [\![t_1]\!]\rho$$

And we say that t_0 is denotationally equivalent to t_1.

From Theorem 9.2 (completeness) it follows that:

$$\equiv_{op} \Rightarrow \equiv_{den}$$

As pointed out in Example 9.1, the opposite does not hold:

$$\equiv_{den} \not\Rightarrow \equiv_{op}$$

So in this sense we can say that the denotational semantics is more abstract then the operational one. Note that if we assume $t_0 \equiv_{den} t_1$ and $t_0, t_1 \neq \bot$ then we can only conclude that $t_0 \to c_0$ and $t_1 \to c_1$ for some suitable c_0 and c_1. So we have $[\![c_0]\!]\rho = [\![c_1]\!]\rho$, but nothing ensures that $c_0 = c_1$ as shown in the Example 9.1 at the beginning of this chapter.

Now we prove that if we restrict our attention only to the integers terms of HOFL, then the corresponding operational and denotational semantics completely agree. This is due to the fact that if c_0 and c_1 are canonical forms in C_{int} then it holds that $[\![c_0]\!]\rho = [\![c_1]\!]\rho \Leftrightarrow c_0 = c_1$.

Theorem 9.8
Let $t : int$ be a closed term of HOFL and $n \in \omega$. Then:

$$\forall \rho.\ [\![t]\!]\rho = \lfloor n \rfloor \Longleftrightarrow t \longrightarrow n$$

Proof.

$\Rightarrow$) If $[\![t]\!]\rho = \lfloor n \rfloor$, then $t \Downarrow$ and thus $t \downarrow$ by the soundness of denotational semantics (not proved here), namely $\exists n'$ such that $t \longrightarrow n'$, but $[\![t]\!]\rho = \lfloor n' \rfloor$ by Theorem9.2, thus $n = n'$ and $t \longrightarrow n$.

$\Leftarrow$) Just Theorem 9.2.

□

9.4. A Simpler Denotational Semantics

In this section we introduce a simpler denotational semantics which we call *unlifted*, which does not use the lifted domains. This semantics is simpler but also less expressive than the lifted one.

We define the following new domains:

$$D_{int} = \mathbf{N}_\bot$$

$$D_{\tau_1 * \tau_2} = D_{\tau_1} \times D_{\tau_2}$$

$$D_{\tau_1 \to \tau_2} = [D_{\tau_1} \to D_{\tau_2}]$$

So we can simply define the interpretation function $[\![t : \tau]\!] \in D_\tau$ as follows:

$$\begin{array}{rcl}
\text{(as before)} & & \\
[\![n]\!]\rho & = & \lfloor n \rfloor \\
[\![x]\!]\rho & = & \rho\, x \\
[\![t_1\ op\ t_2]\!]\rho & = & [\![t_1]\!]\rho\ \underline{op}_\bot\ [\![t_2]\!]\rho \\
[\![if\ t_0\ then\ t_1\ else\ t_2]\!]\rho & = & Cond([\![t_0]\!]\rho, [\![t_1]\!]\rho, [\![t_2]\!]\rho) \\
[\![\mathbf{rec}\ x.t]\!]\rho & = & \mathrm{fix}\, \lambda d.\, [\![t]\!]\rho[^d/_x] \\
\text{(updated definitions)} & & \\
[\![(t_1, t_2)]\!]\rho & = & ([\![t_1]\!]\rho, [\![t_2]\!]\rho) \\
[\![\mathbf{fst}(t)]\!]\rho & = & \pi_1([\![t]\!]\rho) \\
[\![\mathbf{snd}(t)]\!]\rho & = & \pi_2([\![t]\!]\rho) \\
[\![\lambda x.t]\!]\rho & = & \lambda d.\, [\![t]\!]\rho[^d/_x] \\
[\![(t_1\ t_2)]\!]\rho & = & ([\![t_1]\!]\rho)([\![t_2]\!]\rho)
\end{array}$$

Note that the "unlifted" semantics differ from the "lifted" one only in the cases of pairing, projections, abstraction and application. Obviously, on the one hand this denotational semantics is much simpler. On the other hand this semantics is more abstract then the "lifted" one and does not express some interesting properties. For instance, consider the two HOFL terms:

$$t_1 = \mathbf{rec}\ x.x : int \longrightarrow int \quad \text{and} \quad t_2 = \lambda x.\, \mathbf{rec}\ y.y : int \longrightarrow int$$

In the lifted semantics we have $[\![t_1]\!]\rho = \bot_{[\mathbb{N}_\bot \longrightarrow \mathbb{N}_\bot]_\bot}$ and $[\![t_2]\!]\rho = \lfloor \bot_{[\mathbb{N}_\bot \longrightarrow \mathbb{N}_\bot]} \rfloor$ while in unlifted semantics $[\![t_1]\!]\rho = [\![t_2]\!]\rho = \bot_{[\mathbb{N}_\bot \longrightarrow \mathbb{N}_\bot]}$. Note however that $t_1 \not\downarrow$ while $t_2 \downarrow$, thus the completeness property $t \downarrow \Rightarrow \Downarrow t$ does not hold for the unlifted semantics, at least for $t : int \longrightarrow int$, since $t_2 \downarrow$ but $t_2 \not\Downarrow$. However, completeness holds for the unlifted semantics in the case of integers.

As a final comment, notice that the existence of two different, both reasonable, denotational semantics for HOFL shows that denotational semantics is, to some extent, an arbitrary construction, which depends on the properties one wants to express.

Part III.

Concurrency and Logic

10. CCS, the Calculus for Communicating Systems

In the last decade computer science technologies have boosted the growth of large scale distributed and concurrent systems. Their formal study introduces several aspects which are not present in the case of sequential systems. In particular, it emerges the necessity to deal with non-determinism, parallelism, interaction and infinite behaviour. Non-determinism is needed to model time races between different signals and to abstract from programming details which are irrelevant for the interaction behaviour of systems. Parallelism allows agents to perform tasks independently. For our purposes, this will be modelled by using non-determinism. Interaction allows to describe the behaviour of the system from the observational point of view (i.e., the behaviour that the system shows to an external observer). Infinite behaviour allows to study the semantics of non-terminating processes useful in many different contexts (e.g., think about the modelling of operating systems). Accordingly, from the theoretical point of view, some additional efforts must be spent to extend the semantics of sequential systems to that of concurrent systems in a proper way.

As we saw in the previous chapters, the study of sequential programming languages brought to different semantics which allows to prove many different properties. In this chapter we introduce CCS, a specification language which allows to describe concurrent communicating systems. Such systems are composed of agents (i.e., processes) performing tasks by communicating each other through channels.

While infinite behaviour is accounted for also in IMP and HOFL (consider, e.g., the programs **rec** $x.\ x$ and **while** `true` **do skip**), unlike the sequential languages, CCS does not assign the same semantics to all the infinite behaviours (recall that if a sequential program does not terminate its semantics is equal to $\bot$ in the denotational semantics).

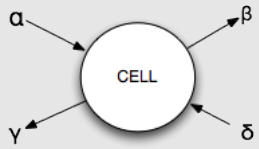
The semantics of sequential languages can be given by defining functions. In the presence of non-deterministic behaviours functions do not seem to provide the right tool to abstract the behaviour of concurrent systems. As we will see this problem is worked out by modelling the system behaviour as a *labelled transition system*, i.e. as a set of states equipped with a transition relation which keeps track of the interactions between the system and its environment. As a consequence, it makes little sense to talk about denotational semantics of CCS. In addition, recall that the denotational semantics is based on fix point theory over CPOs, while it turns out that several interesting properties of non-deterministic systems with non-trivial infinite behaviours are not inclusive (as it is the case of fairness, described in Example 5.14), thus the principle of computational induction does not apply to such properties. Moreover labelled transition systems are often equipped with a modal logic counterpart, which allows to express and prove the relevant properties of the modelled system.

Let us show how CCS works with an example.

Example 10.1 (Dynamic concurrent stack)
Let us consider the problem of modelling an extensible stack. The idea is to represent the stack as a collection of cells that communicate by sending and receiving data over some channels:

- *the* send *operation of data x over channel α is denoted by $\bar{\alpha}x$;*
- *the* receive *operation of data x over channel α is denoted by αx.*

We have one so-called process *(or* agent*) for each cell of the stack. Each process can store one or two values or send a stored value to other processes. All processes involved in the stack have basically the same structure. We represent graphically one of such processes as follow:*



The figure shows that a cell has four channels $\alpha, \beta, \gamma, \delta$ that can be used to communicate with other cells. In general, a process can perform bidirectional operation on its channels. In this particular case, each cell will use each channel for either input or output operations:

- *α is the input channel to receive data from either the external environment or the left cell;*
- *γ is the channel used to send data to either the external environment or the left cell;*
- *β is the channel used to send data to the right cell and to manage the end of the stack;*
- *δ is the channel used to receive data from the right cell and to manage the end of the stack.*

In the following we specify the possible states that a cell can have, each corresponding to some specific behaviour. Note that some states are parametric to certain values that represent, e.g., the particular values stored in that cell. The four possible states are described below:

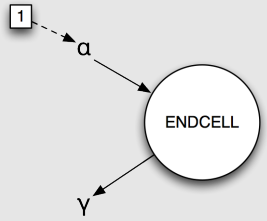
- $CELL_0 = \delta x.$ **if** $x = \$$ **then** $ENDCELL$ **else** $CELL_1(x)$
  This state represents the empty cell. The agent waits for data from the channel δ, when a value is received the agent controls if it is equal to a special termination character \$. If the received data is \$ this means that the agent is the last cell, so it switches to the ENDCELL state. Otherwise, if x is a new value, the agent stores it by moving to the state $CELL_1(x)$.

- $CELL_1(y) = \alpha x.CELL_2(x, y) + \bar{\gamma} y.CELL_0$
  This state represents an agent which contains a value y. In this case the cell can non-deterministically wait for new data on α or send the stored data on γ. In the first case, the cell must store the new value and send the old value to another agent: this task is performed by $CELL_2(x, y)$. In the second case, it is assumed that some other agent wants to extract the stored value; then the cell becomes empty by switching to the state $CELL_0$. Note that the + operator represents a non-deterministic choice performed by the agent. However a particular choice could be forced on a cell by the behaviour of the other cells.

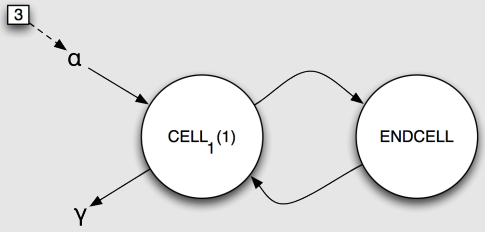
- $CELL_2(x, y) = \bar{\beta} y.CELL_1(x)$
  In this case the cell has currently two parameters x (the newly received value) and y (the previously stored value). The agent must cooperate with its neighbors cells in order to perform a right shift of the data. In order to do that the agent communicates to the right neighbour the old stored value y and moves to state $CELL_1(x)$.

- $ENDCELL = \alpha x.(CELL_1(x) \underbrace{\frown ENDCELL}_{\text{a new bottom cell}}) + \bar{\gamma}\$.\underbrace{\textbf{nil}}_{\text{termination}}$
  This state represents the bottom of the stack. An agent in this state can perform two actions in a non-deterministic way. First it can wait for a new value (in order to perform a right shift), then store the new data and generate a new agent witch represents the new bottom element. Note that the newly created cell ENDCELL will be able to communicate with $CELL_1(x)$ only, because they will have dedicated channels. We will explain later how this can be achieved, when giving the exact definition of the operation $\frown$. Alternatively, the agent can send the special character \$ to the left cell, provided it is able to receive this character. If so, than the left cell is empty and after receiving the \$ character it becomes the new ENDCELL. Then the present agent terminates.

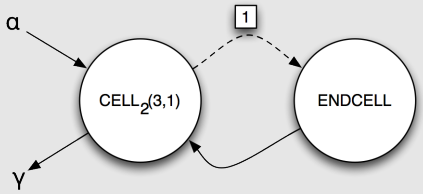
Now we will show how the stack works. Let us start from an empty stack. We have only one cell in the state ENDCELL, whose channels β and δ are made private, written $ENDCELL\backslash\beta\backslash\delta$, because on the "right" side there will be no communication. We perform a push operation in order to fill the stack with a new value. So we send the new value (1 in this case) through the channel α of the cell.



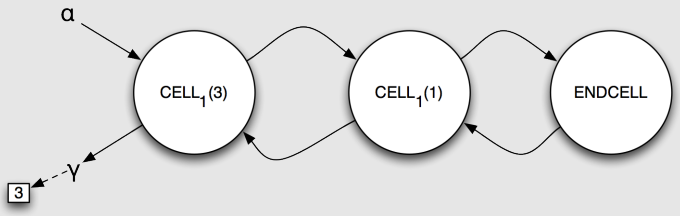
Once the cell receives the new value it generates a new bottom for the stack and changes its state to $CELL_1(1)$ *storing the new value. The result of this operation is the following:*



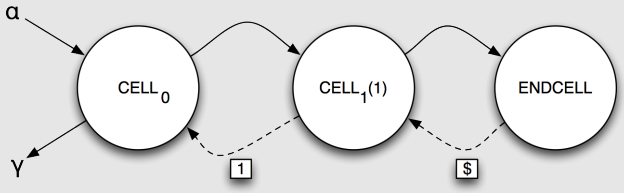
When the stack is stabilized we perform another push, this time with value 3*. In this case the first cell changes its state to* $CELL_2(3,1)$ *in order to perform a right shift.*



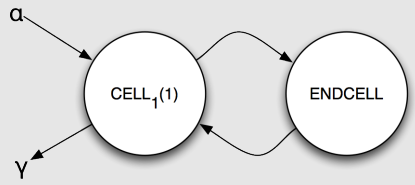
Then, when the second cell receives the value 1 *on his* α' *channel changing its state to* $CELL_1(1)$*, the first cell can stabilize itself on the state* $CELL_1(3)$*. Now we perform a pop operation, which will return the last value pushed into the stack (i.e.* 3*).*



In order to do this we read the value 3 *from the channel* γ *of the first cell. In this case the first cell changes its state to* $CELL_0$*, waiting for a value through the channel* δ*.*



When the second cell become aware of the reading performed by the first cell, it changes its state to $CELL_0$*, and reads the value sent from the third cell. Then, since the received value from ENDCELL is* \$*, it changes its state to ENDCELL. Finally, since a reading operation on* γ'' *have been performed by the second cell, the third cell reduces to* **nil**. *The situation reached after the stabilization of the system is the following:*



The above example shows that processes can synchronize in pairs, by performing dual (input/output) operations. In the following we will present a *pure* version of CCS, where we abstract away from the values communicated on channels.

10.1. Syntax of CCS

The CCS process algebra was introduced by Robin Milner in the early eighties. When presenting the syntax of CCS we will use the following conventions:

$\Delta ::= \alpha, \beta, ...$	Channels and (by coercion) input actions on channels
$\overline{\Delta} ::= \overline{\alpha}, \overline{\beta}, ...$ with $\overline{\Delta} \cap \Delta = \varnothing$	Output actions on channels
$\Lambda ::= \Delta \cup \overline{\Delta}$	Observable actions
$\tau \notin \Lambda$	Unobservable action

We extend the "bar" operation to all the elements in Λ by letting $\overline{\overline{\alpha}} = \alpha$ for all $\alpha \in \Delta$. As we have seen in the example, pairs of dual actions (e.g., α and $\overline{\alpha}$) are used to synchronize two processes. The unobservable action τ denotes a special action that is internal to some agent and that cannot be used to synchronize. Moreover we will use the following convention:

$\mu \in \Lambda \cup \{\tau\}$	generic action
$\lambda \in \Lambda$	generic channel
$\overline{\lambda} \in \Lambda$	generic dual channel

Now we are ready to present the syntax of CCS.

$$p, q ::= x \quad | \quad \mathbf{nil} \quad | \quad \mu.p \quad | \quad p\backslash\alpha \quad | \quad p[\phi] \quad | \quad p + q \quad | \quad p \mid q \quad | \quad \mathbf{rec}\ x.p$$

x represents a process name;

nil is the empty (*inactive*) process;

$\mu.p$ is a process p *prefixed* by the action μ;

$p\backslash\alpha$ is a *restricted* process; it allows to make the channel α private to p;

$p[\phi]$ is a process that behaves like the process obtained from p by applying to it the permutation (a bijective substitution) ϕ of its channel names. However this operation is part of the syntax and $p[\phi]$ is syntactically different than p with the substitution performed on it. Notice that $\phi(\overline{\lambda}) = \overline{\phi(\lambda)}$ and $\phi(\tau) = \tau$;

$p + q$ is a process that can choose non-deterministically to execute either the process p or q;

$p|q$ is the process obtained as the parallel composition of p and q; the actions of p and q can be interleaved and also synchronized;

rec $x.p$ is a recursively defined process.

As usual we will consider the closed terms of this language, i.e., the processes whose process names x are all bound by recursive definitions. We name $\mathscr{P}$ the set of closed CCS processes.

10.2. Operational Semantics of CCS

Definition 10.2 (Labelled Transition System (LTS))
A labelled transition system is a triple $(P, L, \longrightarrow)$, where P is the set of states of the system, L is the set of labels and $\longrightarrow \subseteq P \times L \times P$ is the transition relation. We write $p_1 \xrightarrow{l} p_2$ for $(p_1, l, p_2) \in \longrightarrow$.

The operational semantics of CCS is defined by a suitable LTS whose states are CCS (closed) processes and whose transitions are labelled by actions in $\Lambda \cup \{\tau\}$. Formally, the LTS is given by $(\mathscr{P}, \Lambda \cup \{\tau\}, \longrightarrow)$, where the transition relation $\longrightarrow$ is the least one generated by a set of inference rules. The LTS is thus defined by a rule system whose formulas take the form $p_1 \xrightarrow{\mu} p_2$ meaning that the process p_1 can perform the action μ and reduce to p_2.

While the LTS is the same for all CCS closed terms, starting from a CCS closed term p and using the rules we can define the LTS which represents the operational behaviour of p by considering only processes that are reachable from the state p. Although a term can be the parallel composition of many processes, its operational semantics is represented by a single global state in the LTS. Therefore concurrency and interaction between cooperating agents are not adequately represented in our CCS semantics. Now we introduce the inference rules for CCS:

$$\text{(Act)} \quad \frac{}{\mu.p \xrightarrow{\mu} p}$$

There is only one axiom in the rule system, related to the action prefix operator. It states that the process $\mu.p$ can perform the action μ and reduce to p. For example, we have a transition

$$\alpha.\,\mathbf{nil} \xrightarrow{\alpha} \mathbf{nil}$$

$$\text{(Res)} \quad \frac{p \xrightarrow{\mu} q}{p\backslash\alpha \xrightarrow{\mu} q\backslash\alpha}\ \mu \neq \alpha, \bar{\alpha}$$

If the process is executed under a restriction, then it can perform only actions that do not involve the restricted name. Note that this restriction does not affect the communication internal to the processes, i.e., when $\mu = \tau$ the move cannot be blocked by the restriction.

$$\text{(Rel)} \quad \frac{p \xrightarrow{\mu} q}{p[\phi] \xrightarrow{\phi(\mu)} q[\phi]}$$

For ϕ a permutation of channel names, if p can evolve to q by performing μ, then $p[\phi]$ can evolve to $q[\phi]$ by performing $\phi(\mu)$, i.e., the action μ renamed according to ϕ. We remind that the unobservable action cannot be renamed, i.e., $\phi(\tau) = \tau$ for any ϕ.

$$\text{(Sum)} \quad \frac{p \xrightarrow{\mu} p'}{p + q \xrightarrow{\mu} p'} \qquad \frac{q \xrightarrow{\mu} q'}{p + q \xrightarrow{\mu} q'}$$

This pair of rules deals with non-deterministic choice: process $p + q$ can choose non-deterministically to behave like either process p or q. Moreover note that the choice can be performed only during communication, so in order to discard, e.g., process q, process p must be capable to perform an action μ. For example, for $\phi(\alpha) = \gamma$ and $\phi(\beta) = \beta$ we have

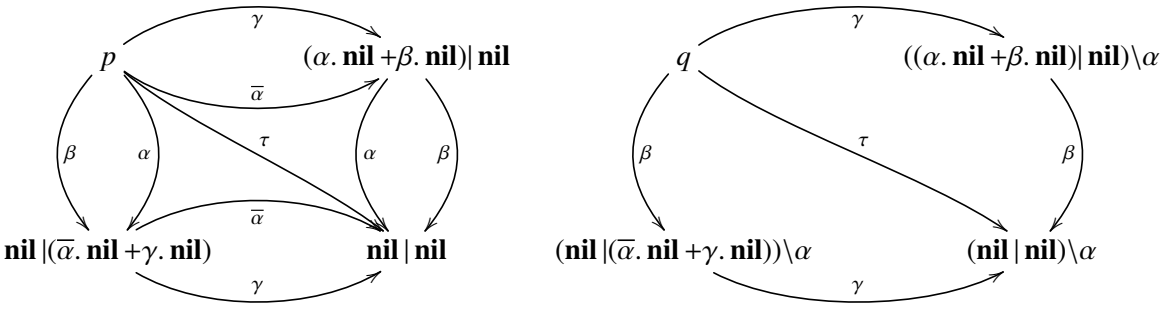
$$(\alpha.\,\mathbf{nil} + \bar{\beta}.\,\mathbf{nil})[\phi]\backslash\alpha \xrightarrow{\gamma} \mathbf{nil} \quad \text{and} \quad (\alpha.\,\mathbf{nil} + \bar{\beta}.\,\mathbf{nil})[\phi]\backslash\alpha \xrightarrow{\bar{\beta}} \mathbf{nil}[\phi]\backslash\alpha$$

$$\text{(Com)} \qquad \frac{p \xrightarrow{\mu} p'}{p|q \xrightarrow{\mu} p'|q} \qquad \frac{q \xrightarrow{\mu} q'}{p|q \xrightarrow{\mu} p|q'}$$

Also in the case of parallel composition some form of non-determinism appears. But unlike the previous case, here non-determinism is needed to simulate the parallel behaviour of the system: in the previous rule non-determinism was a characteristic of the modelled system, in this case it is a characteristic of the semantic style that allows p and q to interleave their actions in $p|q$.

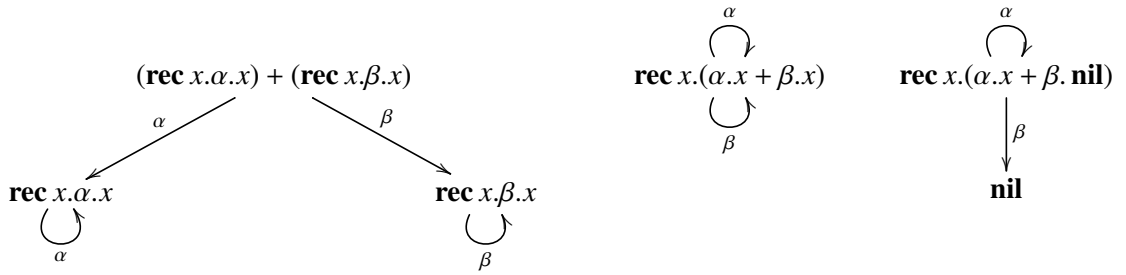
$$\frac{p_1 \xrightarrow{\lambda} p_2 \quad q_1 \xrightarrow{\overline{\lambda}} q_2}{p_1|q_1 \xrightarrow{\tau} p_2|q_2}$$

There is a third rule for parallel composition, which allows processes to perform internal synchronizations. The processes p_1 and p_2 communicate by using the channel λ, which is hidden after the synchronization by using the action τ. In general, if p_1 and p_2 can perform α and $\overline{\alpha}$, respectively, then their parallel composition can perform α, $\overline{\alpha}$ or τ. When parallel composition is used in combination with the restriction operator, like in $(p_1|p_2)\backslash\alpha$, then we can force synchronization on α. For example, see below the LTSs for the processes $p = (\alpha.\,\mathbf{nil} + \beta.\,\mathbf{nil})|(\overline{\alpha}.\,\mathbf{nil} + \gamma.\,\mathbf{nil})$ and $q = p\backslash\alpha$:



$$\text{(Rec)} \qquad \frac{p[\mathbf{rec}\,x.p/x] \xrightarrow{\mu} q}{\mathbf{rec}\,x.p \xrightarrow{\mu} q}$$

The semantics of recursion is similar to the one we have presented for HOFL: to see which moves $\mathbf{rec}\,x.p$ can perform we inspect the process $p[\mathbf{rec}\,x.p/x]$ obtained from p by replacing all free occurrences of the process name x with its full recursive definition $\mathbf{rec}\,x.p$. For example, the possible transitions of the recursive process $\mathbf{rec}\,x.\alpha.x$ are the same ones of $(\alpha.x)[\mathbf{rec}\,x.\alpha.x/x] = \alpha.\,\mathbf{rec}\,x.\alpha.x$, i.e., there is exactly one transition $\mathbf{rec}\,x.\alpha.x \xrightarrow{\alpha} \mathbf{rec}\,x.\alpha.x$. It is interesting to compare the LTSs for the processes $(\mathbf{rec}\,x.\alpha.x) + (\mathbf{rec}\,x.\beta.x)$, $\mathbf{rec}\,x.(\alpha.x + \beta.x)$ and $\mathbf{rec}\,x.(\alpha.x + \beta.\,\mathbf{nil})$, they are shown below:



In the first case either a sequence of α actions or a sequence of β actions is executed. In the second case, any sequence of α and β actions is allowed. Finally, in the last case only sequences of α actions, possibly concluded by a β action, are considered.

We will restrict our attention to the class of *guarded agents*, namely agents in which in case of recursive terms of the form $\mathbf{rec}\ x.p$, each free occurrence of x in p occurs under an action prefix (like in the examples above). This allows us to exclude terms like $\mathbf{rec}\ x.(x \mid p)$ which can lead (in one step) to an unbounded number of parallel repetitions of the same agent, making the LTS infinitely branching (see Example 10.19).

Example 10.3 (Derivation)
Let us show an example of the use of the derivation rules which we have just introduced. Take the following CCS term:

$$(((\mathbf{rec}\ x.\ \alpha.x + \beta.x) \mid (\mathbf{rec}\ x.\ \alpha.x + \gamma.x)) \mid \mathbf{rec}\ x.\ \bar{\alpha}.x)\backslash\alpha$$

First, let us focus on the behaviour of the (deterministic) agent $\mathbf{rec}\ x.\ \bar{\alpha}.x$.

$$\begin{aligned} \mathbf{rec}\ x.\ \bar{\alpha}.x &\xrightarrow{\bar{\alpha}} q && \nwarrow_{Rec} \\ \bar{\alpha}.(\mathbf{rec}\ x.\ \bar{\alpha}.x) &\xrightarrow{\bar{\alpha}} q && \nwarrow_{Act,\ q=\mathbf{rec}\ x.\ \bar{\alpha}.x} \end{aligned}$$

□

Thus:

$$\mathbf{rec}\ x.\ \bar{\alpha}.x \xrightarrow{\bar{\alpha}} \mathbf{rec}\ x.\ \bar{\alpha}.x$$

There are no other rules applicable during the above derivation; thus, the LTS associated with $\mathbf{rec}\ x.\ \bar{\alpha}.x$ *consists of a single state and one looping arrow with label* $\bar{\alpha}$. *Correspondingly, the agent is able to perform the action* $\bar{\alpha}$ *indefinitely. However, when embedded in the larger system above, then the action* $\bar{\alpha}$ *is blocked by the topmost restriction* $\_\ \backslash\alpha$. *Therefore, the only opportunity for* $\mathbf{rec}\ x.\ \bar{\alpha}.x$ *to act is by synchronizing on channel* α *with either one or the other of the two non-deterministic agents* $\mathbf{rec}\ x.\ \alpha.x+\beta.x$ *and* $\mathbf{rec}\ x.\ \alpha.x+\gamma.x$. *In fact the synchronization produces an action* τ *which cannot be blocked by* $\_\ \backslash\alpha$. *Note that each of the two non-deterministic agents is also available to interact with some external agent on another non-restricted channel, respectively* β *or* γ.

By using the rules of the operational semantics of CCS we have, e.g.:

$$\begin{aligned} (((\mathbf{rec}\ x.\ \alpha.x + \beta.x) \mid (\mathbf{rec}\ x.\ \alpha.x + \gamma.x)) \mid \mathbf{rec}\ x.\ \bar{\alpha}.x) \setminus \alpha &\xrightarrow{\mu} q && \nwarrow_{Res,\ q=q'\backslash\alpha} \\ ((\mathbf{rec}\ x.\ \alpha.x + \beta.x) \mid (\mathbf{rec}\ x.\ \alpha.x + \gamma.x)) \mid \mathbf{rec}\ x.\ \bar{\alpha}.x &\xrightarrow{\mu} q', \mu \neq \alpha, \overline{\alpha} && \nwarrow_{Com\ 3rd\ rule,\ \mu=\tau, q'=q_1 \mid q_2} \\ (\mathbf{rec}\ x.\ \alpha.x + \beta.x) \mid \mathbf{rec}\ x.\ \alpha.x + \gamma.x \xrightarrow{\lambda} q_1, \quad \mathbf{rec}\ x.\ \bar{\alpha}.x &\xrightarrow{\bar{\lambda}} q_2 && \nwarrow_{Com\ 2nd\ rule,\ q_1=(\mathbf{rec}\ x.\ \alpha.x+\beta.x) \mid q_3} \\ \mathbf{rec}\ x.\ \alpha.x + \gamma.x \xrightarrow{\lambda} q_3, \quad \mathbf{rec}\ x.\ \bar{\alpha}.x &\xrightarrow{\bar{\lambda}} q_2 && \nwarrow_{Rec} \\ \alpha.(\mathbf{rec}\ x.\ \alpha.x + \gamma.x) + \gamma.(\mathbf{rec}\ x.\ \alpha.x + \gamma.x) \xrightarrow{\lambda} q_3, \quad \mathbf{rec}\ x.\ \bar{\alpha}.x &\xrightarrow{\bar{\lambda}} q_2 && \nwarrow_{Sum} \\ \alpha.(\mathbf{rec}\ x.\ \alpha.x + \gamma.x) \xrightarrow{\lambda} q_3, \quad \mathbf{rec}\ x.\ \bar{\alpha}.x &\xrightarrow{\bar{\lambda}} q_2 && \nwarrow_{Act,\ q_3=\mathbf{rec}\ x.\ \alpha.x+\gamma.x,\ \lambda=\alpha} \\ \mathbf{rec}\ x.\ \bar{\alpha}.x &\xrightarrow{\bar{\alpha}} q_2 && \nwarrow_{Rec} \\ \bar{\alpha}.(\mathbf{rec}\ x.\ \bar{\alpha}.x) &\xrightarrow{\bar{\alpha}} q_2 && \nwarrow_{Act,\ q_2=\mathbf{rec}\ x.\ \bar{\alpha}.x} \end{aligned}$$

□

So we have:

$$\begin{aligned} q_2 &= \mathbf{rec}\ x.\ \bar{\alpha}.x \\ q_3 &= \mathbf{rec}\ x.\ \alpha.x + \gamma.x \\ q_1 &= (\mathbf{rec}\ x.\ \alpha.x + \beta.x) \mid q_3 = (\mathbf{rec}\ x.\ \alpha.x + \beta.x) \mid \mathbf{rec}\ x.\ \alpha.x + \gamma.x \\ q' &= q_1 \mid q_2 = ((\mathbf{rec}\ x.\ \alpha.x + \beta.x) \mid (\mathbf{rec}\ x.\ \alpha.x + \gamma.x)) \mid \mathbf{rec}\ x.\ \bar{\alpha}.x \\ q &= q'\backslash\alpha = (((\mathbf{rec}\ x.\ \alpha.x + \beta.x) \mid (\mathbf{rec}\ x.\ \alpha.x + \gamma.x)) \mid \mathbf{rec}\ x.\ \bar{\alpha}.x)\backslash\alpha \\ \mu &= \tau \end{aligned}$$

and thus:

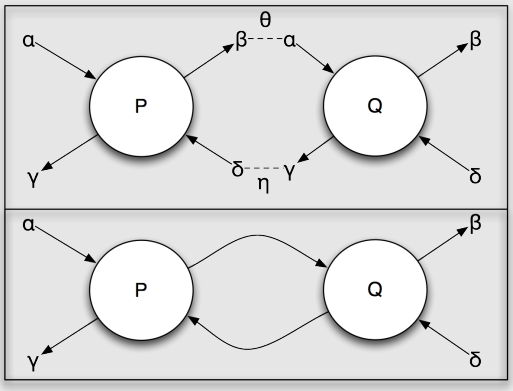
$$(((\mathbf{rec}\ x.\ \alpha.x+\beta.x) \mid (\mathbf{rec}\ x.\ \alpha.x+\gamma.x)) \mid \mathbf{rec}\ x.\ \bar{\alpha}.x)\backslash\alpha \xrightarrow{\tau} (((\mathbf{rec}\ x.\ \alpha.x+\beta.x) \mid (\mathbf{rec}\ x.\ \alpha.x+\gamma.x)) \mid \mathbf{rec}\ x.\ \bar{\alpha}.x)\backslash\alpha$$

Note that during the derivation we had to choose several times between different rules which could be applied; while in general it may happen that wrong choices can lead to dead ends, our choices have been made so to complete the derivation satisfactorily, avoiding any backtracking.

Example 10.4 (Dynamic stack: concatenation operator)
Let us consider again the dynamic stack example by formalizing in CCS the concatenation operator:

$$P \frown Q = (P[^{\vartheta}/_{\beta},^{\eta}/_{\delta}] \mid Q[^{\vartheta}/_{\alpha},^{\eta}/_{\gamma}])\backslash\vartheta\backslash\eta$$

where ϑ and η are two new hidden channels shared by the processes. Note that the locality of these names allows to avoid conflicts between channel's names. For example, on one hand, messages sent on β by P will be redirected to ϑ and must be received by Q which views ϑ as α. On the other hand, messages sent on β by Q are not redirected to ϑ and will appear as message sent on β by the whole process $P \frown Q$.



10.2.1. CCS with value passing

The dynamic stack example considers input/output operations where values can be received/transmitted. This would correspond to extend the syntax of processes to allow action prefixes like $\alpha(x).p$, where p can use the value x received on channel α and $\overline{\alpha}v.p$. Assuming a set of possible values V is fixed, the corresponding operational semantics rules are:

$$\text{(In)}\quad \frac{v \in V}{\alpha(x).p \xrightarrow{\alpha v} p[v/x]} \qquad \text{(Out)}\quad \frac{}{\overline{\alpha}v.p \xrightarrow{\overline{\alpha}v} p}$$

However, when the set V is finite, we can encode the behaviour of $\alpha(x).p$ and $\overline{\alpha}v.p$ just by introducing as many copies α_v of each channel α as the values $v \in V$. If $V = \{v_1, ..., v_n\}$ then:

- an output $\overline{\alpha}v.p$ is represented by the process

$$\overline{\alpha_v}.p$$

- an input $\alpha(x).p$ is represented by the process

$$\alpha_{v_1}.p[v_1/x] + \alpha_{v_2}.p[v_2/x] + ... + \alpha_{v_n}.p[v_n/x]$$

We can also represent quite easily an input followed by a test (for equality) on the received value, like the one used in the encoding of $CELL_0$ in the dynamic stack example.

- a process $\alpha(x).$ **if** $x = v_i$ **then** p **else** q is represented by the process

$$\alpha_{v_1}.q[v_1/x] + ... + \alpha_{v_{i-1}}.q[v_{i-1}/x] + \alpha_{v_i}.p[v_i/x] + \alpha_{v_{i+1}}.q[v_{i+1}/x] + ... + \alpha_{v_n}.q[v_n/x]$$

Example 10.5
Suppose that $V = \{true, false\}$ is the set of booleans. Then a process that waits to receive true on the channel α before executing p, can be written as

$$\mathbf{rec}\ x.\ (\alpha_{false}.x + \alpha_{true}.p)$$

10.2.2. Recursive declarations and the recursive operator

In the dynamic stack example, we have used recursive declarations, one for each possible state of the cell. They can be expressed in CCS using the recursion operator **rec**. In general, suppose we are given a series of recursive declarations, like:

$$\begin{aligned} X_1 &\stackrel{\text{def}}{=} P_1 \\ X_2 &\stackrel{\text{def}}{=} P_2 \\ &\cdots \\ X_n &\stackrel{\text{def}}{=} P_n \end{aligned}$$

where the symbols $X_1, X_2, ..., X_n$ can appear in each of $P_1, P_2, ..., P_n$. For any $i \in \{1, ..., n\}$, let $Q_i = \mathbf{rec}\, X_i.\ P_i$ be the process where all occurrences of X_i in P_i are bound by the recursive operator (while the occurrences of X_j are not bound if $i \neq j$). We can then let

$$\begin{aligned} R_n &= Q_n \\ R_{n-1} &= Q_{n-1}[R_n/X_n] \\ &\cdots \\ R_i &= Q_i[R_n/X_n]...[R_{i+1}/X_{i+1}] \\ &\cdots \\ R_1 &= Q_1[R_n/X_n]...[R_2/X_2] \end{aligned}$$

where $Q[R/X]$ denotes the syntactic replacement of X by R in Q, so that in R_i all occurrences of X_j occur under a recursive operator $\mathbf{rec}\, X_j$ if $j \geq i$. Then R_1 is a (closed) CCS process that corresponds to X_1. If we switch the order in which the recursive declarations are listed, the same procedure can be applied to find CCS processes that correspond to the other symbols $X_2, ..., X_n$.

Example 10.6
For example, suppose we are given the declarations:

$$\begin{aligned} X_1 &\stackrel{\text{def}}{=} \alpha.X_2 \\ X_2 &\stackrel{\text{def}}{=} \beta.X_1 + \gamma.X_3 \\ X_3 &\stackrel{\text{def}}{=} \delta.X_2 \end{aligned}$$

Then we have

$$\begin{aligned} Q_1 &= \mathbf{rec}\, X_1.\ \alpha.X_2 \\ Q_2 &= \mathbf{rec}\, X_2.\ (\beta.X_1 + \gamma.X_3) \\ Q_3 &= \mathbf{rec}\, X_3.\ \delta.X_2 \end{aligned}$$

From which we derive

$$\begin{aligned}
R_3 &= Q_3 \\
&= \mathbf{rec}\, X_3.\ \delta.X_2 \\
R_2 &= Q_2[R_3/X_3] \\
&= \mathbf{rec}\, X_2.\ (\beta.X_1 + \gamma.X_3)[R_3/X_3] \\
&= \mathbf{rec}\, X_2.\ (\beta.X_1 + \gamma.\, \mathbf{rec}\, X_3.\ \delta.X_2) \\
R_1 &= Q_1[R_3/X_3][R_2/X_2] \\
&= \mathbf{rec}\, X_1.\ \alpha.X_2[R_3/X_3][R_2/X_2] \\
&= \mathbf{rec}\, X_1.\ \alpha.X_2[R_2/X_2] \\
&= \mathbf{rec}\, X_1.\ \alpha.\, \mathbf{rec}\, X_2.\ (\beta.X_1 + \gamma.\, \mathbf{rec}\, X_3.\ \delta.X_2)
\end{aligned}$$

If instead we want to derive a CCS process R'_2 that corresponds to X_2 we can let

$$\begin{aligned}
R'_3 &= Q_3 \\
&= \mathbf{rec}\, X_3.\ \delta.X_2 \\
R'_1 &= Q_1[R'_3/X_3] \\
&= \mathbf{rec}\, X_1.\ \alpha.X_2[R'_3/X_3] \\
&= \mathbf{rec}\, X_1.\ \alpha.X_2 \\
R'_2 &= Q_2[R'_3/X_1][R'_1/X_1] \\
&= \mathbf{rec}\, X_2.\ (\beta.X_1 + \gamma.X_3)[R'_3/X_3][R'_1/X_1] \\
&= \mathbf{rec}\, X_2.\ (\beta.X_1 + \gamma.\, \mathbf{rec}\, X_3.\ \delta.X_2)[R'_1/X_1] \\
&= \mathbf{rec}\, X_2.\ (\beta.(\mathbf{rec}\, X_1.\ \alpha.X_2) + \gamma.\, \mathbf{rec}\, X_3.\ \delta.X_2)
\end{aligned}$$

Similarly, for X_3 we let:

$$\begin{aligned}
R''_2 &= Q_2 \\
&= \mathbf{rec}\, X_2.\ (\beta.X_1 + \gamma.X_3) \\
R''_1 &= Q_1[R''_2/X_2] \\
&= \mathbf{rec}\, X_1.\ \alpha.X_2[R''_2/X_2] \\
&= \mathbf{rec}\, X_1.\ \alpha.\, \mathbf{rec}\, X_2.\ (\beta.X_1 + \gamma.X_3) \\
R''_3 &= Q_3[R''_2/X_2][R''_1/X_1] \\
&= \mathbf{rec}\, X_3.\ \delta.X_2[R''_2/X_2][R''_1/X_1] \\
&= \mathbf{rec}\, X_3.\ \delta.\, \mathbf{rec}\, X_2.\ (\beta.X_1 + \gamma.X_3)[R''_1/X_1] \\
&= \mathbf{rec}\, X_3.\ \delta.\, \mathbf{rec}\, X_2.\ (\beta.(\mathbf{rec}\, X_1.\ \alpha.\, \mathbf{rec}\, X_2.\ (\beta.X_1 + \gamma.X_3)) + \gamma.X_3)
\end{aligned}$$

10.3. Abstract Semantics of CCS

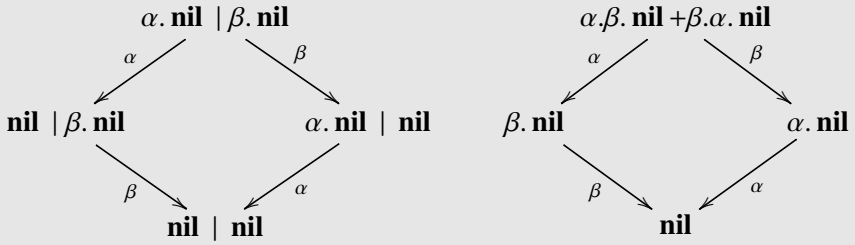
As we saw each CCS agent can be represented by an LTS, i.e., by a labelled graph. It is easy to see that such operational semantics is much more concrete and detailed than the semantics studied for IMP and HOFL. For example, since the states of the LTS are named by agents it is evident that two syntactically different processes like $p|q$ and $q|p$ are associated with different graphs, even if intuitively one would expect that both exhibit the same behaviour. Analogously for $p|\,\mathbf{nil}$ or $p + \mathbf{nil}$ and p. Thus it is important to find a good notion of equivalence, able to provide a more abstract semantics for CCS. As it happens for the denotational semantics of IMP and HOFL, an abstract semantics defined up to equivalence should abstract from the way agents execute, focusing on their external visible behaviours.

In this section we first show that neither graph isomorphism nor trace semantics are fully satisfactorily abstract semantics to capture the features of communicating systems represented in CCS. Next, we introduce a more satisfactory semantics of CCS by defining a relation, called *bisimilarity*, that captures the ability of processes to simulate each other. Finally, we discuss some positive and negative aspects of bisimilarity and present some possible alternatives.

10.3.1. Graph Isomorphism

It is quite obvious to think that two agents must be considered as equivalent if their (LTSs) graphs are isomorphic. Recall that two graphs are said to be isomorphic if there exists a bijection f between the nodes of the graphs that preserves the structure of the graphs, i.e., such that $v \xrightarrow{\alpha} v'$ iff $f(v) \xrightarrow{\alpha} f(v')$.

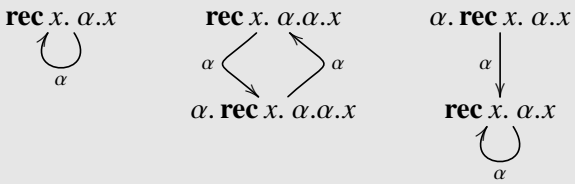
Example 10.7 (Isomorphic agents)
Let us consider the agents $\alpha.\,\mathbf{nil} \mid \beta.\,\mathbf{nil}$ *and* $\alpha.\beta.\,\mathbf{nil} + \beta.\alpha.\,\mathbf{nil}$. *Their LTSs are as follows:*



The two graphs are isomorphic, thus the two agents should be considered as equivalent. This result is surprising, since they have a rather different structure. In fact, the example shows that concurrency can be reduced to non-determinism by graph isomorphism. This is due to the interleaving of the actions performed by processes that are composed in parallel, which is a characteristic of the semantics which we have presented.

This approach is very simple and natural but still leads to semantics that is too concrete, i.e., graph isomorphism still distinguishes too much. We show this fact in the following examples.

Example 10.8
Let us consider the agents $\mathbf{rec}\ x.\ \alpha.x$, $\mathbf{rec}\ x.\ \alpha.\alpha.x$ *and* $\alpha.\,\mathbf{rec}\ x.\ \alpha.x$ *Their LTSs are as follows:*



The three graphs are not isomorphic, but it is hardly possible to distinguish between the agents according to their behaviour: they all are able to execute any sequence of α.

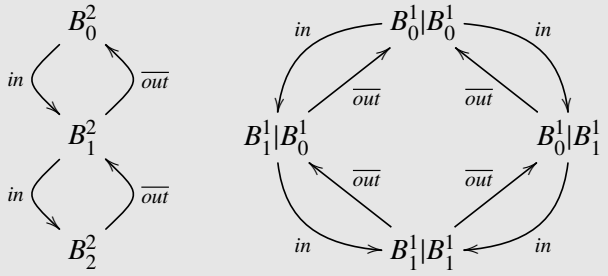
Example 10.9
Let us denote by B^n_k *a buffer of capacity n of which k positions are busy. For example, for representing a buffer of capacity* 1 *in CCS one could let (using recursive definitions):*

$$B^1_0 \stackrel{\text{def}}{=} in.B^1_1$$
$$B^1_1 \stackrel{\text{def}}{=} \overline{out}.B^1_0$$

Analogously, for a buffer of capacity 2*, one could let:*

$$\begin{aligned} B_0^2 &\stackrel{\text{def}}{=} in.B_1^2 \\ B_1^2 &\stackrel{\text{def}}{=} \overline{out}.B_0^2 + in.B_2^2 \\ B_2^2 &\stackrel{\text{def}}{=} \overline{out}.B_1^2 \end{aligned}$$

Another possibility for obtaining an (empty) buffer of capacity 2 *is to use two (empty) buffers of capacity* 1 *composed in parallel:* $B_0^1|B_0^1$*. However the LTSs of* B_0^2 *and* $B_0^1|B_0^1$ *are quite different:*



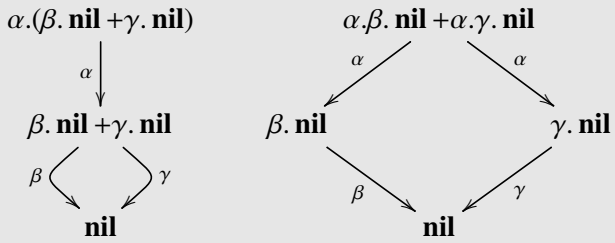
10.3.2. Trace Equivalence

A second approach, trace equivalence, observes the set of traces of an agent, namely the set of sequences of actions labelling any path in its graph. Trace equivalence is strictly coarser than equivalence based on graph isomorphism, since isomorphic graphs have the same traces. Conversely, Example 10.5 shows two agents which are trace equivalent but whose graphs are not isomorphic. In fact, trace equivalence is too coarse: the following example shows that trace equivalence is not able to capture the choice points within agent behaviour.

Example 10.10
Let us consider the following agents:

$$p = \alpha.(\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil}) \qquad q = \alpha.\beta.\,\mathbf{nil} + \alpha.\gamma.\,\mathbf{nil}$$

Their LTSs are as follows:



These two graphs are trace equivalent: the trace sets are both $\{\alpha\beta, \alpha\gamma\}$*. However the agents do not behave in the same way if we regard the choices they make. In the second agent the choice between* β *and* γ *is made during the first step, by selecting one of the two possible* α*. In the first agent, on the contrary, the same choice is made in a second time, after the execution of the unique* α *action.*

The difference is evident if we consider, e.g., that an agent $\bar{\alpha}.\bar{\beta}.\,\mathbf{nil}$ *may be running in parallel, with actions* α*,* β *and* γ *restricted on top: the agent* p *is always able to carry out the complete interaction with* $\bar{\alpha}.\bar{\beta}.\,\mathbf{nil}$*, because after the synchronization on* α *is ready to synchronize on* β*; vice versa, the agent* q *is only able to carry out the complete interaction with* $\bar{\alpha}.\bar{\beta}.\,\mathbf{nil}$ *if the left choice is performed at the time of the first interaction on* α*, as otherwise* $\gamma.\,\mathbf{nil}$ *and* $\bar{\beta}.\,\mathbf{nil}$ *cannot interact. Formally, if we consider the context* $C(\_) = (\_ \mid \overline{\alpha}.\overline{\beta}.nil) \setminus \alpha \setminus \beta \setminus \gamma$ *we have that* $C(q)$ *can deadlock, while* $C(p)$ *cannot. Figure out how embarrassing could be the difference if* α *would mean for a computer to ask if a file should be deleted, and*

β, γ were the user yes/no answer: p would behave as expected, while q could decide to delete the file in the first place, and then deadlock if the the user decides otherwise. See also Example 10.24.

Given all the above, we can argue that neither graph isomorphism nor trace equivalence are good candidates for our behavioural equivalence relation. Still, it is obvious that: 1) isomorphic agents must be retained as equivalent; 2) equivalent agents must be trace equivalent. Thus, our candidate equivalence relation must be situated in between graph isomorphism and trace equivalence.

10.3.3. Bisimilarity

In this section we introduce a class of relations between agents called *bisimulations* and we construct a behavioural equivalence relation between agents called *bisimilarity* as the largest bisimulation. This equivalence relation is shown to be the one we were looking for, namely the one that identifies only those agents which intuitively have the same behaviour.

Let us start with an example which illustrates how bisimilarity works.

Example 10.11 (Game Theory)
In this example we use game theory in order to show that the agents of the example 10.10 are not behaviourally equivalent. In our example game theory is used in order to prove that a system verifies or not a property. We can imagine two player called Alice and Bob, the goal of Alice is to prove that the system has not the property. Bob, on the contrary, wants to show that the system satisfies the property. The game starts and at each turn each player can make a move in order to reach his/her goal. At the end of the game if Alice wins this means that the system does not satisfy the property. If the winner is Bob, instead, the system satisfies the property.

We apply this pattern to the states of LTSs which describe CCS agents. Let us take two states p and q of an LTS. Alice would like to show that p is not behavioural equivalent to q, Bob on the other hand would like to show that p and q have the same behaviour.

Alice starts the game. At each turn of the game Alice executes (if possible) a transition of the transition system of either p or q and Bob must execute a transition with the same label but of the other agent. If Alice cannot move on both p and q, then Alice has lost, since this means that p and q are both deadlocked, and thus obviously equivalent. Alice wins if she can make a move that Bob cannot imitate; or if she has a move which, no matter which is the answer by Bob, will lead to a situation where she can make a move that Bob cannot imitate; or ... and so on for any number of moves. Bob wins if Alice has no such a (finite) strategy. Note that the game does not necessarily terminate: also in this case Bob wins, that is p and q are equivalent.

In the example 10.10, let us take $p = \alpha.(\beta.\mathbf{nil} + \gamma.\mathbf{nil})$ and $q = \alpha.\beta.\mathbf{nil} + \alpha.\gamma.\mathbf{nil}$. Alice starts by choosing p and by executing the only transition labelled α. Then, Bob can choose one of the two transitions labelled α leaving from q. Suppose that Bob chooses the left α transition (but the case where Bob chooses the right transition leads to the same result of the game). So the reached states are $\beta.\mathbf{nil} + \gamma.\mathbf{nil}$ and $\beta.\mathbf{nil}$. In the second turn Alice chooses the transition labelled γ from $\beta.\mathbf{nil} + \gamma.\mathbf{nil}$, and Bob can not simulate this execution. Since Alice has a winning, two-moves strategy, the two agents are not equivalent.

Now we define the same relation in a more formal way, as originally introduced by Robin Milner. It is important to notice that the definition applies to a generic labelled transition systems, namely a set of states P equipped with a ternary relation $\longrightarrow \subseteq P \times L \times P$, where L is a generic set of actions. As we have seen, a unique LTS is associated to CCS, where CCS agents are states and a triple (p, α, q) belongs to $\longrightarrow$ iff $p \xrightarrow{\alpha} q$ is a theorem of the operational semantics. Notice that agents with isomorphic graphs are automatically equivalent.

Definition 10.12 (Strong Bisimulation)
Let R be a binary relation on the set of states of an LTS then it is a strong bisimulation *if*

$$\forall s_1, s_2. \quad s_1 \; R \; s_2 \Rightarrow \begin{cases} \forall \alpha, s_1'. & \text{if } s_1 \xrightarrow{\alpha} s_1' \quad \text{then } \exists s_2' \text{ such that} \quad s_2 \xrightarrow{\alpha} s_2' \text{ and } s_1' \; R \; s_2' \\ \forall \alpha, s_2'. & \text{if } s_2 \xrightarrow{\alpha} s_2' \quad \text{then } \exists s_1' \text{ such that} \quad s_1 \xrightarrow{\alpha} s_1' \text{ and } s_1' \; R \; s_2' \end{cases}$$

For example it is easy to check that the identity relation $\{(p, p) \mid p \text{ is a CCS process }\}$ is a strong bisimulation, that graph isomorphism is a strong bisimulation and that the union $R_1 \cup R_2$ of two strong bisimulation relations R_1 and R_2 is also a strong bisimulation relation. Moreover, given the composition of relations defined by

$$R_1 \circ R_2 \stackrel{\text{def}}{=} \{(p, p') \mid \exists p''.p\ R_1\ p'' \wedge p''\ R_2\ p'\}$$

it can be shown that $R_1 \circ R_2$ is a strong bisimulation when R_1 and R_2 are such.

Definition 10.13 (Strong bisimilarity $\simeq$)
Let s and s' be two states of an LTS, then they are said to be bisimilar *and write $s \simeq s'$ if and only if there exists a strong bisimulation R such that $s\ R\ s'$.*

The relation $\simeq$ is called strong bisimilarity *and is defined as follows:*

$$\simeq \stackrel{\text{def}}{=} \bigcup_{R \text{ is a strong bisimulation}} R$$

Strong bisimilarity $\simeq$ is an equivalence relation on CCS processes. Below we recall the definition of equivalence relation.

Definition 10.14 (Equivalence Relation)
Let $\equiv$ be a binary relation on a set X, then we say that it is an equivalence relation *if it has the following properties:*

- $\forall x, y \in X.\ x \equiv x$ *(Reflexivity)*
- $\forall x, y, z \in X.\ x \equiv y \wedge y \equiv z \Rightarrow x \equiv z$ *(Transitivity)*
- $\forall x, y \in X.\ x \equiv y \Rightarrow y \equiv x$ *(Symmetry)*

Definition 10.15 (Equivalence Class and Quotient Set)
Given an equivalence relation $\equiv$ on X and an element x of X we call equivalence class *of x the subset $[x]$ of X defined as follows:*

$$[x] = \{y \in X \mid x \equiv y\}$$

The set $X_{/\equiv}$ containing all the equivalence classes generated by a relation $\equiv$ on the set X is called quotient set.

We omit the proof of the following theorem that is based on the above mentioned properties of strong bisimulations and on the fact that bisimilarity is a strong bisimulation.

Theorem 10.16
The bisimilarity relation $\simeq$ is an equivalence relation between CCS agents.

Now we will use the fixpoint theory, which we have introduced in the previous chapters, in order to define bisimilarity in a more effective way. Using fixpoint theory we will construct, by successive approximations, the coarsest (maximal) bisimulation between the states of an LTS, which is actually an equivalence relation.

As usual, we define the $CPO_\perp$ on which the approximation function works. The $CPO_\perp$ is defined on the set $\mathscr{P}(P \times P)$, namely the power set of the pairs of states of the LTS. As we saw in the previous chapters the pair $(\mathscr{P}(P \times P), \subseteq)$ (all the subsets of a given set, ordered by inclusion) is a $CPO_\perp$, however it is not the one which we will use. As we said we would like to start from the roughest relation, which considers all the states equivalent and, by using the fixpoint operator, to reach the relation which will identify only bisimilar agents. So we need a $CPO_\perp$ in which a set with more elements is considered smaller than one with few elements. Thus we define the order relation $R \sqsubseteq R' \Leftrightarrow R' \subseteq R$ between subsets of $\mathscr{P}(P \times P)$. The resulting $CPO_\perp$ $(\mathscr{P}(P \times P), \sqsubseteq)$ is represented in figure 10.1 .

Now we define the function $\Phi : \mathscr{P}(P \times P) \to \mathscr{P}(P \times P)$ on relations on P.

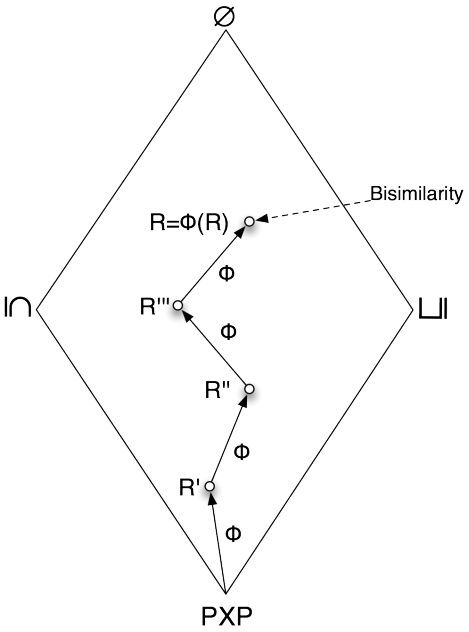


Figure 10.1.: $CPO_{\perp}$ $(\mathscr{P}(P \times P), \sqsubseteq)$

$$p\ \Phi(R)\ q \stackrel{\text{def}}{=} \begin{cases} \forall \mu, p'. \quad p \xrightarrow{\mu} p' \quad \text{implies} \quad \exists q'.\ q \xrightarrow{\mu} q' \quad \text{and} \quad p' R q' \\ \forall \mu, q'. \quad q \xrightarrow{\mu} q' \quad \text{implies} \quad \exists p'.\ p \xrightarrow{\mu} p' \quad \text{and} \quad p' R q' \end{cases}$$

Definition 10.17 (Bisimulation as fixpoint)
Let R be a relation in $\mathscr{P}(P \times P)$ then it is said to be a bisimulation *iff it is a pre-fixpoint of Φ, i.e.:*

$$\Phi(R) \sqsubseteq R \quad (\textit{or equivalently, } R \subseteq \Phi(R))$$

Theorem 10.18 (Bisimilarity as fixpoint)
The function Φ is monotone and continue, i.e.:

$$\begin{aligned} R_1 \sqsubseteq R_2 \quad &\Rightarrow \quad \Phi(R_1) \sqsubseteq \Phi(R_2) \\ \Phi(\bigsqcup_{n \in \omega} R_n) \quad &= \quad \bigsqcup_{n \in \omega} \Phi(R_n) \end{aligned}$$

Moreover, the least fixed point of Φ is the bisimilarity, namely it holds:

$$\simeq \stackrel{\text{def}}{=} \bigsqcap_{R = \Phi(R)} R = \bigsqcup_{n \in \omega} \Phi^n(P \times P)$$

We do not prove the above theorem. Note that monotonicity is obvious, since a larger R will make the conditions on $\Phi(R)$ weaker. Continuity of Φ is granted only if the LTS is finitely branching, namely if every state has a finite number of outgoing transitions. If Φ is not continuous, we still have the existence of a minimal fixpoint, but, it will not be always reachable by the ω chain of approximations.

Example 10.19 (Infinitely branching process)
Let us consider the agent $p = \mathbf{rec}\, x.\, (x \mid \alpha.\, \mathbf{nil})$. The agent p is not guarded, because the occurrence of x in the body of the recursive process is not prefixed by an action. By using the rules of the operational

semantics of CCS we have, e.g.:

$$\begin{aligned}
\mathbf{rec}\, x.\, (x \mid \alpha.\,\mathbf{nil}) &\xrightarrow{\mu} q && \nwarrow_{Rec}\\
(\mathbf{rec}\, x.\, (x \mid \alpha.\,\mathbf{nil})) \mid \alpha.\,\mathbf{nil} &\xrightarrow{\mu} q && \nwarrow_{Com\ 1st\ rule,\ q=q_1 \mid \alpha.\,\mathbf{nil}}\\
\mathbf{rec}\, x.\, (x \mid \alpha.\,\mathbf{nil}) &\xrightarrow{\mu} q_1 && \nwarrow_{Rec}\\
(\mathbf{rec}\, x.\, (x \mid \alpha.\,\mathbf{nil})) \mid \alpha.\,\mathbf{nil} &\xrightarrow{\mu} q_1 && \nwarrow_{Com\ 1st\ rule,\ q_1=q_2 \mid \alpha.\,\mathbf{nil}}\\
\mathbf{rec}\, x.\, (x \mid \alpha.\,\mathbf{nil}) &\xrightarrow{\mu} q_2 && \nwarrow_{Rec}\\
&\ldots\\
\mathbf{rec}\, x.\, (x \mid \alpha.\,\mathbf{nil}) &\xrightarrow{\mu} q_n && \nwarrow_{Rec}\\
(\mathbf{rec}\, x.\, (x \mid \alpha.\,\mathbf{nil})) \mid \alpha.\,\mathbf{nil} &\xrightarrow{\mu} q_n && \nwarrow_{Com\ 2nd\ rule,\ q_n=(\mathbf{rec}\, x.\, (x \mid \alpha.\,\mathbf{nil})) \mid q'}\\
\alpha.\,\mathbf{nil} &\xrightarrow{\mu} q' && \nwarrow_{Act,\ \mu=\alpha,q'=\mathbf{nil}}\\
&\square
\end{aligned}$$

It is then evident that for any $n \in \omega$ we have:

$$\mathbf{rec}\, x.\, (x \mid \alpha.\,\mathbf{nil}) \xrightarrow{\alpha} (\mathbf{rec}\, x.\, (x \mid \alpha.\,\mathbf{nil})) \mid \mathbf{nil} \mid \underbrace{\alpha.\,\mathbf{nil} \mid \cdots \mid \alpha.\,\mathbf{nil}}_{n}$$

The following lemma ensures that if we consider only guarded terms then the LTS is finitely branching.

Lemma 10.20
Let p be a guarded CCS term then $\{q \mid p \xrightarrow{\mu} q\}$ is a finite set.

In order to apply the fixpoint theorem to calculate the bisimilarity, we consider only states which are reachable from the states we want to compare for bisimilarity. If the number of reachable states is finite, i.e. if the system is a finite state automata, the calculation is effective, but possibly quite complex if the number of states is large.

Example 10.21 (Bisimilarity as fixpoint)
Let us consider the example 10.10 which we have already solved with game theory techniques. Now we show the fixpoint approach to the same system. Let us restrict the attention to the set of reachable states and represent the relations by showing the equivalence classes which they induce (over reachable processes) . We start with the coarsest relation, where any two processes are related (just one equivalence class):

$$R_0 = \{\ \{\alpha.(\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil})\ ,\ \alpha.\beta.\,\mathbf{nil} + \alpha.\gamma.\,\mathbf{nil}\ ,\ \beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil}\ ,\ \beta.\,\mathbf{nil}\ ,\ \gamma.\,\mathbf{nil}\ ,\ \mathbf{nil}\}\ \}$$

By applying Φ:

$$R_1 = \Phi(R_0) = \{\ \{\alpha.(\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil}), \alpha.\beta.\,\mathbf{nil} + \alpha.\gamma.\,\mathbf{nil}\}\ ,\ \{\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil}\}\ ,\ \{\beta.\,\mathbf{nil}\}\ ,\ \{\gamma.\,\mathbf{nil}\}\ ,\ \{\mathbf{nil}\}\ \}$$

$$R_2 = \Phi(R_1) = \{\ \{\alpha.(\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil})\}\ ,\ \{\alpha.\beta.\,\mathbf{nil} + \alpha.\gamma.\,\mathbf{nil}\}\ ,\ \{\beta.\,\mathbf{nil} + \gamma.\,\mathbf{nil}\}\ ,\ \{\beta.\,\mathbf{nil}\}\ ,\ \{\gamma.\,\mathbf{nil}\}\ ,\ \{\mathbf{nil}\}\ \}$$

Note that R_2 is a fixpoint, hence it is the coarsest bisimulation.

10.4. Compositionality

In this section we focus our attention on the *compositionality* aspect of the abstract semantics which we have just introduced. For an abstract semantics to be practically relevant it is important that any process used in a system can be replaced with an equivalent process without changing the semantics of the system. Since we have not used structural induction in defining the abstract semantics of CCS, no one ensures any kind of compositionality w.r.t. the possible way of constructing larger systems, i.e., w.r.t. the operators of CCS.

Definition 10.22 (Context)
A context *is a term with a gap which can be filled by inserting any other term of our language. We write* $C[\,]$ *to indicate a* context.

Definition 10.23 (Congruence)
A relation $\sim_{\mathscr{C}}$ *is said to be a* congruence *(with respect to a class of contexts) if:*

$$\forall C[\,].\ p \sim_{\mathscr{C}} q \quad \Rightarrow \quad C[p] \sim_{\mathscr{C}} C[q]$$

In order to guarantee the compositionality of CCS we must show that the bisimilarity is a congruence relation.

Let us now see an example of a relation which is not a congruence.

Example 10.24 (Trace equivalence)
Let us consider the trace equivalence relation, which we have defined in Section 10.3.2. Take the following context:

$$C[\_] = (\ \_ \mid \bar{\alpha}.\bar{\beta}.\mathbf{nil})\backslash\alpha\backslash\beta\backslash\gamma$$

Now we can fill the gaps with the following terms:

$$C[p] = (\alpha.(\beta.\mathbf{nil} + \gamma.\mathbf{nil}) \mid \bar{\alpha}.\bar{\beta}.\mathbf{nil})\backslash\alpha\backslash\beta\backslash\gamma$$

$$C[q] = ((\alpha.\beta.\mathbf{nil} + \alpha.\gamma.\mathbf{nil}) \mid \bar{\alpha}.\bar{\beta}.\mathbf{nil})\backslash\alpha\backslash\beta\backslash\gamma$$

Obviously $C[p]$ *and* $C[q]$ *generate the same set of traces, however one of the processes can "deadlock" before the interaction on* β *takes place, but not the other. The difference can be formalized if we consider the so-called* completed trace semantics.

A completed trace *of a process* p *is a sequence of actions* $\mu_1 \cdots \mu_k$ *(for* $k \geq 0$*) such that there exists a sequence of transitions*

$$p = p_0 \xrightarrow{\mu_1} p_1 \xrightarrow{\mu_2} \cdots \xrightarrow{\mu_{k-1}} p_{k-1} \xrightarrow{\mu_k} p_k \not\rightarrow$$

for some $p_1, ..., p_k$*. The completed traces of a process characterize the sequences of actions that can lead the system to a deadlocked configuration, where no further action is possible.*

The completed trace semantics of p *is the same as that of* q*, namely* $\{\ \alpha\beta\ ,\ \alpha\gamma\ \}$*. However, the completed traces of* $C[p]$ *and* $C[q]$ *are* $\{\ \tau\tau\ \}$ *and* $\{\ \tau\tau\ ,\ \tau\ \}$*, respectively. We can thus conclude that the completed trace semantics is not a congruence.*

10.4.1. Bisimilarity is Preserved by Parallel Composition

In order to show that bisimilarity is a congruence we should prove that the property holds for all the operators of CCS, since this implies that the property holds for all contexts. However we show the proof only for parallel composition, which is a quite interesting case to consider. The other cases follow by similar arguments.

Formally, we need to prove that:

$$p_1 \simeq p_2 \wedge q_1 \simeq q_2 \quad \stackrel{?}{\Longrightarrow} \quad p_1 \mid q_1 \simeq p_2 \mid q_2$$

As usual we assume the premises and we would like to prove:

$$\exists R.\ (p_1 \mid q_1)\ R\ (p_2 \mid q_2)\ \wedge\ R \subseteq \Phi(R)$$

Since $p_1 \simeq p_2$ and $q_1 \simeq q_2$ we have:

$$\begin{array}{ll} p_1\ R_1\ p_2 & \text{for some bisimulation } R_1 \\ q_1\ R_2\ q_2 & \text{for some bisimulation } R_2 \end{array}$$

Now we define a bisimulation that satisfies the requested property:

$$R \stackrel{\text{def}}{=} \{(\hat{p}_1 \mid \hat{q}_1\ ,\ \hat{p}_2 \mid \hat{q}_2) \mid \hat{p}_1\ R_1\ \hat{p}_2\ \wedge\ \hat{q}_1\ R_2\ \hat{q}_2\}$$

By definition it holds $p_1 \mid q_1 \; R \; p_2 \mid q_2$.
Now we show that R is a bisimulation ($R \subseteq \Phi(R)$):

$$P(p_1|q_1 \xrightarrow{\mu} p_1'|q_1') \stackrel{\text{def}}{=} \forall p_2, q_2.\, (p_1|q_1 \; R \; p_2|q_2 \Longrightarrow \exists p_2', q_2'.\, p_2|q_2 \xrightarrow{\mu} p_2'|q_2' \;\wedge\; p_1'|q_1' \; R \; p_2'|q_2')$$

We proceed by rule induction. There are three possible rules for parallel composition (Com). We start by considering the rule:

$$\frac{p \xrightarrow{\mu} p'}{p|q \xrightarrow{\mu} p'|q}$$

The property for this rule is the following:

$$P(p|q \xrightarrow{\mu} p'|q) \stackrel{\text{def}}{=} \forall p_2, q_2.\, (p|q \; R \; p_2|q_2 \Longrightarrow \exists p_2', q_2'.\, p_2|q_2 \xrightarrow{\mu} p_2'|q_2' \;\wedge\; p'|q \; R \; p_2'|q_2')$$

We assume that $p \xrightarrow{\mu} p'$ and that, by definition of R, $p \; R_1 \; p_2$ and $q \; R_2 \; q_2$. Then we have:

$$\exists p_2'.p_2 \xrightarrow{\mu} p_2' \;\wedge\; p' \; R_1 \; p_2'$$

By applying the first (Com) rule:

$$p_2|q_2 \xrightarrow{\mu} p_2'|q_2$$

By definition of R we conclude:

$$p'|q \; R \; p_2'|q_2$$

The proof for the second rule

$$\frac{q \xrightarrow{\mu} q'}{p|q \xrightarrow{\mu} p|q'}$$

is analogous.

Finally, we consider the third (Com) rule:

$$\frac{p \xrightarrow{\lambda} p' \quad q \xrightarrow{\bar{\lambda}} q'}{p|q \xrightarrow{\tau} p'|q'}$$

The property for this rule is the following:

$$P(p|q \xrightarrow{\tau} p'|q') \stackrel{\text{def}}{=} \forall p_2, q_2.\, (p|q \; R \; p_2|q_2 \Longrightarrow \exists p_2', q_2'.\, p_2|q_2 \xrightarrow{\tau} p_2'|q_2' \wedge p'|q' \; R \; p_2'|q_2')$$

Assuming the premise and by definition of R we have:

$$p \xrightarrow{\lambda} p' \qquad q \xrightarrow{\bar{\lambda}} q' \qquad p \; R_1 \; p_2 \qquad q \; R_2 \; q_2$$

Therefore:

$$\begin{array}{lllll} p \xrightarrow{\lambda} p' & \wedge & p \; R_1 \; p_2 & \Rightarrow & \exists p_2'.\, p_2 \xrightarrow{\lambda} p_2' \;\wedge\; p' \; R_1 \; p_2' \\ q \xrightarrow{\bar{\lambda}} q' & \wedge & q \; R_2 \; q_2 & \Rightarrow & \exists q_2'.\, q_2 \xrightarrow{\bar{\lambda}} q_2' \;\wedge\; q' \; R_2 \; q_2' \end{array}$$

By applying the third (Com) rule we obtain:

$$p_2|q_2 \xrightarrow{\tau} p_2'|q_2'$$

We conclude by definition of R:

$$p'|q' \; R \; p_2'|q_2'$$

10.5. Hennessy - Milner Logic

In this section we present a *modal logic* introduced by Matthew Hennessy and Robin Milner. Modal logic allows to express concepts as "there exists a next state such that", or "for all next states", some property holds. Typically, model checkable properties are stated as formulas in some modal logic. In particular, Hennessy-Milner modal logic is relevant for its simplicity and for its close connection with bisimilarity. As we will see, in fact, two bisimilar agents verify the same set of modal logic formulas. This fact shows that bisimilarity is at the right level of abstraction.
First of all we introduce the syntax of the *Hennessy-Milner logic* (HM-logic):

$$F ::= true \quad | \quad false \quad | \quad \wedge_{i\in I} F_i \quad | \quad \vee_{i\in I} F_i \quad | \quad \Diamond_\mu F \quad | \quad \Box_\mu F$$

We write $\mathscr{L}$ for the set of the HM-logic formulas.

The formulas of HM-logic express properties of LTS states, namely in our case of CCS agents. The meanings of the logic operators are the following:

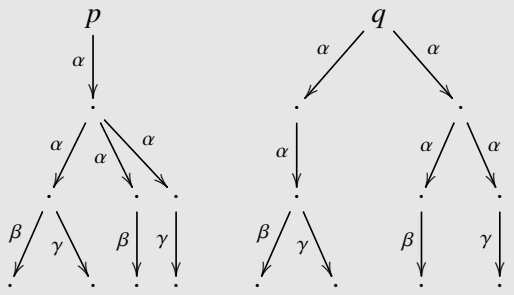
- *true*: is the formula satisfied by every agent. Notice that *true* can be considered a shorthand for an indexed conjunction $\wedge_{i\in I}$ where the set I of indexes is empty.
- $false$: is the formula never satisfied by any agent. Notice that $false$ can be considered a shorthand for an indexed disjunction $\vee i \in I$ where the set I of indexes is empty.
- $\wedge_{\forall i\in I} F_i$: is equivalent to the classic "and" operator applied to the set of formulas $\{F_i\}_{i\in I}$.
- $\vee_{\forall i\in I} F_i$: is equivalent to the classic "or" operator applied to the set of formulas $\{F_i\}_{i\in I}$.
- $\Diamond_\mu F$: it is a *modal operator*, an agent p satisfies this formula if there exists a transition from p to q labelled with μ and the formula F holds in q.
- $\Box_\mu F$: it is also a *modal operator*, an agent p satisfies this formula if for any q such that there is a transition from p to q labelled with μ the formula F holds in q.

As usual in logic satisfaction is defined as a relation $\models$ between formulas and their models, which in our case are states of an LTS.

Definition 10.25 (Satisfaction relation)
The satisfaction relation $\models \subseteq P \times \mathscr{L}$ *is defined as follows:*

$$\begin{array}{lll} p \models true & & \\ p \models \wedge_{i\in I} F_i & \textit{iff} & \forall i \in I.\ p \models F_i \\ p \models \vee_{i\in I} F_i & \textit{iff} & \exists i \in I.\ p \models F_i \qquad \forall i \in I \\ p \models \Diamond_\mu F & \textit{iff} & \exists p'.\ p \xrightarrow{\mu} p' \wedge p' \models F \\ p \models \Box_\mu F & \textit{iff} & \forall p'.\ p \xrightarrow{\mu} p' \Rightarrow p' \models F \end{array}$$

Example 10.26 (non-equivalent agents)
Let us consider two CCS agents p and q associated with the following graphs:



We would like to show a formula F which is satisfied by one of the two agents and not by the other. For

example we can take:

$$F = \Diamond_\alpha \Box_\alpha (\Diamond_\beta true \wedge \Diamond_\gamma true)$$

we have:

$$q \models F \qquad p \not\models F$$

In fact in q we can choose the left α-transition and we reach a state that satisfies $\Box_\alpha(\Diamond_\beta true \wedge \Diamond_\gamma true)$ (i.e., the (only) state reachable by an α-transition can perform both γ and β). On the contrary, the agent p does not satisfy the formula F because after the unique α-transition it is possible to take α-transitions that lead to states where either β or γ is enabled, but not both.

The HM-logic induces an obvious equivalence on CCS processes: two agents are logically equivalent if they satisfy the same set of formulas. Now we present two theorems which allow us to connect bisimilarity and modal logic. As we said this connection is very important both from theoretical and practical point of view. We start by introducing a measure over formulas to estimate the maximal number of consecutive steps that must be taken into account to check the validity of the formulas.

Definition 10.27 (Depth of a formula)
We define the depth *of a formula as follows:*

$$\begin{aligned} D(true) &= 0 \\ D(false) &= 0 \\ D(\wedge_{i\in I} F_i) &= \max(D(F_i) \mid i \in I) \\ D(\vee_{i\in I} F_i) &= \max(D(F_i) \mid i \in I) \\ D(\Diamond_\mu F) &= D(F) + 1 \\ D(\Box_\mu F) &= D(F) + 1 \end{aligned}$$

We will denote the set of logic formulas of depth k with $\mathscr{L}_k = \{F \mid D(F) = k\}$.

The first theorem ensures that if two agents are not distinguished by the k^{th} iteration of the fixpoint calculation of bisimilarity, then no formula of depth k can distinguish between the two agents, and viceversa.

Theorem 10.28
Let $\sim_k$ be defined as follows:

$$p \sim_k q \Leftrightarrow p \; \Phi^k(P \times P) \; q$$

and let p and q be two CCS agents. Then, we have:

$$p \sim_k q \quad \textit{iff} \quad \forall F \in \mathscr{L}_k.\, (p \models F) \Leftrightarrow (q \models F)$$

The second theorem generalizes the above correspondence by setting up a connection between formulas of any depth and bisimilarity. The proof is by induction on the depth of formulas.

Theorem 10.29
Let p and q two CCS agents, then we have:

$$p \simeq q \quad \textit{iff} \quad \forall F.(p \models F) \Leftrightarrow (q \models F)$$

It is worth reading this result both in the positive sense, namely bisimilar agents satisfy the same set of HM formulas; and in the negative sense, namely if two agents are not bisimilar, then there exists a formula which distinguishes between them. From a theoretical point of view these theorems show that bisimilarity distinguishes all and only those agents which are really different because they enjoy different properties. These results witness that the relation $\simeq$ is a good choice from the logical point of view.

10.6. Axioms for Strong Bisimilarity

Finally, we show that strong bisimilarity can be finitely axiomatized. First we present a theorem which allows to derive for every non recursive CCS agent a suitable normal form.

Theorem 10.30
Let p be a (non-recursive) CCS agent, then there exists a CCS agent strongly bisimilar to p built only with prefix, sum and nil.

Proof. We proceed by structural recursion. First define two binary operators $\rfloor$ and $\|$, where $\rfloor q$ means that q does not perform any action, and $p_1 \| p_2$ means that p_1 and p_2 must perform a synchronization. This corresponds to say that the operational semantics rules for $p \rfloor q$ and $p \| q$ are:

$$\frac{p \xrightarrow{\mu} p'}{p \rfloor q \xrightarrow{\mu} p' | q} \qquad \frac{p \xrightarrow{\lambda} p' \quad q \xrightarrow{\bar{\lambda}} q'}{p \| q \xrightarrow{\tau} p' | q'}$$

We show how to decompose the parallel operator, then we show the other cases:

$$p_1 \mid p_2 \simeq p_1 \rfloor p_2 \; + \; p_2 \rfloor p_1 \; + \; p_1 \| p_2$$

Moreover we have the following equalities:

$$\begin{aligned}
\mu.p \rfloor q &\simeq \mu.(p \mid q) \\
(p_1 + p_2) \rfloor q &\simeq p_1 \rfloor q \; + \; p_2 \rfloor q \\
\\
\mu_1.p_1 \| \mu_2.p_2 &\simeq \mathbf{nil} \;\text{ if }\; \mu_1 \neq \bar{\mu}_2 \\
\lambda.p_1 \| \bar{\lambda}.p_2 &\simeq \tau.(p_1 \mid p_2) \\
(p_1 + p_2) \| q &\simeq p_1 \| q \; + \; p_2 \| q \\
p \| (q_1 + q_2) &\simeq p \| q_1 \; + \; p \| q_2 \\
\\
(\mu.p) \setminus \alpha &\simeq \mathbf{nil} \;\text{ if }\; \mu \in \{\alpha, \overline{\alpha}\} \\
(\mu.p) \setminus \alpha &\simeq \mu.(p \setminus \alpha) \;\text{ if }\; \mu \neq \alpha, \overline{\alpha} \\
(p_1 + p_2) \setminus \alpha &\simeq p_1 \setminus \alpha \; + \; p_2 \setminus \alpha \\
\\
(\mu.p)[\phi] &\simeq \phi(\mu).p[\phi] \\
(p_1 + p_2)[\phi] &\simeq p_1[\phi] \; + \; p_2[\phi] \\
\\
\mathbf{nil} \setminus \alpha &\simeq \mathbf{nil}[\phi] \simeq \mathbf{nil} \mid \mathbf{nil} \simeq \mathbf{nil} \rfloor p \simeq \mathbf{nil} \| p \simeq p \| \mathbf{nil} \simeq \mathbf{nil}
\end{aligned}$$

□

From the previous theorem, it follows that every finite CCS agent can be equivalently written using action prefix, sum and **nil**. Then, the axioms that characterize the strong bisimilarity relation are the following:

$$\begin{aligned}
p + \mathbf{nil} &= p \\
p_1 + p_2 &= p_2 + p_1 \\
p_1 + (p_2 + p_3) &= (p_1 + p_2) + p_3 \\
p + p &= p
\end{aligned}$$

Note that the axioms simply assert that processes with sum define an idempotent, commutative monoid whose neutral element is **nil**.

10.7. Weak Semantics of CCS

Let us now see an example that illustrates the limits of strong bisimilarity as a behavioural equivalence between agents.

Example 10.31
Let p and q be the following CCS agents:

$$p = \tau.\,\mathbf{nil} \qquad q = \mathbf{nil}$$

Obviously the two agents are distinguished by the (invisible) action τ. So they are not bisimilar, but, since we consider τ as an internal action, not visible from outside of the system, we have that, according to the observable behaviours, they should not be distinguished.

The above example shows that strong bisimilarity is not abstract enough. So we could think to abstract away from the invisible (τ-labelled) transitions by defining a new relation. This relation is called *weak bisimilarity*. We start by defining a new, more abstract, LTS.

10.7.1. Weak Bisimilarity

Definition 10.32 (Weak transitions)
We let $\Longrightarrow$ be the weak transition relation *on the set of states of an LTS defined as follows:*

$$p \overset{\tau}{\Rightarrow} q \quad \textit{iff} \quad p \overset{\tau}{\rightarrow} \ldots \overset{\tau}{\rightarrow} q \;\vee\; p = q$$
$$p \overset{\lambda}{\Rightarrow} q \quad \textit{iff} \quad p \overset{\tau}{\Rightarrow} p' \overset{\lambda}{\rightarrow} q' \overset{\tau}{\Rightarrow} q$$

Note that $p \overset{\tau}{\Rightarrow} q$ means that q can be reached from p via a possibly empty sequence of τ-transitions, i.e., $\overset{\tau}{\Rightarrow}$ coincides with the reflexive and transitive closure $(\overset{\tau}{\rightarrow})^*$ of invisible transition $\overset{\tau}{\rightarrow}$, while $p \overset{\lambda}{\Rightarrow} q$ requires the execution of one visible transition (the one labelled with λ).

Now, as done for the strong bisimilarity, we define a function $\Psi : \mathscr{P}(P \times P) \rightarrow \mathscr{P}(P \times P)$ which takes a relation on P and returns another relation by exploiting weak transitions:

$$p\ \Psi(R)\ q \overset{\text{def}}{=} \begin{cases} \forall \mu, p'. \quad p \overset{\mu}{\rightarrow} p' \quad \text{implies} \quad \exists q'.\, q \overset{\mu}{\Rightarrow} q' \quad \text{and} \quad p'\ R\ q' \\ \forall \mu, q'. \quad q \overset{\mu}{\rightarrow} q' \quad \text{implies} \quad \exists p'.\, p \overset{\mu}{\Rightarrow} p' \quad \text{and} \quad p'\ R\ q' \end{cases}$$

And we define the *weak bisimilarity* as follows:

$$p \approx q \;\text{ iff }\; \exists R.\; p\ R\ q \wedge \Psi(R) \sqsubseteq R$$

This relation seems to improve the notion of equivalence w.r.t. $\simeq$, because $\approx$ abstracts away from the invisible transitions as we required. Unfortunately, there are two problems with this relation. First, the $\Rightarrow$ LTS is infinite branching also for guarded terms (consider e.g. $rec\ x.(\tau.x|\alpha.nil)$, analogous to the agent discussed in example 10.19). Thus function Ψ is not continuous, and the minimal fixpoint, which exists anyway, cannot be reached in general with an ω-chain of approximations. Second, and much worse, weak bisimilarity is not a congruence with respect to the + operator, as the following example shows. As a (minor) consequence, weak bisimilarity, differently than strong bisimilarity, cannot be axiomatized.

Example 10.33
Let p and q be the following CCS agents:

$$p = \alpha.\,\mathbf{nil} \qquad q = \tau.\alpha.\,\mathbf{nil}$$

Obviously for the weak equivalence we have $p \approx q$, since their behaviours differ only by the ability to perform an invisible action τ. Now we define the following context:

$$C[\_] = \_ + \beta.\,\mathbf{nil}$$

Then by embedding p and q within the context C[_] we obtain:

$$C[p] = \alpha.\,\mathbf{nil} + \beta.\,\mathbf{nil} \not\approx \tau.\alpha.\,\mathbf{nil} + \beta.\,\mathbf{nil} = C[q]$$

In fact $C[q]$ can perform a τ-transition and become $\alpha.\,\mathbf{nil}$, while $C[p]$ has only one invisible weak transition that can be used to match such a step, but such weak transition is the idle step $C[p] \stackrel{\tau}{\Rightarrow} C[p]$ and $C[p]$ is clearly not equivalent to $\alpha.\,\mathbf{nil}$ (because the former can perform a β-transition that the latter cannot simulate). This phenomenon is due to the fact that τ-transitions are not observable but can be used to discard some non-deterministic choices. While quite unpleasant, the above fact is not in any way due to a CCS weakness, or misrepresentation of reality, but rather enlightens a general property of nondeterministic choice in systems represented as black boxes.

10.7.2. Weak Observational Congruence

As shown by the Example 10.33, weak bisimilarity is not a congruence relation. In this section we will show a possible (partial) solution. Since weak bisimilarity equivalence is a congruence for all operators except sum, to fix our problem it is enough to impose closure for all sum contexts.

Let us consider the Example 10.33, where the execution of a τ-transition forces the system to make a choice which is invisible to an external observer. In order to make this kind of choices observable we can define the relation $\cong$ as follows:

$$p \cong q \;\text{ iff }\; p \approx q \;\wedge\; \forall r.\, p + r \approx q + r$$

This relation, called *weak observational congruence*, can be defined directly as:

$$p \cong q \stackrel{\text{def}}{=} \begin{cases} \forall p'.\quad p \xrightarrow{\tau} p' \quad \text{implies} \quad \exists q'.\, q \xrightarrow{\tau}\xrightarrow{\tau} q' \qquad \text{and} \quad p' \approx q' \\ \forall \lambda, p'.\quad p \xrightarrow{\lambda} p' \quad \text{implies} \quad \exists q'.\, q \xRightarrow{\lambda} q' \qquad \text{and} \quad p' \approx q' \\ \text{(and vice versa)} \end{cases}$$

As we can see we avoided the possibility to stop after the execution of an internal action. Notice however that this is not a recursive definition, since $\cong$ is simply defined in terms of $\approx$. Now it is obvious that $\alpha.\,\mathbf{nil} \not\cong \tau\alpha.\,\mathbf{nil}$.

The relation $\cong$ is a congruence but as we can see in the following example it is not a (weak) bisimulation according to Ψ, namely $\cong \not\subseteq \Psi(\cong)$.

Example 10.34
Let p and q defined as follows:

$$p = \alpha.\tau.\beta.\,\mathbf{nil} \quad \textit{and} \quad q = \alpha.\beta.\,\mathbf{nil}$$

we have:

$$p \cong q$$

but, according to the weak bisimulation game, if Alice plays α on p, Bob has no chance of playing α and of reaching a state in relation $\cong$ with the continuation of p. Letting

$$p' = \tau.\beta.\,\mathbf{nil} \quad \textit{and} \quad q' = \beta.\,\mathbf{nil}$$

we have $p' \not\cong q'$. Thus $\cong$ is not a pre-fixpoint of Ψ.

It is possible to prove that the equivalence relation $\cong$ can be axiomatized by adding to the axioms for strong bisimilarity the following three Milner's τ laws:

$$\begin{aligned} &p + \tau.p = \tau.p \\ &\mu.(p + \tau.q) = \mu.(p + \tau.q) + \mu.q \\ &\mu.\tau.p = \mu.p \end{aligned}$$

10.7.3. Dynamic Bisimilarity

As shown by the Example 10.34 the observational congruence is not a bisimulation. In this section we present the largest relation which is at the same time a congruence and a Ψ-bisimulation. It is called *dynamic bisimilarity* and was introduced by Vladimiro Sassone.

We define the dynamic bisimilarity $\approx_d$ as the largest relation that satisfies:

$$p \approx_d q \quad \text{implies} \quad \forall C.\ C[p]\ \Psi(\approx_d)\ C[q]$$

In this case, at every step we close the relation by comparing the behaviour w.r.t. any possible embedding context. In terms of game theory this definition can be viewed as "at each turn Alice is also allowed to insert both agents into the same context in order to win."

We can define the dynamic bisimilarity as follows:

$$p\ \Theta(R)\ q \stackrel{\text{def}}{=} \begin{cases} \forall p'.\quad p \xrightarrow{\tau} p' \quad \text{implies} \quad \exists q'.\ q \xrightarrow{\tau} \stackrel{\tau}{\Longrightarrow} q' \quad \text{and} \quad p'\ R\ q' \\ \forall \lambda, p'.\quad p \xrightarrow{\lambda} p' \quad \text{implies} \quad \exists q'.\ q \stackrel{\lambda}{\Longrightarrow} q' \quad \text{and} \quad p'\ R\ q' \\ \text{(and vice versa)} \end{cases}$$

Then, R is a *dynamic bisimulation* if $\Theta(R) \sqsubseteq R$, and the dynamic bisimilarity is obtained by letting:

$$\approx_d = \bigsqcup_{R \subseteq \Theta(R)} R$$

Example 10.35
Let p and q be defined as in the Example 10.34.

$$p = \alpha.\tau.\beta.\,\mathbf{nil} \quad \textit{and} \quad q = \alpha.\beta.\,\mathbf{nil}$$

$$p' = \tau.\beta.\,\mathbf{nil} \quad \textit{and} \quad q' = \beta.\,\mathbf{nil}$$

we have:

$$p \not\approx_d q \quad \textit{and} \quad p' \not\approx_d q'$$

As for the observational congruence we can finitely axiomatize the dynamic bisimilarity. The axiomatization of $\approx_d$ is obtained by omitting the third Milner's τ law as follows:

$$p + \tau p = \tau p$$
$$\mu(p + \tau q) = \mu(p + \tau q) + \mu q$$

11. Temporal Logic and μ-Calculus

As we have discussed in the previous chapter (see Section 10.5) modal logic is a powerful tool that allows to check some behavioral properties of systems. In Section 10.5 the focus was on Hennessy-Milner logic, whose main limitation is due to its finitary structure: only local properties can be investigated. In this chapter we show some extensions of Hennessy-Milner logic that increase the expressiveness of the formulas. The most powerful language that we will present is the *μ-Calculus*. It allows to express complex constraints about the infinite behaviour of our systems.

Classically, we can divide the properties to be investigated in three categories:

- *safety*: if the property expresses the fact that something bad will not happen.
- *liveness*: if the property expresses the fact that something good will happen.
- *fairness*: if the property expresses the fact that something good will happen infinitely many times.

11.1. Temporal Logic

The first step in extending modal logic is to introduce the concept of time in our models. This will extend the expressiveness of modal logic, making it able to talk about concepts like "ever","never" or "sometimes". In order to represent the concept of time in our logics we have to represent it in a mathematical fashion. In our discussion we assume that the time is discrete and infinite.

While temporal logic shares similarities with HM-logic, note that:

- temporal logic is based on a set of *atomic propositions* whose validity is associated with a set of states, i.e., the observations are taken on states and not on (actions labeling the) arcs;
- temporal operators allows to look further than the "next" operator of HML;
- as we will see, the choice of representing the time as linear (linear temporal logic) or as tree (computation tree logic) will lead to different types of logic, that roughly correspond to the trace semantic view vs the bisimulation semantics view.

11.1.1. Linear Temporal Logic

In the case of *Linear Temporal Logic* (LTL) the time is represented as a line. This means that the evolutions of the system are linear, they proceed from a state to another without making any choice. The formulas of LTL are based on a set of *atomic propositions*, which can be composed using the classical logic operators together with the following temporal operators:

- O: is called *next* operator. The formula $O\phi$ means that ϕ is true in the next state (i.e., in the next instant of time). Some literature uses X or N in place of O.
- F: is called *finally* operator. The formula $F\phi$ means that ϕ is true sometime in the future.
- G: The formula $G\phi$ means that ϕ is always (*globally*) valid in the future.
- U: is called *until* operator. The formula $\phi_1 U \phi_2$ means that ϕ_1 is true until the first time that ϕ_2 is true.

The syntax of LTL is as follows:

$$\phi ::= true \mid false \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid O\phi \mid F\phi \mid G\phi \mid \phi_1 U \phi_2$$

In the following we let $\phi_0 \rightarrow \phi_1$ denote the logical implication, whose meaning is $(\neg\phi_0) \vee \phi_1$.

In order to represent the state of the system while the time elapses we introduce the following mathematical structure.

Definition 11.1 (Linear structure)
Let P be a set of atomic propositions and $S : P \rightarrow 2^{\omega}$ be a function from the atomic propositions to subsets of natural numbers defined as follows:

$$\forall p \in P.\ S(p) = \{x \in \omega \mid x \text{ satisfies } p\}$$

Then we call the pair (S, P) a linear structure.

In a linear structure, the natural numbers $0, 1, 2 \ldots$ represent the time instants, and the states in them, and S represents for every predicate the states where it holds, or, alternatively, it represents for every state the predicates which it satisfies.The operators of LTL allows to quantify (existentially and universally) w.r.t. the traversed states. To define the satisfaction relation, we need to check properties on future states, like some sort of "time travel". To this aim we define the following *shifting* operation on S:

$$\forall i \in \omega\ \forall p \in P.\ S^i(p) = \{x - i \mid x \geq i \ \wedge\ x \in S(p)\}$$

As done for the HM-logic, we define the satisfaction operator $\models$ as follows:

$$\begin{array}{lcl}
S & \models & true \\
S & \models & p \quad \text{if } 0 \in S(p) \\
S & \models & \neg\phi \quad \text{if it is not true that } S \models \phi \\
S & \models & \phi_1 \wedge \phi_2 \quad \text{if } S \models \phi_1 \text{ and } S \models \phi_2 \\
S & \models & \phi_1 \vee \phi_2 \quad \text{if } S \models \phi_1 \text{ or } S \models \phi_2 \\
S & \models & O\phi \quad \text{if } S^1 \models \phi \\
S & \models & F\phi \quad \text{if } \exists i \in \omega \text{ such that } S^i \models \phi \\
S & \models & G\phi \quad \text{if } \forall i \in \omega \text{ it holds } S^i \models \phi \\
S & \models & \phi_1 U \phi_2 \quad \text{if } \exists i \in \omega \text{ such that } S^i \models \phi_2 \text{ and } \forall j < i.\ S^j \models \phi_1
\end{array}$$

Two LTL formulas ϕ and ϕ' are called *equivalent*, written $\phi \equiv \phi'$ if for any S we have $S \models \phi$ iff $S \models \phi'$.

From the satisfaction relation it is easy to check that the operators F and G can be expressed in terms of the until operator as follows:

$$\begin{array}{lcl}
F\phi & \equiv & true\ U\ \phi \\
G\phi & \equiv & \neg F \neg\phi \quad \equiv \quad \neg(true\ U\ \neg\phi)
\end{array}$$

We now show some examples that illustrate how powerful the LTL is.

Example 11.2

- *$G\neg p$: expresses the fact that p will never happen, so it is a safety property.*
- *$p \rightarrow Fq = \neg p \vee Fq$: expresses the fact that if p happens then also q will happen sometime in the future.*
- *GFp: expresses the fact that p happens infinitely many times in the future, so it is a fairness property.*
- *FGp: expresses the fact that p will always hold some time in the future.*
- *$G(request \rightarrow (request\ U\ grant))$: expresses the fact that whenever a request is made it holds continuously until it is eventually granted.*

11.1.2. Computation Tree Logic

In this section we introduce CTL and CTL^* two logics which use trees as models of the time. CTL and CTL^* extend LTL with two operators which allows to express properties on paths over trees. The difference between CTL and CTL^* is that the former is a restricted version of the latter. So we start by introducing CTL^*.

We introduce two new operators on paths:

- E: the formula $E\phi$ (read "possibly ϕ") means that there *exists* some path that satisfies ϕ;
- A: the formula $A\phi$ (read "inevitably ϕ") means that each path of the tree satisfies ϕ, i.e., that ϕ is satisfied along *all* paths.

The syntax of CTL is as follows:

$$\phi ::= true \mid false \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid O\phi \mid F\phi \mid G\phi \mid \phi_1 U \phi_2 \mid E\phi \mid A\phi$$

This time the state of the system is represented by using infinite trees as follows.

Definition 11.3 (Infinite tree)
Let $T = (V, \rightarrow)$ be a tree, with V the set of nodes, v_0 the root and $\rightarrow \subseteq V \times V$ the parent-child relation. We say that T is an infinite tree *if the following holds:*

$$\rightarrow \text{ is total on } V, \text{ namely } \forall v \in V\ \exists w \in V.\ v \rightarrow w$$

Definition 11.4 (Branching structure)
Let P be a set of atomic propositions, $T = (V, \rightarrow)$ be an infinite tree and $S : P \rightarrow 2^V$ be a function from the atomic propositions to subsets of nodes of V defined as follows:

$$\forall p \in P.\ S(p) = \{x \in V \mid x \text{ satisfies } p\}$$

Then we call (T, S, P) a branching structure.

We are interested in infinite paths on trees.

Definition 11.5 (Infinite paths)
Let $(V, \rightarrow)$ be an infinite tree and $\pi = v_0, v_1, \ldots, v_n, \ldots$ be an infinite sequence of nodes in V. We say that π is an infinite path *over $(V, \rightarrow)$ iff*

$$\forall i \in \omega.\ v_i \rightarrow v_{i+1}$$

As for the linear case, we need a shifting operators on path. So for $\pi = v_0, v_1, \ldots, v_n, \ldots$ we let π^i be defined as follows:

$$\forall i \in \omega.\ \pi^i = v_i, v_{i+1}, \ldots$$

Let (T, S, P) be a branching structure and $\pi = v_0, v_1, \ldots, v_n, \ldots$ be an infinite path. We define the $\models$ relation as follows:

state operators:

$$
\begin{aligned}
S,\pi &\models true \\
S,\pi &\models p \quad \text{if } v_0 \in S(p) \\
S,\pi &\models \neg\phi \quad \text{if it is not true that } S,\pi \models \phi \\
S,\pi &\models \phi_1 \wedge \phi_2 \quad \text{if } S,\pi \models \phi_1 \text{ and } S,\pi \models \phi_2 \\
S,\pi &\models \phi_1 \vee \phi_2 \quad \text{if } S,\pi \models \phi_1 \text{ or } S,\pi \models \phi_2 \\
S,\pi &\models O\phi \quad \text{if } S,\pi^1 \models \phi \\
S,\pi &\models F\phi \quad \text{if } \exists i \in \omega \text{ such that } S,\pi^i \models \phi \\
S,\pi &\models G\phi \quad \text{if } \forall i \in \omega \text{ it holds } S,\pi^i \models \phi \\
S,\pi &\models \phi_1 U \phi_2 \quad \text{if } \exists i \in \omega \text{ such that } S,\pi^i \models \phi_2 \text{ and for all } j < i.\ S,\pi^j \models \phi_1
\end{aligned}
$$

path operators

$$
\begin{aligned}
S,\pi &\models E\phi \quad \text{if there exists } \pi_1 = v_0, v_1', \ldots, v_n', \ldots \text{ such that } S,\pi_1 \models \phi \\
S,\pi &\models A\phi \quad \text{if for all paths } \pi_1 = v_0, v_1', \ldots, v_n', \ldots \text{ we have } S,\pi_1 \models \phi
\end{aligned}
$$

Two CTL^* formulas ϕ and ϕ' are called *equivalent*, written $\phi \equiv \phi'$ if for any S,π we have $S,\pi \models \phi$ iff $S,\pi \models \phi'$.

Let us see some examples.

Example 11.6

- *EOp: it is the analogous of the next operator $\Diamond$ in modal logic.*
- *AGp: expresses the fact that p happens in all reachable states.*
- *EFp: expresses the fact that p happens in some reachable state.*
- *AFp: expresses the fact that on every path there exists a state where p holds.*
- *$E(pUq)$: expresses the fact that there exists a path where p holds until q.*
- *$AGEFp$: in every future exists a successive future where p holds.*

The formulas of CTL are obtained by restricting CTL^*: a CTL^* formula is a CTL formula if the followings hold:

- A and E appear only immediately before a linear operator (i.e., F,G,U and O).
- each linear operator appears immediately after a quantifier (i.e., A and E).

It is evident that CTL and LTL are both subsets[1] of CTL^*, but they are not equivalent to each other. Without going into the detail, we mention that:

- no CTL formula is equivalent to the LTL formula $F(Gp)$;
- no LTL formula is equivalent to the CTL formula $AG(p \rightarrow (EOq \wedge EO\neg q))$

Finally, we note that all CTL formulas can be written in terms of the minimal set of operators *true*, $\neg$, $\vee$, EG, EU, EO. In fact, for the remaining operators we have the following logical equivalences:

$$
\begin{aligned}
EF\phi &\equiv E(true\ U\ \phi) \\
AO\phi &\equiv \neg(EO\neg\phi) \\
AG\phi &\equiv \neg(EF\neg\phi) \equiv \neg E(true\ U\ \neg\phi) \\
AF\phi &\equiv A(true\ U\ \phi) \equiv \neg(EG\neg\phi) \\
A(\phi\ U\ \varphi) &\equiv \neg(E(\neg\varphi\ U\ \neg(\phi \vee \varphi)) \vee EG\neg\varphi)
\end{aligned}
$$

[1] An LTL formula ϕ is read as the CTL^* formula $A\phi$.

11.2. μ-Calculus

Now we introduce the μ-calculus. The idea is to add the least and greatest fixpoint operators to modal logic. This fits nicely with the fact that many interesting properties can be conveniently expressed as fixpoints. The two operators that we introduce are the following:

- $\mu x.\phi$ is the least fixpoint of ϕ.
- $\nu x.\phi$ is the greatest fixpoint of ϕ.

The syntax of μ-calculus is as follows:

$$\phi ::= true \mid false \mid x \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \Diamond\phi \mid \Box\phi \mid \mu x.\phi \mid \nu x.\phi$$

Note that, in order to apply the fixpoint operators we require that ϕ is monotone, this means that any occurrence of x in ϕ must be preceded by an even number of negations. The μ-calculus is interpreted on LTSs.

Let $(V, \to)$ be an LTS, X be the set of predicate variables and P be a set of predicates, we introduce a function $\rho : P \cup X \to 2^V$ which associates to each predicate and each free variable a subset of vertices. Then we define the denotational semantics of μ-calculus which maps each predicate to the subset of states in which it holds as follows:

$$\begin{aligned}
[\![true]\!]\rho &= V \\
[\![false]\!]\rho &= \varnothing \\
[\![x]\!]\rho &= \rho x \\
[\![p]\!]\rho &= \rho p \\
[\![\neg\phi]\!]\rho &= V \setminus [\![\phi]\!]\rho \\
[\![\phi_1 \wedge \phi_2]\!]\rho &= [\![\phi_1]\!]\rho \cap [\![\phi_2]\!]\rho \\
[\![\phi_1 \vee \phi_2]\!]\rho &= [\![\phi_1]\!]\rho \cup [\![\phi_2]\!]\rho \\
[\![\Diamond\phi]\!]\rho &= \{\, v \mid \exists v'.\, v \to v' \ \wedge\ v' \in [\![\phi]\!]\rho \,\} \\
[\![\Box\phi]\!]\rho &= \{\, v \mid \forall v'.\, v \to v' \Rightarrow v' \in [\![\phi]\!]\rho \,\} \\
[\![\mu x.\phi]\!]\rho &= \mathrm{fix}\, \lambda S.\, [\![\phi]\!]\rho[^S/_x] \\
[\![\nu x.\phi]\!]\rho &= \mathrm{Fix}\, \lambda S.\, [\![\phi]\!]\rho[^S/_x]
\end{aligned}$$

Example 11.7

- $[\![\mu x.x]\!]\rho = \varnothing$
- $[\![\nu x.x]\!]\rho = V$
- $[\![\mu x.\Diamond x]\!]\rho = \mathrm{fix}\, \lambda S.\{v \mid \exists v'.\, v \to v' \wedge v' \in S\}$
  we have:
  $$S_0 = \varnothing \qquad S_1 = \{v \mid \exists v'.\, v \to v' \wedge v' \in \varnothing\} = \varnothing \quad \textit{(fixpoint reached)}$$
- $[\![\mu x.\Box x]\!]\rho = \mathrm{fix}\, \lambda S.\{v \mid \forall v'.\, v \to v' \Rightarrow v' \in S\}$
  we have:

  $S_0 = \varnothing \qquad S_1 = \{v \mid \forall v'.\, v \to v' \Rightarrow v' \in \varnothing\} = \{v \mid v \not\to\}$ *the set of vertices with no outgoing arcs*

  $S_2 = \{v \mid \forall v'.\, v \to v' \Rightarrow v' \in S_1\} =$ *the set of vertices with outgoing paths of length at most* 1

  $S_n = \{v \mid \forall v'.\, v \to v' \Rightarrow v' \in S_{n-1}\} =$ *the set of vertices with outgoing paths of length at most* $n-1$

  $$\bigcup_{i \in \omega} S_i = \textit{ vertices with only finite outgoing paths}$$

- $[\![\nu x.\Box x]\!]\rho = \text{Fix}\,\lambda S.\{v \mid \forall v', v \to v' \Rightarrow v' \in S\}$
  we have:
  $$S_0 = V \qquad S_1 = \{v \mid \forall v'.\ v \to v' \Rightarrow v' \in V\} = V \qquad \textit{(fixpoint reached)}$$

- $[\![\mu x.p \vee \Diamond x]\!]\rho = \text{fix}\,\lambda S.\rho p \cup \{v \mid \exists v'.v \to v' \wedge v' \in S\}$ *(similar to EFp, meaning some node in ρp is reachable)*
  we have:

  $S_0 = \emptyset$ $\qquad S_1 = \rho p$ $\qquad S_2 = \rho p \cup \{v \mid \exists v'.\ v \to v' \wedge v' \in \rho p\} = \rho p$ *is reachable in at most one step*

  $S_n = \rho p$ *is reachable in at most* $n-1$ *steps* $\qquad \bigcup_{i \in \omega} S_i = \rho p$ *is reachable (in any number of steps)*

- $[\![\nu x.\mu y.(p \wedge \Diamond x) \vee \Diamond y]\!]\rho$ *(corresponds to $EGFp$)*
  start a path, $\mu y.(p \wedge \Diamond x) \vee \Diamond y$ means that after a finite number of steps you find a vertex where both (1) p holds and (2) you can reach a vertex where the property recursively holds.

- $[\![\mu x.(p \wedge \Box x \wedge \Diamond x) \vee q]\!]\rho = \text{fix}\,\lambda S.(\rho p \cap \{v \mid \forall v'.\ v \to v' \Rightarrow v' \in S\} \cap \{v \mid \exists v'.\ v \to v' \wedge v' \in S\}) \cup \rho q$ *(corresponds to $ApUq$)*

  Note that in this case the $\Diamond x$ is necessary in order to ensure that the state is not a deadlock one.

- $[\![\mu x.(p \wedge \Diamond x) \vee q]\!]\rho = \text{fix}\,\lambda S.(\rho p \cap \{v \mid \exists v'.\ v \to v' \wedge v' \in S\}) \cup \rho q$ *(corresponds to $EpUq$)*

11.3. Model Checking

The problem of model checking consists in the, possibly automatic, verification of whether a given model of a system meets or not a given logic specification of the properties the system should satisfy, like absence of deadlocks.

The main ingredients of model checking are:

- an LTS (the model) and a vertex (the initial state);
- a formula (in temporal or modal logic) you want to check (for that state in the model)

The result of model checking should be either a positive answer (the given state in the model satisfies the formula) or some counterexample explaining one possible reason why the formula is not satisfied.

In the case of concurrent systems, the LTS is often given implicitly, as the one associated with a term of some process algebra, because in this way the structure of the system is handled more conveniently. However the size of the actual translation can explode even if the system is finite state. For example, let $p_i = \alpha_i.\,\mathbf{nil}$ for $i = 1, .., n$ and take the CCS process $s = p_1 \mid p_2 \mid \cdots \mid p_n$: the number of reachable states of the resulting model is 2^n.

One possibility to tackle the state explosion problem is to minimize the system according to some suitable equivalence. Note that minimization can take place also while combining subprocesses and not just at the end. Of course, this technique is viable only if the minimization is related to an equivalence relation that respects the properties to be checked. For example, the μ-calculus is invariant w.r.t. bisimulation, thus we can minimize CCS processes up to bisimilarity before model checking them.

In model checking algorithms, it is often convenient to proceed by evaluating formulas with the aid of dynamic programming. The idea is to work in a bottom-up fashion: starting from the atomic predicates that appear in the formula, we mark all the states with the sub formulas they satisfy. When a variable is encountered, a separate activation of the procedure is allocated for computing the fixpoint of the corresponding recursive definition. The complexity becomes very large in the case of formulas that involve many least and greatest fix points in alternation.

12. π-Calculus

The structures of today's communication systems are not statically defined, but they change continuously according to the needs of the users. The CCS calculus we saw in Chapter 10 is unsuitable for modeling such systems, since its communication structure (the channels) cannot evolve dynamically. In this chapter we present the π-calculus, an extension of CCS introduced by Robin Milner, Joachim Parrow and David Walker in 1989 which allows to model mobile systems. The main feature of the π-calculus is its ability of creating new channel names and of sending them in messages allowing agents to change their connections. For example, consider the case of the CCS-like process (with value passing)

$$(P|Q)\backslash a \mid R$$

and suppose that P and Q can communicate over the channel a, which is private to them, and that P and R share a channel b for exchanging messages. If we allow channel names to be sent as message values, then it could be the case P sends the name a over the channel b, like in

$$P = \overline{b}a.P'$$

that Q waits for a message on a, like in

$$Q = a(x).Q'$$

and that R wants to input a channel name on b, where to send a message m, like in

$$R = b(y).\overline{y}m.\,\mathbf{nil}$$

After the communication between P and R takes place over the channel b we would like the scope of a be extended to include R, like in

$$((P'|Q) \mid \overline{a}m.\,\mathbf{nil})\backslash a$$

so that Q can then input m:

$$((P'|Q'[m/x]) \mid \mathbf{nil})\backslash a$$

All this cannot be achieved in CCS, where restriction is a static operator. Moreover, suppose a process S is initially running in parallel with R, like in

$$((P|Q)\backslash a \mid S) \mid R$$

After the communication over b we would like the name a to be private to P, Q, R but not known by S, thus if a is already used by S, it must be the case that after the scope extrusion a fresh private name c, not available to S, is used by P, Q, R, like in

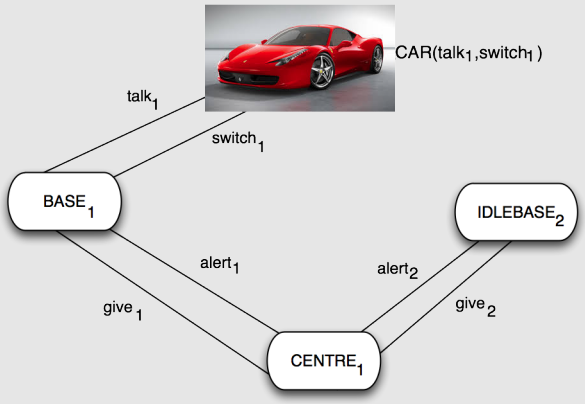
$$(((P'[c/a]|Q[c/a]) \mid S) \mid \overline{c}m.\,\mathbf{nil})\backslash c$$

so that the message $\overline{c}m$ cannot be intercepted by S.

The general mechanism for handling name mobility makes the formalization of the semantics of the π-calculus more complicated than that of CCS, especially because of the side-conditions that serve to guarantee that certain names are fresh.

Let us start with an example which illustrates how the π-calculus can formalize a mobile telephone system.

Example 12.1 (Mobile Phones)
The following figure which represents a mobile phone network: while the car travels, the phone can communicate with different bases in the city, but just one at a time, typically the closest to its position. The communication centre decides when the base must be changed and then the channel for accessing the new base is sent to the car through the switch channel.



As for CCS, also in this case we describe agent behaviour by defining the reachable states:

$$CAR(talk, switch) \stackrel{\text{def}}{=} \overline{talk}.CAR(talk, switch) + switch(talk', switch').CAR(talk', switch')$$

A car can talk on the channel assigned by the communication centre (action $\overline{talk}$). Alternatively the car can receive (action $switch(talk', switch')$) a new pair of channels ($talk'$ and $switch'$) and change the base to which it is connected.

$$BAS\,E_i \stackrel{\text{def}}{=} BAS\,E(talk_i, switch_i, give_i, alert_i) \stackrel{\text{def}}{=}$$
$$talk_i.BAS\,E_i + give_i(talk', switch').\overline{switch_i}(talk', switch').IDLEBAS\,E_i$$

$$IDLEBAS\,E_i \stackrel{\text{def}}{=} IDLEBAS\,E(talk_i, switch_i, give_i, alert_i) \stackrel{\text{def}}{=} alert_i.BAS\,E_i$$

A generic base can be in two possible states: $BAS\,E$ or $IDLEBAS\,E$. In the first case the base is connected to the car, so either the phone can talk or the base can receive two channels from the centre and send them to the car for allowing it to change base. In the second case the base is idle, so it can only be awakened by the communication centre.

$$CENTRE_1 \stackrel{\text{def}}{=} CENTRE_1(give_1, alert_1, give_2, alert_2) = \overline{give_1}(talk_2, switch_2).\overline{alert_2}.CENTRE_2$$
$$CENTRE_2 \stackrel{\text{def}}{=} CENTRE_2(give_1, alert_1, give_2, alert_2) = \overline{give_2}(talk_1, switch_1).\overline{alert_1}.CENTRE_1$$

The communication centre can be in different states according to which base is active. In the example there are only two possible states for the communication centre ($CENTRE_1$ and $CENTRE_2$), because only two bases are considered.

$$SYSTEM_1 \stackrel{\text{def}}{=} (CAR(talk_1, switch_1)|BAS\,E_1|IDLEBAS\,E_2|CENTRE_1)$$

Finally we have the process which represents the entire system in the state where the first car is talking.

Example 12.2 (Secret Channel via Trusted Server)
As another example, consider two processes Alice (A) and Bob (B) that want to establish a secret channel using a trusted server (S) with which they already have trustworthy communication link c_{AS} (for Alice to send private messages to the server) and c_{SB} (for the server to send private messages to Bob). The system can be represented by the expression:

$$Sys \stackrel{\text{def}}{=} (c_{AS})(c_{BS})(A|S|B)$$

where the restrictions *(c_{AS}) and (c_{BS}) guarantees that the link c_{AS} and c_{SB} are not visible from the*

environment and where the processes A, S and B are specified as follows:

$$\begin{aligned} A &\stackrel{\text{def}}{=} (c_{AB})\bar{c}_{AS}\, c_{AB}.\bar{c}_{AB}m.p_A \\ S &\stackrel{\text{def}}{=} !c_{AS}(x).\bar{c}_{SB}x.\,\mathbf{nil} \\ B &\stackrel{\text{def}}{=} c_{SB}(y).y(m).q_B \end{aligned}$$

Name restriction, written (c_{AB}) is similar to the CCS operator $\backslash c_{AB}$, with the important difference that in π-calculus the scope of the restriction can change as the process evolves. Alice defines a private name c_{AB} that wants to use for communicating with B, then Alice sends the name c_{AB} to the trusted server over their private shared link c_{AS} and finally sends the message m on the channel c_{AB} and continues as p_A. The server continuously wait for messages from Alice on channel c_{AS} and forwards the content to Bob. Here the replication operator ! allows to serve multiple requests from Alice. Bob waits to receive the name y from the server over the channel c_{SB} and then uses y to input the message from Alice and continue as q_B (which can now use both y and m).

12.1. Syntax of π-calculus

The π-calculus has been introduced to model communicating systems where channel names, representing addresses and links, can be created and forwarded. To this aim we rely on a set of channel names $x, y, z, ...$ and extend the CCS actions with the ability to send and receive channel names. In these notes we present the monadic version of the calculus, namely the version where names can be sent only one at a time. We introduce the syntax, with productions for processes and for actions.

$$\begin{aligned} p &::= \mathbf{nil} \mid \pi.p \mid [x = y]p \mid p + p \mid p|p \mid (y)p \mid !p \\ \pi &::= \tau \mid x(y) \mid \overline{x}y \end{aligned}$$

The meaning of the operators for building π-calculus processes is the following:

- **nil** is the inactive agent.
- $\pi.p$ is an agent which can perform an action π and then act like p.
- $[x = y]p$ is the conditional process, which acts like p if $x = y$, and which remains blocked otherwise.
- $p + q$ is the non-deterministic choice between two processes.
- $p|q$ is the parallel composition of two processes.
- $(y)p$ denotes the restriction of the channel y, which makes the name y private in p.[1]
- $!p$ is a replicated process: it behaves as if an unbounded number of concurrent occurrences of p were available in parallel. It is the analogous of the CCS recursive process **rec** x. $(x|p)$.

The meaning of the actions is the following:

- τ as usual is the invisible action.
- $x(y)$ is the input on channel x, the received value would be stored in y.
- $\overline{x}y$ is the output on channel x of the name y.

[1] In the literature the restriction operator is sometimes written $(\nu y)p$ to remark the fact the the name y is "new" to p: we prefer not to use the symbol ν to avoid any conflict with the maximal fixpoint operator of the μ-calculus.

In the above case, we call x the *subject* of the communication (i.e., the channel name where the communication takes place) and y the *object* of the communication (i.e., the channel name that is transmitted or received). As in the λ-calculus, in the π-calculus we have *bound* and *free* occurrence of names. The bounding operators of π-calculus are input and restriction:

- $x(y).p$ (name y is bound in p).
- $(y)p$ (name y is bound in p).

On the contrary, the output prefix is not binding, i.e., if we take the process $\overline{x}y.p$ then the name y is free. Formally, we define by structural induction:

$$\begin{array}{rcl} fn(\mathbf{nil}) & = & \varnothing \\ fn(\tau.p) & = & fn(p) \\ fn(x(y).p) & = & \{x\} \cup (fn(p) \setminus \{y\}) \\ fn(\overline{x}y.p) & = & \{x, y\} \cup fn(p) \\ fn([x = y].p) & = & \{x, y\} \cup fn(p) \\ fn(p_0 + p_1) & = & fn(p_0) \cup fn(p_1) \\ fn(p_0|p_1) & = & fn(p_0) \cup fn(p_1) \\ fn((y).p) & = & fn(p) \setminus \{y\} \\ fn(!p) & = & fn(p) \end{array} \qquad \begin{array}{rcl} bn(\mathbf{nil}) & = & \varnothing \\ bn(\tau.p) & = & bn(p) \\ bn(x(y).p) & = & \{y\} \cup bn(p) \\ bn(\overline{x}y.p) & = & bn(p) \\ bn([x = y].p) & = & bn(p) \\ bn(p_0 + p_1) & = & bn(p_0) \cup bn(p_1) \\ bn(p_0|p_1) & = & bn(p_0) \cup bn(p_1) \\ bn((y).p) & = & \{y\} \cup bn(p) \\ bn(!p) & = & bn(p) \end{array}$$

Note that for both $x(y).p$ and $\overline{x}y.p$ the name x is free in p. Moreover we define the *name set* of p as follows:

$$n(p) = fn(p) \cup bn(p)$$

Unlike for CCS, the restriction operator $(y)p$ does not bind statically the scope of y to coincide with p. In fact in the π-calculus channel names are values, so the process p can send the name y to another process which thus becomes part of the scope of y. The possibility to enlarge the scope of a restricted name is a very useful feature of the π-calculus, called *extrusion*, which allows to modify the structure of private communications between agents.

12.2. Operational Semantics of π-calculus

Likewise CCS, we define the operational semantics by using a rule system, where well formed formulas are triples $p \xrightarrow{\alpha} q$ as for CCS. The possible actions α that can label the transitions are:

1. the silent action τ;
2. the input $x(y)$ of name y on channel x;
3. the free output $\bar{x}y$ of name y on channel x;
4. the bound output $\bar{x}(y)$ of a previously restricted name y on channel x (name extrusion).

The definition of free names $fn(\cdot)$, bound names $bn(\cdot)$ and names $n(\cdot)$ are extended to labels by letting:

$$\begin{array}{rcl} fn(\tau) & = & \varnothing \\ fn(x(y)) & = & \{x\} \\ fn(\overline{x}y) & = & \{x, y\} \\ fn(\overline{x}(y)) & = & \{x\} \end{array} \qquad \begin{array}{rcl} bn(\tau) & = & \varnothing \\ bn(x(y)) & = & \{y\} \\ bn(\overline{x}y) & = & \varnothing \\ bn(\overline{x}(y)) & = & \{y\} \end{array}$$

$$n(\alpha) = fn(\alpha) \cup bn(\alpha)$$

We can now present the operational rules of the π-calculus and briefly comment on them.

$$\text{(Tau)}\ \frac{}{\tau.p \xrightarrow{\tau} p}$$

The rule (Tau) allows to perform invisible actions.

$$\text{(Out)}\ \frac{}{\overline{x}\,y.p \xrightarrow{\overline{x}\,y} p}$$

As we said the π-calculus processes can exchange messages which can contain information (i.e., channel names). The rule (Out) allows p to send the name y on the channel x.

$$\text{(In)}\ \frac{}{x(y).p \xrightarrow{x(w)} p\{^{w}/_{y}\}}\ w \notin fn((y)p)$$

The rule (In) allows to receive in input over x some channel name. The received name w is bound to the name y in the process p. In order to avoid name conflicts, we assume w does not appear as a free name in $(y)p$, i.e., the transition is defined only when w is *fresh*.

$$\text{(SumL)}\ \frac{p \xrightarrow{\alpha} p'}{p + q \xrightarrow{\alpha} p'} \qquad \text{(SumR)}\ \frac{q \xrightarrow{\alpha} q'}{p + q \xrightarrow{\alpha} q'}$$

The rules (SumL) and (SumR) allow the system $p + q$ to behave as p or q.

$$\text{(Match)}\ \frac{p \xrightarrow{\alpha} p'}{[x = x]p \xrightarrow{\alpha} p'}$$

The rule (Match) allows to check the condition between square bracket and unblock the process p. If the matching condition is not satisfied we can not continue the execution.

$$\text{(ParL)}\ \frac{p \xrightarrow{\alpha} p'}{p|q \xrightarrow{\alpha} p'|q}\ bn(\alpha) \cap fn(q) = \varnothing \qquad \text{(ParR)}\ \frac{q \xrightarrow{\alpha} q'}{p|q \xrightarrow{\alpha} p|q'}\ bn(\alpha) \cap fn(p) = \varnothing$$

As for CCS the two rules (ParL) and (ParR) allow the interleaved execution of two π-calculus agents. The side conditions guarantee that the bound names in α (if any) are fresh w.r.t. the idle process. Notice that if we assume that the bound names of α are fresh wrt. the premise of the rule, thanks to the side condition we can conclude that they are fresh also wrt. the consequence.

$$\text{(ComL)}\ \frac{p \xrightarrow{\overline{x}\,z} p' \quad q \xrightarrow{x(y)} q'}{p|q \xrightarrow{\tau} p'|(q'\{^{z}/_{y}\})} \qquad \text{(ComR)}\ \frac{p \xrightarrow{x(y)} p' \quad q \xrightarrow{\overline{x}\,z} q'}{p|q \xrightarrow{\tau} p'\{^{z}/_{y}\}|q'}$$

The rules (ComL) and (ComR) allow the synchronization of two parallel process. The formal name y is replaced with the actual name z in the continuation of the receiver.

$$\text{(Res)}\ \frac{p \xrightarrow{\alpha} p'}{(y)p \xrightarrow{\alpha} (y)p'}\ y \notin n(\alpha)$$

The rule (Res) expresses the fact that if a name y is restricted on top of the process p, then any action which does not involve y can be performed by p.

Now we present the most important rules of π-calculus Open and Close, dealing with *scope extrusion* of channel names. Rule Open *publishes*, i.e. makes free, a private channel name, while rule Close restricts again the name, but with a broader scope.

$$\text{(Open)}\ \frac{p \xrightarrow{\overline{x}\,y} p'}{(y)p \xrightarrow{\overline{x}(w)} p'\{^{w}/_{y}\}}\ y \neq x \quad w \notin fn((y)p)$$

The rule (Open) publishes the private name w, which is guaranteed to be fresh.

$$\text{(CloseL)}\ \frac{p \xrightarrow{\overline{x}\,(w)} p' \quad q \xrightarrow{x(w)} q'}{p|q \xrightarrow{\tau} (w)(p'|q')} \qquad \text{(CloseR)}\ \frac{p \xrightarrow{x(w)} p' \quad q \xrightarrow{\overline{x}\,(w)} q'}{p|q \xrightarrow{\tau} (w)(p'|q')}$$

The rules (CloseL) and (CloseR) transform the object of the communication over x in a private channel between p and q. Name extrusion is a convenient primitive for formalizing secure data transmission, as implemented e.g. via cryptographic protocols.

$$\text{(Rep)}\ \frac{p|!p \xrightarrow{\alpha} p'}{!p \xrightarrow{\alpha} p'}$$

The last rule deals with replication. It allows to replicate a process as many times as needed, in a reentrant fashion, without consuming it. Notice that $!p$ is able also to perform the synchronizations of $p|p$, if any. We conclude this section by showing an example of the use of the rule system.

Example 12.3 (A derivation)
Let us consider the following agent:

$$(((y)\overline{x}\ y.p) \mid q) \mid x(z).r$$

The process $(y)\overline{x}\ y.p$ would like to set up a private channel with $x(z).r$, which however should remain hidden to q.

By using the rule system:

$$\begin{array}{rrrl}
 & & (((y)\overline{x}\ y.p) \mid q) \mid x(z).r \xrightarrow{\alpha} q_1 & \nwarrow_{\text{(Close), } q_1=(w)(q_2\mid q_3),\ \alpha=\tau} \\
 & ((y)\overline{x}\ y.p) \mid q \xrightarrow{\overline{x}(w)} q_2 & x(z).r \xrightarrow{x(w)} q_3 & \nwarrow_{\text{(ParL), } q_2=q_4\mid q} \\
 (y)\overline{x}\ y.p \xrightarrow{\overline{x}(w)} q_4 & w \notin fn(q) & x(z).r \xrightarrow{x(w)} q_3 & \nwarrow_{\text{(Open), } q_4=q_5\{^w/_y\}} \\
\overline{x}\ y.p \xrightarrow{\overline{x}\ y} q_5 & w \notin fn(q)\ \ w \notin fn((y).p) & x(z).r \xrightarrow{x(w)} q_3 & \nwarrow_{\text{(Out)+(In), } q_3=r\{^w/_z\},\ q_5=p} \\
 & w \notin fn(q)\ \ w \notin fn((y).p)\ \ w \notin fn((z).r) & &
\end{array}$$

so we have:

$$\begin{aligned}
q_5 &= p \\
q_4 &= q_5\left\{^w/_y\right\} = p\left\{^w/_y\right\} \\
q_3 &= r\{^w/_z\} \\
q_2 &= q_4 \mid q = p\left\{^w/_y\right\} \mid q \\
q_1 &= (w)(q_2 \mid q_3) = (w)\Big(\ (p\{^w/_y\} \mid q)\ \mid\ (r\{^w/_z\})\ \Big)
\end{aligned}$$

In conclusion:

$$(((y)\overline{x}\ y.p) \mid q) \mid x(z).r \xrightarrow{\tau} (w)\Big(\ (p\{^w/_y\} \mid q)\ \mid\ (r\{^w/_z\})\ \Big)$$

under the conditions:

$$w \notin fn(q)\ \ w \notin fn((y).p)\ \ w \notin fn((z).r)$$

12.3. Structural Equivalence of π-calculus

As we have already noticed for CCS, there are different terms representing essentially the same process. As the complexity of the calculus increases, it is more and more convenient to manipulate terms up to some intuitive structural axioms. In the following we denote by $\equiv$ the least congruence over π-calculus processes that includes α-conversion of bound names and that is induced by the following set of axioms. The relation $\equiv$ is called *structural equivalence*.

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
p + \mathbf{nil} & \equiv & p & p + q & \equiv & q + p & (p + q) + r & \equiv & p + (q + r) \\
p \mid \mathbf{nil} & \equiv & p & p \mid q & \equiv & q \mid p & (p \mid q) \mid r & \equiv & p \mid (q \mid r) \\
(x)\,\mathbf{nil} & \equiv & \mathbf{nil} & (y)(x)p & \equiv & (x)(y)p & (x)(p \mid q) & \equiv & p \mid (x)q \ \text{ if } \ x \notin fn(p) \\
[x = y]\,\mathbf{nil} & \equiv & \mathbf{nil} & [x = x]\,p & \equiv & p & p \mid !p & \equiv & !p
\end{array}$$

12.3.1. Reduction semantics

The operational semantics of π-calculus is much more complicated than that of CCS because it needs to handle name passing and scope extrusion. By exploiting structural equivalence we can define a so-called *reduction semantics* that is simpler to understand. The idea is to define an LTS with silent labels only that models all the interactions that can take place in a process, without considering interaction with the environment. This is accomplished by first rewriting the process to a structurally equivalent normal form and then by applying basic reduction rules. In fact it can be proved that for each π-calculus process p there exists:

- a finite number of names $x_1, x_2, ..., x_k$;
- a finite number of guarded sums $s_1, s_2, ..., s_n$;
- and a finite number of processes $p_1, p_2, ..., p_m$

such that

$$P \equiv (x_1)...(x_k)(s_1|...|s_n|!p_1|...!p_m)$$

Then, a reduction is either a silent action performed by some s_i or a communication from an input prefix of say s_i with an output prefix of say s_j. We write the reduction relation as a binary relation on processes using the notation $p \mapsto q$ for indicating that p reduces to q in one step. The rules defining the relation $\mapsto$ are the following:

$$\frac{}{\tau.p + s \mapsto p} \qquad \frac{}{(x(y).p_1 + s_1)|(\bar{x}z.p_2 + s_2) \mapsto p_1\left\{{}^{z}/_{y}\right\}|p_2}$$

$$\frac{p \mapsto p'}{p|q \mapsto p'|q} \qquad \frac{p \mapsto p'}{(x)p \mapsto (x)p'} \qquad \frac{p \equiv q \quad q \mapsto q' \quad q' \equiv p'}{p \mapsto p'}$$

The reduction semantics can be put in correspondence with the (silent transitions of the) labelled operational semantics by the following theorem.

Theorem 12.4 (Harmony Lemma)
For any π-calculus processes p, p' and any action α we have that:

1. $\exists q.\ p \equiv q \ \wedge\ q \xrightarrow{\alpha} p'$ *implies that* $\exists q'.\ p \xrightarrow{\alpha} q' \ \wedge\ q' \equiv p'$
2. $p \mapsto p'$ *iff* $\exists q.\ p \xrightarrow{\tau} q \ \wedge\ q \equiv p'$.

12.4. Abstract Semantics of π-calculus

Now we present an abstract semantics of π-calculus, namely we do not consider the internal structure of terms but focus on their behaviours. As we saw in CCS one of the main goals of abstract semantics is to find the correct degree of abstraction. Thus also in this case there are many kinds of bisimulations that lead to different bisimilarities, which are useful in different circumstances depending on the properties that we want to study.

We start from *strong bisimulation* of π-calculus which is an extended version of the strong bisimulation of CCS. Then we will present the *weak bisimulation* for π-calculus. An important new feature of π-calculus is the choice of the time the names used as objects of input transitions are assigned their actual values. If they are assigned *before* the choice of the (bi)simulating transition, namely if the choice of the transition may depend on the assigned value, we get the *early* bisimulation. Instead, if the choice must hold for all possible names we have the *late* bisimulation case. As we will see in short, the latter option leads to a finer semantics.

12.4.1. Strong Early Ground Bisimulations

In *early* bisimulation we require that for each name w that an agent can receive on a channel x there exists a state q' in which the bisimilar agent will be after receiving w on x. This means that the bisimilar agent can choose a different transition (and thus a different state q') depending on the observed name w. Formally, a binary relation S on π-calculus agents is a *strong early ground bisimulation* if:

$$\forall p, q. \quad p\ S\ q \Rightarrow \begin{cases} \forall p'.\ \text{if } p \xrightarrow{\tau} p' \text{ then } \exists q'.\ q \xrightarrow{\tau} q' \text{ and } p'\ S\ q' \\ \forall x, y, p'.\ \text{if } p \xrightarrow{\overline{x}y} p' \text{ then } \exists q'.\ q \xrightarrow{\overline{x}y} q' \text{ and } p'\ S\ q' \\ \forall \alpha, p'.\ \text{if } p \xrightarrow{\overline{x}(y)} p' \text{ with } y \notin fn(q), \text{ then } \exists q'.\ q \xrightarrow{\overline{x}(y)} q' \text{ and } p'\ S\ q' \\ \forall x, y, p'.\ \text{if } p \xrightarrow{x(y)} p' \text{ with } y \notin fn(q), \text{ then } \forall w.\ \exists q'.\ q \xrightarrow{x(y)} q' \text{ and } p'\{^w/_y\}\ S\ q'\{^w/_y\} \\ \text{(and vice versa)} \end{cases}$$

The same condition can be written in a more compact form as:

$$\forall p, q. \quad p\ S\ q \Rightarrow \begin{cases} \forall \alpha, p'.\ \text{if } p \xrightarrow{\alpha} p' \text{ with } \alpha \neq x(y) \ \wedge\ bn(\alpha) \cap fn(q) = \varnothing, \text{ then } \exists q'.\ q \xrightarrow{\alpha} q' \text{ and } p'\ S\ q' \\ \forall x, y, p'.\ \text{if } p \xrightarrow{x(y)} p' \text{ with } y \notin fn(q), \text{ then } \forall w.\ \exists q'.\ q \xrightarrow{x(y)} q' \text{ and } p'\{^w/_y\}\ S\ q'\{^w/_y\} \\ \text{(and vice versa)} \end{cases}$$

Two agents p and q are said to be *early bisimilar*, written $p \overset{\circ}{\sim}_E q$, iff:

$$p\ S\ q \text{ for some strong early ground bisimulation } S.$$

Notice that the conditions $bn(\alpha) \notin fn(q)$ and $y \notin fn(q)$ are required, since otherwise a bound name in the action which is fresh in p could be not fresh in q.

12.4.2. Strong Late Ground Bisimulations

In this case of late bisimulation, we require that, if an agent p can perform an input operation on a channel x, then there exists a state q' in which the bisimilar agent will be after receiving any possible value on x. Formally, a binary relation S on π-calculus agents is a *strong late ground bisimulation* if:

$$\forall p, q. \quad p\ S\ q \Rightarrow \begin{cases} \forall \alpha, p'.\ \text{if } p \xrightarrow{\alpha} p' \text{ with } \alpha \neq x(y) \ \wedge\ bn(\alpha) \cap fn(q) = \varnothing, \text{ then } \exists q'.\ q \xrightarrow{\alpha} q' \text{ and } p'\ S\ q' \\ \forall x, y, p'.\ \text{if } p \xrightarrow{x(y)} p' \text{ with } y \notin fn(q), \text{ then } \exists q'.\ q \xrightarrow{x(y)} q' \text{ and } \forall w.\ p'\{^w/_y\}\ S\ q'\{^w/_y\} \\ \text{(and vice versa)} \end{cases}$$

As usual we have that two agents p and q are said to be *late bisimilar*, written $p \overset{\circ}{\sim}_L q$ iff:

$$p\ S\ q \text{ for some strong late ground bisimulation } S.$$

Let us show an example which illustrates the difference between late and early bisimilarities.

Example 12.5 (Early vs late bisimulation)
We show two processes that are early bisimilar but not late bisimilar. Let us consider the processes:

$$\begin{aligned} p &= x(y).\tau.\,\mathbf{nil} + x(y).\,\mathbf{nil} \\ q &= p + x(y).[y = z].\tau.\,\mathbf{nil} \end{aligned}$$

The two processes p and q are early bisimilar. In fact, let q perform an input operation on x by choosing the right branch of the + operation. Then, if the received name y is equal to z, then p can choose to perform the left input operation and reach the state $\tau.\,\mathbf{nil}$ *which is equal to the state reached by q. Otherwise, if* $y \neq z$*, then the guard* $[y = z]$ *is not satisfied and q is blocked and p can choose to perform the right input and reach the state nil.*

On the contrary, if late bisimilarity is considered, then the two agents are not equivalent. In fact p should find a state which can handle all the possible value sent on x. If we choose to move on the left, the choice can work well when $y = z$ *but not in the other cases. On the other hand, if we choose to move on the right the choice does not work well with* $y = z$*.*

The above example shows that late bisimulation is not coarser than early. In fact, it is possible to prove that late bisimulation is strictly finer than early: if two processes are late bisimilar, then they are early bisimilar.

12.4.3. Strong Full Bisimilarity

Unfortunately both early and late ground bisimilarities are not congruences, even in the strong case, as shown by the following counterexample.

Example 12.6 (Ground bisimilarities are not congruences)
Let us consider the following agents:

$$p = \overline{x}\,x.\,\mathbf{nil} \mid x'(y).\,\mathbf{nil} \qquad q = \overline{x}\,x.x'(y).\,\mathbf{nil} + x'(y).\overline{x}\,x.\,\mathbf{nil}$$

The agents p and q are bisimilar (according to both early and late bisimilarities), as they generate isomorphic transition systems. Now, in order to show that ground bisimulations are not congruences, we define the following context:

$$C[\,\_\,] = z(x')(\,\_\,)$$

by filling the hole of $C[\,\_\,]$ once with p and once with q we obtain:

$$p' = C[p] = z(x')(\overline{x}\,x.\,\mathbf{nil} \mid x'(y).\,\mathbf{nil}) \qquad q' = C[q] = z(x')(\overline{x}\,x.x'(y).\,\mathbf{nil} + x'(y).\overline{x}\,x.\,\mathbf{nil})$$

p' and q' are not bisimilar. In fact, if the name x is received on z the agent p' becomes the agent $\overline{x}\,x.\,\mathbf{nil} \mid x(y).\,\mathbf{nil}$ that can perform an internal synchronization τ; q' on the other hand cannot perform the same action τ after receiving x as input on z.

The problem illustrated by the previous example is due to aliasing, and it appears often in programming languages with both global variables and parameter passing to procedures. It can be solved by defining a finer relation between agents called *strong early full bisimilarity* and defined as follows:

$$p \sim_E q \Leftrightarrow p\sigma \stackrel{\circ}{\sim}_E q\sigma \text{ for every substitution } \sigma$$

where a substitution σ is a function from names to names that is equal to the identity function almost everywhere (i.e. it differs from the identity function only on a finite number of elements of the domain). It is possible to define *strong late full bisimilarity* in a similar way.

12.4.4. Weak Early and Late Ground Bisimulations

As for CCS, we can define the weak versions of bisimulation relations. The definition of weak early ground bisimulation is the following:

$$\forall p,q.\quad p\ S\ q \Rightarrow \begin{cases} \forall \alpha, p'.\ \text{if } p \xrightarrow{\alpha} p' \text{ with } \alpha \neq x(y)\ \wedge\ bn(\alpha) \cap fn(q) = \varnothing,\ \text{then } \exists q'.\ q \stackrel{\alpha}{\Longrightarrow} q' \text{ and } p'\ S\ q' \\ \forall x, y, p'.\ \text{if } p \xrightarrow{x(y)} p' \text{ with } y \notin fn(q),\ \text{then } \forall w.\ \exists q'.\ q \stackrel{x(y)}{\Longrightarrow} q' \text{ and } p'\{^w/_y\}\ S\ q'\{^w/_y\} \\ \text{(and vice versa)} \end{cases}$$

where here α could be τ. So we define the corresponding bisimilarity as follows:

$$p \stackrel{\bullet}{\approx}_E q \text{ iff } p\ S\ q \text{ for some weak early ground bisimulation } S.$$

The late version of the weak ground bisimulation is the following:

$$\forall p,q.\quad p\ S\ q \Rightarrow \begin{cases} \forall \alpha, p'.\ \text{if } p \xrightarrow{\alpha} p' \text{ with } \alpha \neq x(y)\ \wedge\ bn(\alpha) \cap fn(q) = \varnothing,\ \text{then } \exists q'.\ q \stackrel{\alpha}{\Longrightarrow} q' \text{ and } p'\ S\ q' \\ \forall x, y, p'.\ \text{if } p \xrightarrow{x(y)} p' \text{ with } y \notin fn(q),\ \text{then } \exists q'.\ q \stackrel{x(y)}{\Longrightarrow} q' \text{ and } \forall w.\ p'\{^w/_y\}\ S\ q'\{^w/_y\} \\ \text{(and vice versa)} \end{cases}$$

So we define the corresponding bisimilarity as follow:

$$p \overset{\bullet}{\approx}_L q \text{ iff } p\ S\ q \text{ for some weak late ground bisimulation } S.$$

As in the strong case, weak ground bisimilarities are not congruences due to aliasing. In addition, weak bisimilarities are not congruences for a + context, as it was already the case for CCS. Both problems can be fixed by combining the solutions we have shown for weak CCS and for π-calculus strong ground bisimilarities.

Part V.

Appendices

A. Summary

A.1. Induction rules 3.1.2

A.1.1. Noether

Let $\prec$ be a well-founded relation over the set A and let P be a unary predicate over A. Then:

$$\frac{\forall a \in A.\ (\forall b \prec a.\ P(b)) \rightarrow P(a)}{\forall a \in A.P(a)}$$

A.1.2. Weak Mathematical Induction 3.1.3

Let P be a unary predicate over ω.

$$\frac{P(0) \quad \forall n \in \omega.(P(n) \rightarrow P(n+1))}{\forall n \in \omega.P(n)}$$

A.1.3. Strong Mathematical Induction 3.1.4

Let P be a unary predicate over ω.

$$\frac{P(0) \quad \forall n \in \omega.(\forall i \leq n.P(i)) \rightarrow P(n+1)}{\forall n \in \omega.P(n)}$$

A.1.4. Structural Induction 3.1.5

Let Σ be a signature, T_Σ be the set of terms over Σ and P be a property defined on T_Σ.

$$\frac{\forall t \in T_\Sigma.\ (\forall t' < t.\ P(t')) \Rightarrow P(t)}{\forall t \in T_\Sigma.\ P(t)}$$

A.1.5. Derivation Induction 3.1.6

Let R be a set of inference rules and D the set of derivations defined on R. We define:

$$\frac{\forall \{x_1, \ldots, x_n\}/y \in R.\ (P(d_1) \wedge \ldots \wedge P(d_n)) \Rightarrow P(\{d_1, \ldots, d_n\}/y)}{\forall d \in D.\ P(d)}$$

where $d_1, \ldots, d_n$ are derivation for $x_1, \ldots, x_n$.

A.1.6. Rule Induction 3.1.7

Let R be a set of rules and I_R the set of theorems of R.

$$\frac{\forall (X/y) \in R \quad (X \subseteq I_R \quad \forall x \in X.\ P(x)) \Longrightarrow P(y)}{\forall x \in I_R.P(x)}$$

A.1.7. Computational Induction 5.4

Let P be a property, $(D, \sqsubseteq)$ a $CPO_\perp$ and F a monotone, continuous function on it. We define:

$$\frac{P \text{ inclusive} \quad \perp \in P \quad \forall d \in D.d \in P \Longrightarrow F(d) \in P}{fix(F) \in P}$$

A.2. IMP 2

A.2.1. IMP Syntax 2.1

$$\begin{array}{rcl} a & ::= & n \mid x \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1 \\ b & ::= & v \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \vee b_1 \mid b_0 \wedge b_1 \\ c & ::= & \textbf{skip} \mid x := a \mid c_0; c_1 \mid \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 \mid \textbf{while } b \textbf{ do } c \end{array}$$

A.2.2. IMP Operational Semantics 2.2

A.2.2.1. IMP Arithmetic Expressions

$$\frac{}{\langle x, \sigma\rangle \to \sigma(x)}\ (\text{ide}) \qquad \frac{}{\langle n, \sigma\rangle \to n}\ (\text{num}) \qquad \frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 + a_1, \sigma\rangle \to n_0 + n_1}\ (\text{sum})$$

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 - a_1, \sigma\rangle \to n_0 - n_1}\ (\text{dif}) \qquad \frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 \times a_1, \sigma\rangle \to n_0 \times n_1}\ (\text{prod})$$

A.2.2.2. IMP Boolean Expressions

$$\frac{}{\langle v, \sigma\rangle \to v}\ (\text{bool}) \qquad \frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 = a_1, \sigma\rangle \to (n_0 = n_1)}\ (\text{equ}) \qquad \frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1}{\langle a_0 \leq a_1, \sigma\rangle \to (n_0 \leq n_1)}\ (\text{leq})$$

$$\frac{\langle b, \sigma\rangle \to v}{\langle \neg b, \sigma\rangle \to \neg v}\ (\text{not}) \qquad \frac{\langle b_0, \sigma\rangle \to v_0 \quad \langle b_1, \sigma\rangle \to v_1}{\langle b_0 \vee b_1, \sigma\rangle \to (v_0 \vee v_1)}\ (\text{or}) \qquad \frac{\langle b_0, \sigma\rangle \to v_0 \quad \langle b_1, \sigma\rangle \to v_1}{\langle b_0 \wedge b_1, \sigma\rangle \to (v_0 \wedge v_1)}\ (\text{and})$$

A.2.2.3. IMP Commands

$$\frac{}{\langle \textbf{skip}, \sigma\rangle \to \sigma}\ (\text{skip}) \qquad \frac{\langle a, \sigma\rangle \to m}{\langle x := a, \sigma\rangle \to \sigma[{}^{m}/_{x}]}\ (\text{assign})$$

$$\frac{\langle c_0, \sigma\rangle \to \sigma'' \quad \langle c_1, \sigma''\rangle \to \sigma'}{\langle c_0; c_1, \sigma\rangle \to \sigma'}\ (\text{seq}) \qquad \frac{\langle b, \sigma\rangle \to \textbf{true} \quad \langle c_0, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'}\ (\text{iftt}) \qquad \frac{\langle b, \sigma\rangle \to \textbf{false} \quad \langle c_1, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \to \sigma'}\ (\text{ifff})$$

$$\frac{\langle b, \sigma\rangle \to \textbf{true} \quad \langle c, \sigma\rangle \to \sigma'' \quad \langle \textbf{while } b \textbf{ do } c, \sigma''\rangle \to \sigma'}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'}\ (\text{whtt}) \qquad \frac{\langle b, \sigma\rangle \to \textbf{false}}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma}\ (\text{whff})$$

A.2.3. IMP Denotational Semantics 5

A.2.3.1. IMP Arithmetic Expressions $\mathscr{A} : Aexpr \to (\Sigma \to \mathbb{N})$

$$\begin{array}{rcl} \mathscr{A}[\![n]\!]\sigma & = & n \\ \mathscr{A}[\![x]\!]\sigma & = & \sigma x \\ \mathscr{A}[\![a_0 + a_1]\!]\sigma & = & (\mathscr{A}[\![a_0]\!]\sigma) + (\mathscr{A}[\![a_1]\!]\sigma) \\ \mathscr{A}[\![a_0 - a_1]\!]\sigma & = & (\mathscr{A}[\![a_0]\!]\sigma) - (\mathscr{A}[\![a_1]\!]\sigma) \\ \mathscr{A}[\![a_0 \times a_1]\!]\sigma & = & (\mathscr{A}[\![a_0]\!]\sigma) \times (\mathscr{A}[\![a_1]\!]\sigma) \end{array}$$

A.2.3.2. IMP Boolean Expressions $\mathscr{B} : Bexpr \to (\Sigma \to \mathbb{B})$

$$\begin{aligned}
\mathscr{B}[\![v]\!]\sigma &= v \\
\mathscr{B}[\![a_0 = a_1]\!]\sigma &= (\mathscr{A}[\![a_0]\!]\sigma) = (\mathscr{A}[\![a_1]\!]\sigma) \\
\mathscr{B}[\![a_0 \leq a_1]\!]\sigma &= (\mathscr{A}[\![a_0]\!]\sigma) \leq (\mathscr{A}[\![a_1]\!]\sigma) \\
\mathscr{B}[\![\neg b_0]\!]\sigma &= \neg(\mathscr{B}[\![b]\!]\sigma) \\
\mathscr{B}[\![b_0 \vee b_1]\!]\sigma &= (\mathscr{B}[\![b_0]\!]\sigma) \vee (\mathscr{B}[\![b_1]\!]\sigma) \\
\mathscr{B}[\![b_0 \wedge b_1]\!]\sigma &= (\mathscr{B}[\![b_0]\!]\sigma) \wedge (\mathscr{B}[\![b_1]\!]\sigma)
\end{aligned}$$

A.2.3.3. IMP Commands $\mathscr{C} : Com \to (\Sigma \rightharpoonup \Sigma)$

$$\begin{aligned}
\mathscr{C}[\![\mathbf{skip}]\!]\sigma &= \sigma \\
\mathscr{C}[\![x := a]\!]\sigma &= \sigma\left[{}^{\mathscr{A}[\![a]\!]\sigma}/_{x}\right] \\
\mathscr{C}[\![c_0; c_1]\!]\sigma &= \mathscr{C}[\![c_1]\!]^*(\mathscr{C}[\![c_0]\!]\sigma) \\
\mathscr{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!]\sigma &= \mathscr{B}[\![b]\!]\sigma \to \mathscr{C}[\![c_0]\!]\sigma, \mathscr{C}[\![c_1]\!]\sigma \\
\mathscr{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] &= fix\,\Gamma = \bigsqcup_{n\in\omega} \Gamma^n(\perp_{\Sigma\to\Sigma_\perp})
\end{aligned}$$

where $\Gamma : (\Sigma \to \Sigma_\perp) \to \Sigma \to \Sigma_\perp$ is defined as follows:

$$\Gamma\varphi\sigma \quad = \quad \mathscr{B}[\![b]\!]\sigma \to \varphi^*(\mathscr{C}[\![c]\!]\sigma), \sigma$$

A.3. HOFL 6.1

A.3.1. HOFL Syntax 6.1

$t ::=$	$x \mid$		variables
	$n \mid$		constants
	$t_0 + t_1 \mid$	$t_0 - t_1 \mid t_0 \times t_1 \mid$	arithmetic operators
	$\mathbf{if}\ t\ \mathbf{then}\ t_0\ \mathbf{else}\ t_1 \mid$		conditional
	$(t_0, t_1) \mid \mathbf{fst}(t) \mid \mathbf{snd}(t) \mid$		pairing and projection operators
	$\lambda x.t \mid (t_0\ t_1) \mid$		function abstraction and application
	$\mathbf{rec}\ x.t$		recursive definition

A.3.2. HOFL Types 6.1.1

$$\tau ::= int \mid \tau * \tau \mid \tau \to \tau$$

A.3.3. HOFL Typing Rules 6.1.1

$$n : int \qquad \frac{t_0 : int \quad t_1 : int}{t_0\ \mathrm{op}\ t_1 : int}\ \text{with op} = +, -, \text{x} \qquad \frac{t_0 : int \quad t_1 : \tau \quad t_2 : \tau}{\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 : \tau}$$

$$\frac{t_0 : \tau_0 \quad t_1 : \tau_1}{(t_0, t_1) : \tau_0 * \tau_1} \qquad \frac{t : \tau_0 * \tau_1}{\mathbf{fst}(t) : \tau_0} \qquad \frac{t : \tau_0 * \tau_1}{\mathbf{snd}(t) : \tau_1}$$

$$\frac{x : \tau_0 \quad t : \tau_1}{\lambda x.t : \tau_0 \to \tau_1} \qquad \frac{t_1 : \tau_0 \to \tau_1 \qquad t_0 : \tau_0}{(t_1\ t_0) : \tau_1}$$

$$\frac{x : \tau \quad t : \tau}{\mathbf{rec}\ x.t : \tau}$$

A.3.4. HOFL Operational Semantics 6.2

A.3.4.1. HOFL Canonical Forms

$$\frac{\varnothing}{n \in C_{int}} \qquad \frac{t_0 : \tau_0 \quad t_1 : \tau_1 \quad t_0, t_1 \text{ closed}}{(t_0, t_1) \in C_{\tau_0 * \tau_1}} \qquad \frac{\lambda x.t : \tau_0 \to \tau_1 \quad \lambda x.t \text{ closed}}{\lambda x.t \in C_{\tau_0 \to \tau_1}}$$

A.3.4.2. HOFL Axiom

$$\frac{}{c \to c}$$

A.3.4.3. HOFL Arithmetic and Conditional Expressions

$$\frac{t_0 \to n_0 \quad t_1 \to n_1}{t_0 \text{ op } t_1 \to n_0 \underline{op} n_1} \qquad \frac{t \to 0 \quad t_0 \to c_0}{\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \to c_0} \qquad \frac{t \to n \quad n \neq 0 \quad t_1 \to c_1}{\textbf{if } t \textbf{ then } t_0 \textbf{ else } t_1 \to c_1}$$

A.3.4.4. HOFL Pairing Rules

$$\frac{t \to (t_0, t_1) \quad t_0 \to c_0}{\textbf{fst}(t) \to c_0} \qquad \frac{t \to (t_0, t_1) \quad t_1 \to c_1}{\textbf{snd}(t) \to c_1}$$

A.3.4.5. HOFL Function Application

$$\frac{t_1 \to \lambda x.t_1' \quad t_1'[{}^{t_0}/_x] \to c}{(t_1 \quad t_0) \to c} \quad \text{(lazy)}$$

$$\frac{t_1 \to \lambda x.t_1' \quad t_0 \to c_0 \quad t_1'[{}^{c_0}/_x] \to c}{(t_1 \quad t_0) \to c} \quad \text{(eager)}$$

A.3.4.6. HOFL Recursion

$$\frac{t[{}^{\textbf{rec } x.t}/_x] \to c}{\textbf{rec } x.t \to c}$$

A.3.5. HOFL Denotational Semantics 8 $[\![t : \tau]\!] : Env \longrightarrow (V_\tau)_\perp$

$$\begin{aligned}
[\![n]\!]\rho &= \lfloor n \rfloor \\
[\![x]\!]\rho &= \rho x \\
[\![t_0 \textbf{ op } t_1]\!]\rho &= [\![t_0]\!]\rho \; \underline{op}_\perp \; [\![t_1]\!]\rho \\
[\![\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2]\!]\rho &= Cond\,([\![t_0]\!]\rho, [\![t_1]\!]\rho, [\![t_2]\!]\rho) \\
[\![(t_0, t_1)]\!]\rho &= \lfloor ([\![t_0]\!]\rho, [\![t_1]\!]\rho) \rfloor \\
[\![\textbf{fst}(t)]\!]\rho &= \textbf{let } v \Leftarrow [\![t]\!]\rho. \quad \pi_1 v \\
[\![\textbf{snd}(t)]\!]\rho &= \textbf{let } v \Leftarrow [\![t]\!]\rho. \quad \pi_2 v \\
[\![\lambda x.t]\!]\rho &= \left\lfloor \lambda d.\, [\![t]\!]\rho[{}^d/_x] \right\rfloor \\
[\![(t_1 \; t_0)]\!] &= \textbf{let } \varphi \Leftarrow [\![t_1]\!]\rho.\varphi([\![t_0]\!]\rho) \\
[\![\textbf{rec } \; x.t]\!]\rho &= \textbf{fix } \lambda d.\, [\![t]\!]\rho[{}^d/_x]
\end{aligned}$$

A.4. CCS 10

A.4.1. CCS Syntax 10.1

$$p, q ::= x \quad | \quad \mathbf{nil} \quad | \quad \mu.p \quad | \quad p\backslash\alpha \quad | \quad p[\phi] \quad | \quad p + p \quad | \quad p \mid p \quad | \quad \mathbf{rec}\, x.p$$

A.4.2. CCS Operational Semantics 10.2

$$\text{(Act)} \quad \frac{}{\mu.p \xrightarrow{\mu} p} \qquad \text{(Res)} \quad \frac{p \xrightarrow{\mu} q}{p\backslash\alpha \xrightarrow{\mu} q\backslash\alpha}\ \mu \neq \alpha, \bar{\alpha} \qquad \text{(Rel)} \quad \frac{p \xrightarrow{\mu} q}{p[\phi] \xrightarrow{\phi(\mu)} q[\phi]}$$

$$\text{(Sum)} \quad \frac{p \xrightarrow{\mu} p'}{p + q \xrightarrow{\mu} p'} \qquad \frac{q \xrightarrow{\mu} q'}{p + q \xrightarrow{\mu} q'}$$

$$\text{(Com)} \quad \frac{p \xrightarrow{\mu} p'}{p|q \xrightarrow{\mu} p'|q} \qquad \frac{q \xrightarrow{\mu} q'}{p|q \xrightarrow{\mu} p|q'} \qquad \frac{p_1 \xrightarrow{\lambda} p_2 \quad q_1 \xrightarrow{\bar{\lambda}} q_2}{p_1|q_1 \xrightarrow{\tau} p_2|q_2}$$

$$\text{(Rec)} \quad \frac{p[\mathbf{rec}\, x.p/x] \xrightarrow{\mu} q}{\mathbf{rec}\, x.p \xrightarrow{\mu} q}$$

A.4.3. CCS Abstract Semantics 10.3

A.4.3.1. CCS Strong Bisimulation 10.3.3

$$\forall p, q. \quad p\ \Phi(R)\ q \stackrel{\text{def}}{=} \begin{cases} \forall \mu, p'. \quad p \xrightarrow{\mu} p' \quad \text{implies} \quad \exists q'.\ q \xrightarrow{\mu} q' \quad \text{and} \quad p' R q' \\ \forall \mu, q'. \quad q \xrightarrow{\mu} q' \quad \text{implies} \quad \exists p'.\ p \xrightarrow{\mu} p' \quad \text{and} \quad p' R q' \end{cases}$$

A.4.3.2. CCS Weak Bisimulation 10.7

The weak transition relation $\Rightarrow$is defined as follows:

$$p \stackrel{\tau}{\Rightarrow} q \quad \text{iff} \quad p \xrightarrow{\tau} \ldots \xrightarrow{\tau} q \ \vee \ p = q$$
$$p \stackrel{\lambda}{\Rightarrow} q \quad \text{iff} \quad p \stackrel{\tau}{\Rightarrow} p' \xrightarrow{\lambda} q' \stackrel{\tau}{\Rightarrow} q$$

A weak bisimulation is defined as follows:

$$\forall p, q. \quad p\ \Psi(R)\ q \stackrel{\text{def}}{=} \begin{cases} \forall \mu, p'. \quad p \xrightarrow{\mu} p' \quad \text{implies} \quad \exists q'.\ q \stackrel{\mu}{\Rightarrow} q' \quad \text{and} \quad p'\ R\ q' \\ \forall \mu, q'. \quad q \xrightarrow{\mu} q' \quad \text{implies} \quad \exists p'.\ p \stackrel{\mu}{\Rightarrow} p' \quad \text{and} \quad p'\ R\ q' \end{cases}$$

A.4.3.3. CCS Observational Congruence 10.7.2

$$\forall p, q. \quad p \cong q \stackrel{\text{def}}{=} \begin{cases} \forall p'. \quad p \longrightarrow p' \quad \text{implies} \quad \exists q'.\ q \stackrel{\tau}{\longrightarrow\!\!\!\!\Rightarrow} q' \quad \text{and} \quad p' \approx q' \\ \forall \lambda, p'. \quad p \xrightarrow{\lambda} p' \quad \text{implies} \quad \exists q'.\ q \stackrel{\lambda}{\Longrightarrow} q' \quad \text{and} \quad p' \approx q' \\ \text{(and vice versa)} \end{cases}$$

Alternatively $\cong$ can be defined as follows:

$$p \cong q \ \text{ iff } \ p \approx q \ \wedge \ \forall r.\ p + r \approx q + r$$

A.4.3.4. CCS Axioms for Observational Congruence (Milner τ Laws) 10.7.2

$$
\begin{aligned}
&p + \tau.p = \tau.p \\
&\mu.(p + \tau.q) = \mu.(p + \tau.q) + \mu.q \\
&\mu.\tau.p = \mu.p
\end{aligned}
$$

A.4.3.5. CCS Dynamic Bisimulation 10.7.3

$$
\forall p, q. \quad p \; \Theta(R) \; q \stackrel{\text{def}}{=} \begin{cases} \forall p'. \quad p \xrightarrow{\tau} p' \quad \text{implies} \quad \exists q'.\, q \xrightarrow{\tau} \stackrel{\tau}{\Longrightarrow} q' \quad \text{and} \quad p' \; R \; q' \\ \forall \lambda, p'. \quad p \xrightarrow{\lambda} p' \quad \text{implies} \quad \exists q'.\, q \stackrel{\lambda}{\Longrightarrow} q' \quad \text{and} \quad p' \; R \; q' \\ \text{(and vice versa)} \end{cases}
$$

A.4.3.6. CCS Axioms for Dynamic Bisimilarity 10.7.3

$$
\begin{aligned}
&p + \tau p = \tau p \\
&\mu(p + \tau q) = \mu(p + \tau q) + \mu q
\end{aligned}
$$

A.5. Temporal and Modal Logic

A.5.1. Hennessy - Milner Logic 10.5

The syntax of Hennessy-Milner logic formulas is:

$$
F ::= true \quad | \quad false \quad | \quad \wedge_{i \in I} F_i \quad | \quad \vee_{i \in I} F_i \quad | \quad \Diamond_\mu F \quad | \quad \Box_\mu F
$$

The *satisfaction relation* $\models \subseteq P \times \mathscr{L}$ is defined as follows:

$$
\begin{aligned}
&p \models true \\
&p \models \wedge_{i \in I} F_i \quad \text{iff} \quad \forall i \in I.\; p \models F_i \\
&p \models \vee_{i \in I} F_i \quad \text{iff} \quad \exists i \in I.\; p \models F_i \qquad \forall i \in I \\
&p \models \Diamond_\mu F \quad \text{iff} \quad \exists p'.\; p \xrightarrow{\mu} p' \wedge p' \models F \\
&p \models \Box_\mu F \quad \text{iff} \quad \forall p'.\; p \xrightarrow{\mu} p' \Rightarrow p' \models F
\end{aligned}
$$

A.5.2. Linear Temporal Logic 11.1.1

The syntax of LTL is as follows:

$$
\phi ::= true \quad | \quad false \quad | \quad p \quad | \quad \neg\phi \quad | \quad \phi_1 \wedge \phi_2 \quad | \quad \phi_1 \vee \phi_2 \quad | \quad O\phi \quad | \quad F\phi \quad | \quad G\phi \quad | \quad \phi_1 U \phi_2
$$

Let P be a set of atomic propositions and $S : P \to 2^\omega$. We define the satisfaction operator $\models$ as follows:

$$
\begin{aligned}
S \;&\models\; true \\
S \;&\models\; p \quad \text{if } 0 \in S(p) \\
S \;&\models\; \neg\phi \quad \text{if it is not true that } S \models \phi \\
S \;&\models\; \phi_1 \wedge \phi_2 \quad \text{if } S \models \phi_1 \text{ and } S \models \phi_2 \\
S \;&\models\; \phi_1 \vee \phi_2 \quad \text{if } S \models \phi_1 \text{ or } S \models \phi_2 \\
S \;&\models\; O\phi \quad \text{if } S^1 \models \phi \\
S \;&\models\; F\phi \quad \text{if } \exists i \in \mathbb{N} \text{ such that } S^i \models \phi \\
S \;&\models\; G\phi \quad \text{if } \forall i \in \mathbb{N} \text{ it holds } S^i \models \phi \\
S \;&\models\; \phi_1 U \phi_2 \quad \text{if } \exists i \in \mathbb{N} \text{ such that } S^i \models \phi_2 \text{ and } \forall j < i.\; S^j \models \phi_1
\end{aligned}
$$

A.5.3. Computation Tree Logic 11.1.2

The syntax of CTL is as follows:

$$\phi ::= true \mid false \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid O\phi \mid F\phi \mid G\phi \mid \phi_1 U \phi_2 \mid E\phi \mid A\phi$$

Let (T, S, P) be a branching structure and $\pi = v_0, v_1, \ldots, v_n, \ldots$ be an infinite path. We define the $\models$ relation for as follows:

state operators:

$$\begin{array}{lcl}
S,\pi & \models & true \\
S,\pi & \models & p \quad \text{if } v_0 \in S(p) \\
S,\pi & \models & \neg\phi \quad \text{if it is not true that } S,\pi \models \phi \\
S,\pi & \models & \phi_1 \wedge \phi_2 \quad \text{if } S,\pi \models \phi_1 \text{ and } S,\pi \models \phi_2 \\
S,\pi & \models & \phi_1 \vee \phi_2 \quad \text{if } S,\pi \models \phi_1 \text{ or } S,\pi \models \phi_2 \\
S,\pi & \models & O\phi \quad \text{if } S,\pi^1 \models \phi \\
S,\pi & \models & F\phi \quad \text{if } \exists i \in \omega \text{ such that } S,\pi^i \models \phi \\
S,\pi & \models & G\phi \quad \text{if } \forall i \in \omega \text{ it holds } S,\pi^i \models \phi \\
S,\pi & \models & \phi_1 U \phi_2 \quad \text{if } \exists i \in \omega \text{ such that } S,\pi^i \models \phi_2 \text{ and for all } j < i.\ S,\pi^j \models \phi_1
\end{array}$$

path operators

$$\begin{array}{lcl}
S,\pi & \models & E\phi \quad \text{if there exists } \pi_1 = v_0, v_1', \ldots, v_n', \ldots \text{ such that } S,\pi_1 \models \phi \\
S,\pi & \models & A\phi \quad \text{if for all paths } \pi_1 = v_0, v_1', \ldots, v_n', \ldots \text{ we have } S,\pi_1 \models \phi
\end{array}$$

This semantics apply both for CTL and CTL^*. The formulas of CTL are obtained by restricting CTL^*: a CTL^* formula is a CTL formula if the followings hold:

- A and E appear only immediately before a linear operator (i.e., F,G,U and O).
- each linear operator appears immediately after a quantifier (i.e., A and E).

A.6. μ-Calculus 11.2

The syntax of μ-calculus is as follows:

$$\phi ::= true \mid false \mid x \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \Diamond\phi \mid \Box\phi \mid \mu x.\phi \mid \nu x.\phi$$

We define the denotational semantics of μ-calculus which maps each predicate to the subset of states in which it holds as follows:

$$\begin{array}{rcl}
[\![true]\!]\rho & = & V \\
[\![false]\!]\rho & = & \varnothing \\
[\![x]\!]\rho & = & \rho x \\
[\![p]\!]\rho & = & \rho p \\
[\![\neg\phi]\!]\rho & = & V \setminus [\![\phi]\!]\rho \\
[\![\phi_1 \wedge \phi_2]\!]\rho & = & [\![\phi_1]\!]\rho \cap [\![\phi_2]\!]\rho \\
[\![\phi_1 \vee \phi_2]\!]\rho & = & [\![\phi_1]\!]\rho \cup [\![\phi_2]\!]\rho \\
[\![\Diamond\phi]\!]\rho & = & \{\ v \mid \exists v'.v \to v' \ \wedge\ v' \in [\![\phi]\!]\rho\ \} \\
[\![\Box\phi]\!]\rho & = & \{\ v \mid \forall v'.v \to v' \Rightarrow v' \in [\![\phi]\!]\rho\ \} \\
[\![\mu x.\phi]\!]\rho & = & \text{fix}\,\lambda S.\,[\![\phi]\!]\rho[^S/_x] \\
[\![\nu x.\phi]\!]\rho & = & \text{Fix}\,\lambda S.\,[\![\phi]\!]\rho[^S/_x]
\end{array}$$

A.7. π-calculus 12

A.7.1. π-calculys Syntax 12.1

$$\begin{array}{lcl} p & ::= & \mathbf{nil} \mid \alpha.p \mid [x = y]p \mid p + p \mid p|p \mid (y)p \mid !p \\ \alpha & ::= & \tau \mid x(y) \mid \overline{x}y \end{array}$$

A.7.2. π-calculus Operational Semantics 12.2

$$\text{(Tau)}\ \frac{}{\tau.p \xrightarrow{\tau} p} \qquad \text{(Out)}\ \frac{}{\overline{x}\,y.p \xrightarrow{\overline{x}\,y} p} \qquad \text{(In)}\ \frac{}{x(y).p \xrightarrow{x(w)} p\{^w/_y\}}\ w \notin fn((y)p)$$

$$\text{(SumL)}\ \frac{p \xrightarrow{\alpha} p'}{p + q \xrightarrow{\alpha} p'} \qquad \text{(SumR)}\ \frac{q \xrightarrow{\alpha} q'}{p + q \xrightarrow{\alpha} q'} \qquad \text{(Match)}\ \frac{p \xrightarrow{\alpha} p'}{[x = x]p \xrightarrow{\alpha} p'}$$

$$\text{(ParL)}\ \frac{p \xrightarrow{\alpha} p'}{p|q \xrightarrow{\alpha} p'|q}\ bn(\alpha) \cap fn(q) = \varnothing \qquad \text{(ParR)}\ \frac{q \xrightarrow{\alpha} q'}{p|q \xrightarrow{\alpha} p|q'}\ bn(\alpha) \cap fn(p) = \varnothing$$

$$\text{(ComL)}\ \frac{p \xrightarrow{\overline{x}\,z} p' \quad q \xrightarrow{x(y)} q'}{p|q \xrightarrow{\tau} p'|(q'\{^z/_y\})} \qquad \text{(ComR)}\ \frac{p \xrightarrow{x(y)} p' \quad q \xrightarrow{\overline{x}\,z} q'}{p|q \xrightarrow{\tau} p'\{^z/_y\}|q'}$$

$$\text{(Res)}\ \frac{p \xrightarrow{\alpha} p'}{(y)p \xrightarrow{\alpha} (y)p'}\ y \notin n(\alpha) \qquad \text{(Open)}\ \frac{p \xrightarrow{\overline{x}\,y} p'}{(y)p \xrightarrow{\overline{x}(w)} p'\{^w/_y\}}\ y \neq x \quad w \notin fn((y)p)$$

$$\text{(CloseL)}\ \frac{p \xrightarrow{\overline{x}\,(w)} p' \quad q \xrightarrow{x(w)} q'}{p|q \xrightarrow{\tau} (w)(p'|q')} \qquad \text{(CloseR)}\ \frac{p \xrightarrow{x(w)} p' \quad q \xrightarrow{\overline{x}\,(w)} q'}{p|q \xrightarrow{\tau} (w)(p'|q')} \qquad \text{(Rep)}\ \frac{p|!p \xrightarrow{\alpha} p'}{!p \xrightarrow{\alpha} p'}$$

A.7.3. π-calculus Abstract Semantics 12.4

A.7.3.1. Strong Early Ground Bisimulation 12.4.1

$$\forall p, q.\quad p\ S\ q \Rightarrow \begin{cases} \forall \alpha, p'.\ \text{if } p \xrightarrow{\alpha} p' \text{ with } \alpha \neq x(y)\ \wedge\ bn(\alpha) \cap fn(q) = \varnothing,\ \text{then } \exists q'.\ q \xrightarrow{\alpha} q' \text{ and } p'\ S\ q' \\ \forall x, y, p'.\ \text{if } p \xrightarrow{x(y)} p' \text{ with } y \notin fn(q),\ \text{then } \forall w.\ \exists q'.\ q \xrightarrow{x(y)} q' \text{ and } p'\{^w/_y\}\ S\ q'\{^w/_y\} \\ \text{(and vice versa)} \end{cases}$$

We define the strong early ground bisimilarity as follows:

$$p \overset{\circ}{\sim}_E q \Leftrightarrow p\ S\ q \text{ for some strong early ground bisimulation } S$$

A.7.3.2. Strong Early Full Bisimilarity 12.4.3

$$p \sim_E q \Leftrightarrow p\sigma \overset{\circ}{\sim}_E q\sigma \ \text{ for every substitution } \sigma$$

A.7.3.3. Strong Late Ground Bisimulation 12.4.2

$$\forall p,q.\quad p\ S\ q \Rightarrow \begin{cases} \forall \alpha, p'.\ \text{if } p \xrightarrow{\alpha} p' \text{ with } \alpha \neq x(y)\ \wedge\ bn(\alpha)\cap fn(q)=\varnothing,\ \text{then } \exists q'.\ q \xrightarrow{\alpha} q' \text{ and } p'\ S\ q' \\ \forall x, y, p'.\ \text{if } p \xrightarrow{x(y)} p' \text{ with } y \notin fn(q),\ \text{then } \exists q'.\ \forall w.\ q \xrightarrow{x(y)} q' \text{ and } p'\{^w/_y\}\ S\ q'\{^w/_y\} \\ \text{(and vice versa)} \end{cases}$$

We define the strong late ground bisimilarity as follows:

$$p \overset{\circ}{\sim}_L q \Leftrightarrow p\ S\ q \text{ for some late early ground bisimulation } S$$

A.7.3.4. Strong Late Full Bisimilarity 12.4.3

$$p \sim_L q \Leftrightarrow p\sigma \overset{\circ}{\sim}_L q\sigma \text{ for every substitution } \sigma$$

A.7.3.5. Weak Early Ground Bisimulation 12.4.4

$$\forall p,q.\quad p\ S\ q \Rightarrow \begin{cases} \forall \alpha, p'.\ \text{if } p \xrightarrow{\alpha} p' \text{ with } \alpha \neq x(y)\ \wedge\ bn(\alpha)\cap fn(q)=\varnothing,\ \text{then } \exists q'.\ q \overset{\alpha}{\Longrightarrow} q' \text{ and } p'\ S\ q' \\ \forall x, y, p'.\ \text{if } p \xrightarrow{x(y)} p' \text{ with } y \notin fn(q),\ \text{then } \forall w.\ \exists q'.\ q \overset{x(y)}{\Longrightarrow} q' \text{ and } p'\{^w/_y\}\ S\ q'\{^w/_y\} \\ \text{(and vice versa)} \end{cases}$$

where here α could be τ. We define the corresponding bisimilarity as follows:

$$p \overset{\bullet}{\approx}_E q \text{ iff } p\ S\ q \text{ for some weak early ground bisimulation } S.$$

A.7.3.6. Weak Late Ground Bisimulation 12.4.4

$$\forall p,q.\quad p\ S\ q \Rightarrow \begin{cases} \forall \alpha, p'.\ \text{if } p \xrightarrow{\alpha} p' \text{ with } \alpha \neq x(y)\ \wedge\ bn(\alpha)\cap fn(q)=\varnothing,\ \text{then } \exists q'.\ q \overset{\alpha}{\Longrightarrow} q' \text{ and } p'\ S\ q' \\ \forall x, y, p'.\ \text{if } p \xrightarrow{x(y)} p' \text{ with } y \notin fn(q),\ \text{then } \exists q'.\ q \overset{x(y)}{\Longrightarrow} q' \text{ and } \forall w.\ p'\{^w/_y\}\ S\ q'\{^w/_y\} \\ \text{(and vice versa)} \end{cases}$$

We define the corresponding bisimilarity as follow:

$$p \overset{\bullet}{\approx}_L q \text{ iff } p\ S\ q \text{ for some weak late ground bisimulation } S.$$

A.8. LTL for Action, Non-determinism and Probability

Let S be a set of states, T be a set of transitions, L be a set of labels, $D(S)$ and $D(L\times S)$ be respectively the set of discrete probabilistic distributions over S and over $L\times S$.

- CCS: $\alpha : S \longrightarrow \mathscr{P}(L\times S)$
- DTMC: $\alpha : S \longrightarrow (D(S)+1)$
- CTMC: $\alpha : S \to S \to \mathbb{R}$
- Reactive probabilistic transition systems: $\alpha : S \to L \to (D(S)+1)$
- Generative probabilistic transition systems: $\alpha : S \to (D(L\times S)+1)$
- Segala Automata: $\alpha : S \to \mathscr{P}(D(L\times S))$
- Simple Segala Automata: $\alpha : S \to \mathscr{P}(L\times D(S))$

A.9. Larsen-Skou Logic 14.1.1.1

We define the Larsen-Skou satisfaction relation as follows:

$$
\begin{array}{lcl}
s \models \mathbf{true} & & \\
s \models \varphi_1 \wedge \varphi_2 & \Leftrightarrow & s \models \varphi_1 \text{ and } s \models \varphi_2 \\
s \models \neg\varphi & \Leftrightarrow & \neg s \models \varphi \\
s \models \langle l \rangle_q \varphi & \Leftrightarrow & \alpha\, s\, l\, [\![\varphi]\!] \geq q \text{ where } [\![\varphi]\!] = \{s \in S \mid s \models \varphi\}
\end{array}
$$

A.10. PEPA 15

A.10.1. PEPA Syntax 15.2.1

$$P ::= \mathbf{nil} \mid (\alpha, r).P \mid P + P \mid P \underset{L}{\bowtie} P \mid P/L \mid C$$

A.10.2. PEPA Operational Semantics 15.2.2

$$\frac{}{(\alpha, r).P \xrightarrow{(\alpha,r)} P} \qquad \frac{P \xrightarrow{(\alpha,r)} P'}{P + Q \xrightarrow{(\alpha,r)} P'} \qquad \frac{Q \xrightarrow{(\alpha,r)} Q'}{P + Q \xrightarrow{(\alpha,r)} Q'}$$

$$\frac{P \xrightarrow{(\alpha,r)} P' \quad \alpha \notin L}{P/L \xrightarrow{(\alpha,r)} P'/L} \qquad \frac{P \xrightarrow{(\alpha,r)} P' \quad \alpha \in L}{P/L \xrightarrow{(\tau,r)} P'/L}$$

$$\frac{P \xrightarrow{(\alpha,r)} P' \quad \alpha \notin L}{P \underset{L}{\bowtie} Q \xrightarrow{(\alpha,r)} P' \underset{L}{\bowtie} Q} \qquad \frac{Q \xrightarrow{(\alpha,r)} Q' \quad \alpha \notin L}{P \underset{L}{\bowtie} Q \xrightarrow{(\alpha,r)} P \underset{L}{\bowtie} Q'}$$

$$\frac{P \xrightarrow{(\alpha,r_1)} P' \quad Q \xrightarrow{(\alpha,r_2)} Q' \quad \alpha \in L}{P \underset{L}{\bowtie} Q \xrightarrow{(\alpha,r)} P' \underset{L}{\bowtie} Q'} \quad \text{where } r = \min(r_\alpha(P), r_\alpha(Q)) * \frac{r_1}{r_\alpha(P)} * \frac{r_2}{r_\alpha(Q)}$$

$$\frac{P \xrightarrow{(\alpha,r)} P' \quad (C \stackrel{\text{def}}{=} P) \in \Delta}{C \xrightarrow{(\alpha,r)} P'}$$

where we have denoted by $r_\alpha(P)$ the *apparent rate* of action α in P, which is defined by structural recursion as follows:

$$
\begin{array}{rcl}
r_\alpha(\mathbf{nil}) & = & 0 \\
r_\alpha((\beta, r).P) & = & \begin{cases} r & \text{if } \alpha = \beta \\ 0 & \text{if } \alpha \neq \beta \end{cases} \\
r_\alpha(P + Q) & = & r_\alpha(P) + r_\alpha(Q) \\
r_\alpha(P/L) & = & \begin{cases} r_\alpha(P) & \text{if } \alpha \notin L \\ 0 & \text{if } \alpha \in L \end{cases} \\
r_\alpha(P \underset{L}{\bowtie} Q) & = & \begin{cases} min(r_\alpha(P), r_\alpha(Q)) & \text{if } \alpha \in L \\ r_\alpha(P) + r_\alpha(Q) & \text{if } \alpha \notin L \end{cases}
\end{array}
$$