By definition of product there is a morphism $\langle f_i\rangle : \Sigma \to \Pi_{\in I} A_i$ such that $\pi_i \circ \langle f_i\rangle = f_i$. Then we can derive:

$$\pi_i \circ \langle f_i\rangle \circ \bot = f_i \circ \bot = \pi_i \circ x$$
$$\pi_i \circ \langle f_i\rangle \circ \top = f_i \circ \top = \pi_i \circ y \ .$$

Then $\langle f_i\rangle \circ \bot = x$ and $\langle f_i\rangle \circ \top = y$, and A is Σ-linked. In the proof we have only used the hypothesis that x is pointwise less than y. The map $\langle f_i\rangle$ proves that $x \leq_A y$.

(3) If $x \leq_A y$ then there is a morphism $f : \Sigma \to A$ such that $f(\bot) = x$ and $f(\top) = y$. If $x, y : B$ then f can be restricted to $f' : \Sigma \to B$. By monotonicity it follows $x = f'(\bot) \leq_B f'(\top) = y$. Vice versa, suppose $x \leq_B y$, $f : A \to \Sigma$, and $f(x) = \top$, then f can be restricted to B, and by definition of intrinsic ordering $f(y) = \top$. □

Definition 15.5.18 *The pointwise order $\leq_{ext}$ on functions $f, g : A \to B$ is defined as:*

$$f \leq_{ext} g \quad \textit{iff} \quad \forall x : A\,(f \circ x \leq_B g \circ x) \ .$$

The following theorem provides the basic insight into the structure of Σ^A.

Theorem 15.5.19 *Let A be a per. Then: (1) The per Σ^A is separated and Σ-linked.*
(2) The intrinsic order on Σ^A coincides with the pointwise order.
(3) The per Σ^A is complete.

PROOF. We start with the construction of a lub. Let $AS^{ext}(\Sigma^A)$ be the collection of functions $\chi : N \to (\Sigma^A)$ such that $\chi(n) \leq_{ext} \chi(n+1)$, that is the collection of ascending sequences with respect to the pointwise order. We define a function $\sigma : AS^{ext}(\Sigma^A) \to \Sigma^A$ (looking at Σ as $1 \rightharpoonup 1$):

$$\sigma(\chi) = \lambda x : A.\lambda z : 1.\text{if } (\exists n\, \chi(n)(x) \downarrow) \text{ then } \downarrow \ .$$

It is immediately verified that $\sigma(\chi)$ is the lub of χ with respect to the pointwise order.

Next, suppose $f, g : A \to \Sigma$ and $f \leq_{ext} g$. We build $h : \Sigma \to (\Sigma^A)$ such that $h(K) = g$ and $h(K^c) = f$. This will prove that Σ^A is Σ-linked and that the pointwise and intrinsic orders coincide. Consider a family of chains $c(n, i) : A \to \Sigma$ defined as follows (cf. proof theorem 15.5.7):

$$c(n,i)(x) = \begin{cases} g(x) & \text{if } nn \downarrow i \\ f(x) & \text{otherwise} \ . \end{cases}$$

We observe that for all n, $\lambda i.c(n, i) : AS^{ext}(\Sigma^A)$ and that:

$$\sigma(\lambda i.c(n,i)) = \begin{cases} g & \text{if } n \in K \\ f & \text{if } n \in K^c \ . \end{cases}$$

Then we can define h as specified above. By proposition 15.5.14(2), we can conclude that Σ^A is Σ-linked. Finally observe that since pointwise and intrinsic order coincide, the function σ proves that Σ^A is complete. □

Basically the same proof technique is used to prove the following result.

Proposition 15.5.20 *If A is a complete and separated per then A is Σ-linked.*

PROOF. Let $x \leq_A y$ and consider the following family of chains:

$$c(n,i) = \begin{cases} y & \text{if } nn \downarrow i \\ x & \text{otherwise .} \end{cases}$$

Fixed n, the ascending sequence $\lambda i.c(n,i)$ is apparently innocuous, as it can take at most two values. However for a general per we do not know how to compute the lub of this sequence. For complete per's we can use σ_A to define:

$$h(n) = \sigma_A(\lambda i.c(n,i)) .$$

Observe that $h : \{K, K^c\} \to A$ with $h(K) = y$ and $h(K^c) = x$. □

Next we consider a condition stronger than separation and Σ-linkage which is due to [FMRS92].

Definition 15.5.21 (extensional per) *A per A is* extensional *if there is a per B such that $[A] \subseteq [\Sigma^B]$. We denote with* **exper** *the full sub-category of extensional per's.*

Exercise 15.5.22 *Show that the following is an equivalent definition of extensional per. A is an exper if there is $X \subseteq D$ such that $[A] \subseteq [\Sigma^{Diag(X)}]$, where $Diag(X) = \{(x,x) \mid x \in X\}$. Hint: for A per, $[\Sigma^A] \subseteq [\Sigma^{Diag(|A|)}]$.*

Proposition 15.5.23 *Let $[A] \subseteq [\Sigma^B]$ be an extensional per. Then:*
(1) A is separated and Σ-linked.
(2) If $f, g : A$ then $f \leq_A g$ iff $\forall b : B\, (f \circ b \leq_\Sigma g \circ b)$.

PROOF. (1) By proposition 15.4.3, every per Σ^B is separated, and separated per's are closed under subobject. By theorem 15.5.19, Σ^B is Σ-linked and by proposition 15.5.17, Σ-linked per's are closed under subobjects obtained by selecting a subset of the quotient space.

(2) By proposition 15.5.17, $f \leq_A g$ iff $f \leq_{\Sigma^B} g$. By theorem 15.5.19, we know that the order on Σ^B is pointwise. □

Theorem 15.5.24 *The category exper is reflective in the category of separated per's.*

PROOF. We use $A \Rightarrow \Sigma$ as a linear notation for Σ^A. We already know that every extensional per is separated. We define a reflector $L_{ex} : \Sigma\mathbf{per} \to \mathbf{exper}$ as follows:

$$[L_{ex}(A)] = \{[\lambda^* u.ud]_{(A \Rightarrow \Sigma) \Rightarrow \Sigma} \mid d \in |A|\} .$$

This is an exper as by definition $[L_{ex}(A)] \subseteq [(A \Rightarrow \Sigma) \Rightarrow \Sigma]$. The universal morphism $e_A : A \to L_{ex}(A)$ is the one realized by $\lambda^* d.\lambda^* u.ud$. Intuitively it takes an element d to the collection of its neighbourhoods $\lambda^* u.ud$. By construction e_A is an epi, moreover it is also a mono if A is separated. Note that L_{ex} can also work as a reflector from **per** to **exper**.

Next we show that if B is extensional then $e_B : B \to L_{ex}(B)$ is an iso. Suppose $[B] \subseteq [C \Rightarrow \Sigma]$. We take $e_B^{-1} : L_{ex}(B) \to B$ as the morphism realized by $\lambda^* i.\lambda^* c.i(\lambda^* u.uc)$. Let us try to type, semantically, this term. First one can check that:

$$\lambda^* i.\lambda^* c.i(\lambda^* u.uc) \in |((B \Rightarrow \Sigma) \Rightarrow \Sigma) \Rightarrow (C \Rightarrow \Sigma)| \ .$$

Since $B \subseteq (C \Rightarrow \Sigma)$, we can "coerce" u from B to $C \Rightarrow \Sigma$. Since $L_{ex}(B) \subseteq (B \Rightarrow \Sigma) \Rightarrow \Sigma$, we can also type the term as follows:

$$\lambda^* i.\lambda^* c.i(\lambda^* u.uc) \in |L_{ex}(B) \Rightarrow (C \Rightarrow \Sigma)| \ . \tag{15.1}$$

Finally, looking at the definition of $L_{ex}(B)$ we can prove

$$\lambda^* i.\lambda^* c.i(\lambda^* u.uc) \in |L_{ex}(B) \Rightarrow B| \ .$$

Suppose $\theta \, L_{ex}(B) \, \theta'$. Then there is $f \in |B|$ such that

$$\theta \, (B \Rightarrow \Sigma) \Rightarrow \Sigma \, \lambda^* v.vf \, (B \Rightarrow \Sigma) \Rightarrow \Sigma \, \theta' \ .$$

We compute:

$$\begin{aligned} (\lambda^* i.\lambda^* c.i(\lambda^* u.uc))(\lambda^* v.vf) &= \lambda^* c.(\lambda^* v.vf)(\lambda^* u.uc)) \\ &= \lambda^* c.(\lambda^* u.uc)f \\ &= \lambda^* c.fc \ . \end{aligned}$$

From the typing 15.1 we derive:

$$\lambda^* c.\theta(\lambda^* u.uc) \, (C \Rightarrow \Sigma) \, \lambda^* c.fc \, (C \Rightarrow \Sigma) \, \lambda^* c.\theta'(\lambda^* u.uc) \ .$$

Since, $\lambda^* c.fc(C \Rightarrow \Sigma)f$ and $f \in |B|$, it follows $\lambda^* c.\theta(\lambda^* u.uc) \, B \, \lambda^* c.\theta'(\lambda^* u.uc)$. To show that e_B^{-1} is an iso, we compute the realizers:

$$\begin{aligned} (\lambda^* i.\lambda^* c.i(\lambda^* u.uc))(\lambda^* dw.wd)f &= (\lambda^* i.\lambda^* c.i(\lambda^* u.uc))(\lambda^* w.wf) \\ &= \lambda^* c.(\lambda^* w.wf)(\lambda^* u.uc) \\ &= \lambda^* c.fc \ . \end{aligned}$$

Vice versa $(\lambda^* dw.wd)(\lambda^* c.fc) = \lambda^* w.w(\lambda^* c.fc)$. Finally, given $\phi \in |A \Rightarrow B|$ we define $\phi' \in |L_{ex}(A) \Rightarrow L_{ex}(B)|$ as follows:

$$\phi' = \lambda^* i.\lambda^* u.i(\lambda^* a.u(\phi a)) \ .$$

If $f = [\phi]_{A \Rightarrow B}$ set $L_{ex}(f) = [\phi']_{L_{ex}(A) \Rightarrow L_{ex}(B)}$. □

Theorem 15.5.25 *(1) The categories of extensional per's and complete extensional per's are closed under arbitrary intersections.*
*(2) The category of complete extensional per's (***cexper***) is reflective in* **exper.**

PROOF HINT. (1) If $[A_i] \subseteq [\Sigma^{B_i}]$ for $i \in I$, then $[\bigcap_{i\in I} A_i] \subseteq [\Sigma^{\bigcup_{i\in I} B_i}]$. This shows that **exper** is closed under arbitrary intersections. Note that the fixed point combinator σ defined in the proof of theorem 15.5.19 has a realizer that works uniformly on all ascending sequence. This realizer can be used to prove that $\bigcap_{i\in I} A_i$ is complete if the A_i's are complete.

(2) Suppose $[A] \subseteq [\Sigma^B]$. We define the reflection $L_c(A)$ as the least cexper such that:

$$[A] \subseteq L_c(A) \subseteq [\Sigma^B] \; .$$

If $\phi|\!\!-f : A \to B$ for A exper and B cexper then let $L_c(f) = [\phi]_{B^{Lc(A)}}$. We use the fact that realized functions are continuous to show that $L_c(f)$ is well-defined. □

Exercise 15.5.26 *Show that the category of (complete) extensional per's is cartesian closed.*

To summarize we have proven the following reflections when working over the pca $(\omega, \bullet)$:

$$\textbf{cexper} \subset_> \textbf{exper} \subset_> \Sigma\textbf{per} \subset_> \textbf{per} \subset_> \omega\textbf{-set} \; .$$

From left to right: theorem 15.5.25, theorem 15.5.24, theorem 15.4.12, and proposition 15.1.9. The category of complete separated per's can also be shown to be reflective in Σ**per** when appropriately formulated in the internal language of the effective topos [Pho90], however this proof lies outside the realm of our introductive approach to realizability.

15.6 Per's over D_∞

We identify a category of *complete uniform* per's (cuper's), which is a full subcategory of the category of per's when working over a specific D_∞ λ-model.

Definition 15.6.1 *Let D be the initial solution of the equation:*

$$D = (D \rightharpoonup D) + (D \times D)$$

in the category of cpo's and injection-projection pairs where $+$ is the coalesced sum.

We note that in general $in_l : C \to C + C'$ and $in_r : C' \to C + C'$ form the injection part of an injection-projection pair. We define $D_0 = \{\bot\}$ and $D_{n+1} = (D_n \rightharpoonup D_n) + (D_n \times D_n)$ with injection projection pairs $(i_n, j_n) : D_n \to D$.

We remark that D is bifinite. Let $p_n = j_n \circ i_n : D \to D$ be a projection such that $im(p_n) = i_n(D_n)$. We consider the following injection-projection pairs: $(i_\rightharpoonup, j_\rightharpoonup) : (D \rightharpoonup D) \to D$ and $(i_\times, j_\times) : D \times D \to D$. As usual we define for $d, e \in D$:

$$\begin{array}{ll} \langle d, e\rangle & = i_\times(d, e) \\ de & = j_\rightharpoonup(d)(e) \\ d_n & = p_n(d) \; . \end{array}$$

The application de has the properties required for a pca. We will use the following properties (cf. section 3.1):

$$\begin{array}{lll} \langle d, e\rangle_{n+1} & = \langle d_n, e_n\rangle & \\ d_{n+1}e & = d_{n+1}e_n & = (de_n)_n \ . \end{array}$$

Exercise 15.6.2 *Prove the properties above following section 3.1.*

In this section D stands for the domain specified in definition 15.6.1. Whenever we speak of a relation we intend by default a binary relation over D. For $A \in per_D$ we let $A_n = A \cap (im(p_n) \times im(p_n))$. In order to distinguish indexes from approximants we write indexes in superscript position, so d_n^i is the n-th approximant of the i-th element.

Definition 15.6.3 *A relation R is:*
(1) pointed *if* $(\bot_D, \bot_D) \in R$.
(2) complete *if for all directed* $X \subseteq A$, $\bigvee X \in A$. [2]
(3) uniform *if* $A \neq \emptyset$ *and* $\forall n \in \omega\, (d\, A\, e \Rightarrow d_n\, A\, e_n)$.

The uniformity condition will play an important role in proving that the associated quotient space is algebraic and in solving domain equations.

Proposition 15.6.4 *The category of complete uniform per's is cartesian closed.*

PROOF HINT. We define the terminal object as $1 = D \times D$. For the product let

$$d\,(A_1 \times A_2)\,e \ \text{ iff } \ \pi_i(j_\times(d))\, A_i\, \pi_i(j_\times(e)) \ \text{ for } i = 1, 2 \ .$$

The exponent is defined as usual:

$$f\, B^A\, g \ \text{ iff } \ \forall d, e\, (d\, A\, e \Rightarrow fd\, B\, ge) \ .$$

Let us check that B^A is uniform if A, B are. From $\bot\, B\, \bot$, $\bot\, B^A\, \bot$ follows. Suppose $f\, B^A\, g$ and $d\, A\, e$. To show $f_n d\, B\, g_n e$ observe:

$$d\, A\, e \Rightarrow d_n\, A\, e_n \Rightarrow fd_n\, B\, ge_n \Rightarrow (fd_n)_n\, B\, (ge_n)_n$$

and we know $(fd_n)_n = f_{n+1}d$. □

Exercise 15.6.5 *Following section 15.2 define an interpretation of system F in cuper's.*

Complete per's (cper's for short) are closed under intersections. Then we can complete a per to a cper as follows.

[2] In this section "complete" has a different meaning than in the previous section.

Definition 15.6.6 (completion) *Let A be a per over D_∞. The least complete per containing A is defined as:*

$$\underline{A} = \bigcap\{B \mid B \text{ cper and } B \supseteq A\} .$$

In the following we give an inductive characterization of $\underline{A}$.

Definition 15.6.7 *Let R be a binary relation on D. We define:*

$$\begin{array}{lll} Sup(R) & = \{\bigvee X \mid X \text{ directed in } R\} & \text{(directed closure)} \\ TC(R) & = \bigcap\{S \mid S \text{ transitive and } S \supseteq R\} & \text{(transitive closure)} . \end{array}$$

Proposition 15.6.8 *(1) If R is symmetric (pointed) then $Sup(R)$ and $TC(R)$ are symmetric (pointed).*
(2) If A is a pointed per then $TC(Sup(A))$ is a pointed per.

PROOF. Immediate. □

Definition 15.6.9 *Let A be a pointed per. Define*

$$\begin{array}{ll} A(0) & = A \\ A(\alpha+1) & = TC(Sup(A(\alpha))) \\ A(\mu) & = \bigcup_{\alpha<\mu} A(\alpha) \quad (\mu \text{ limit ordinal}) . \end{array}$$

Let A be a pointed per. Then for cardinality reasons there is some β such that $A(\beta) = \underline{A}$. The following lemma points out the effect of the completion process on the function space and on uniformity.

Lemma 15.6.10 *(1) If A and B are pointed per's then $B^A \subseteq \underline{B}^{\underline{A}}$.*
(2) If A is a uniform per then $\underline{A}$ is a cuper.

PROOF. (1) By induction on α we show that $B^A \subseteq B(\alpha)^{A(\alpha)}$. The base and limit case are clear. Suppose $f\, B^A\, g$. We distinguish two cases.

- If $d = \bigvee_{i\in I} d^i$ and $e = \bigvee_{i\in I} e^i$, where $\{(d^i, e^i)\}_{i\in I}$ is directed in $A(\alpha)$ then $\{(fd^i, ge^i)\}_{i\in I}$ is directed in $B(\alpha)$ and therefore:

$$(fd, ge) = (\bigvee_{i\in I} fd^i, \bigvee_{i\in I} ge^i) \in Sup(B(\alpha)) .$$

- If $d\, TC(Sup(A(\alpha)))\, e$ then we can apply the previous case to each edge of the path connecting d to e.

(2) By induction on α we show that $A(\alpha)$ is uniform. The base and limit cases are clear. Suppose $d\, A(\alpha+1)\, e$. Again we distinguish two cases:

- If $d = \bigvee_{i\in I} d^i$ and $e = \bigvee_{i\in I} e^i$, where $\{(d^i, e^i)\}_{i\in I}$ is directed in $A(\alpha)$, we show $d_n\, Sup(A(\alpha))\, e_n$ by observing that $(\bigvee_{i\in I} d^i)_n = \bigvee_{i\in I}(d^i)_n$ and $\{((d^i)_n, (e^i)_n)\}_{i\in I}$ is directed in $A(\alpha)$. Hence $Sup(A(\alpha))$ is uniform.
- Suppose $d^1\, TC(Sup(A(\alpha)))\, d^k$ because $d^1\, Sup(A(\alpha))\, d^2 \cdots d^{k-1}\, Sup(A(\alpha))\, d^k$. Then $(d^1)_n\, Sup(A(\alpha))\, (d^2)_n \cdots (d^{k-1})_n\, Sup(A(\alpha))\, (d^k)_n$, as $Sup(A(\alpha))$ is uniform by the previous case. Therefore $(d^1)_n\, TC(Sup(A(\alpha)))\, (d^k)_n$. □

Exercise 15.6.11 *Show that the category of complete per's is reflective in the category of pointed per's, and that the category of complete uniform per's is reflective in the category of uniform per's.*

The intrinsic preorder $\leq_A$ on a cuper A induces a preorder on $|A|$ as follows

Definition 15.6.12 (induced preorder) *Let A be a cuper and $d, e \in |A|$. We define:*

$$d \leq_A e \quad \textit{iff} \quad [d]_A \leq_A [e]_A \ .$$

In the following we characterize the induced preorder.

Definition 15.6.13 *Let A be a cuper. Define $\preceq_A = TC(A \cup (\leq_D \cap |A|^2))$.*

Lemma 15.6.14 *Let A be a cuper. Then $\preceq_A$ is a uniform preorder on $|A|$.*

PROOF. We observe that $A \cup (\leq_D \cap |A|^2)$ is uniform and that transitive closure preserves uniformity. □

Lemma 15.6.15 *Let $d \in \mathcal{K}(D)$ be a compact element and let A be a cuper. Then the following set is a Scott open:*

$$W(d) = \{e \in D \mid \exists e' (d \preceq_A e' \leq_D e)\} \ .$$

PROOF. Clearly $W(d)$ is upward closed. Suppose $e = \bigvee_{i \in I} e^i \in W(d)$ for $\{e^i\}_{i \in I}$ directed. From $d \preceq_A e' \leq_D \bigvee_{i \in I} e^i$ we derive:

$$\exists n, j \, (d = d_n \preceq_A e'_n \leq_D (\bigvee_{i \in I} e^i)_n = e^j_n \leq e^j) \ .$$

This follows from the uniformity of $\preceq_A$ and the fact that $im(p_n)$ is finite. We can conclude $e_j \in W(d)$. □

Remark 15.6.16 *Let $d \in \mathcal{K}(D)$ be a compact element and A be a cuper. Then $U(d) = \{[e]_A \mid d \preceq_A e\} \in \Sigma(A)$. It is enough to observe $W(d) \cap |A| = |U(d)|$.*

Lemma 15.6.17 *Let $d \in \mathcal{K}(D)$ be a compact element and A be a cuper. Then:*

$$d \preceq_A e \quad \textit{iff} \quad d \leq_A e \ .$$

PROOF. By remark 15.4.7 it follows $d \preceq_A e$ implies $d \leq_A e$. Vice versa, suppose $d \leq_A e$ and *not* $d \preceq_A e$. Build the Scott open $W(d)$ as in lemma 15.6.15 and the sub-per $U(d)$ as in remark 15.6.16. Then $[d]_A \in U(d)$ and $[e]_A \notin U(d)$ which contradicts $d \leq_A e$. □

Theorem 15.6.18 *Let A be a cuper. Then:*
(1) The induced preorder is the least complete preorder containing $\preceq_A$.
(2) The preorder $\leq_A$ is uniform.

PROOF. We denote with $\preceq_A^c$ the least complete preorder containing $\preceq_A$.

(1) We already know that $\preceq_A \subseteq \leq_A$. Hence $\preceq_A^c \subseteq \leq_A$ since $\leq_A$ is complete. Vice versa, suppose $d \leq_A e$. Then $\forall n\,(d_n \leq_D d \leq_A e)$. So $\forall n\,(d_n \leq_A e)$ and by lemma 15.6.17, $\forall n\,(d_n \preceq_A e)$. By completeness $d = \bigvee_{n<\omega} d_n \preceq_A^c e$.

(2) We know from lemma 15.6.14 that $\preceq_A$ is uniform and we have already observed in lemma 15.6.10 that the completion process preserves uniformity. □

Theorem 15.6.19 *Let A be a separated cuper. Then $([A], \leq_A)$ is a bifinite domain.*

PROOF. Clearly $([A], \leq_A)$ is a poset with least element $[\bot]_A$.

• We show that any (infinite) directed set $\{[d^i]_A\}_{i\in I}$ has a lub. Given $J' \subseteq J$ we say that J' is *cofinal* with J if:

$$\forall i \in J\, \exists j \in J'\, (d^i \leq_A d^j)\ .$$

Let $X_n = \{e \in D \mid \forall i \in I\, \exists j \in I\, (d^i \leq_A d^j \text{ and } e = d^j_n)\}$, in other words $e \in X_n$ if there is a subset J of I, cofinal with I, and such that $\forall j \in J\, (e = d^j_n)$. We remark:

• $X_n \subseteq im(p_n) \cap |A|$ is finite since $im(p_n)$ is finite. Moreover X_n is non-empty since at least one element in $im(p_n)$ will be hit infinitely often when projecting elements in the directed set.

• $\forall e \in X_n\, \exists e' \in X_{n+1}\, (e \leq_D e')$. We show this by induction on n. If $n = 0$ then $e = \bot$ and every e' will do. If $e \in X_n$ then there is a J, cofinal with I such that $J \subseteq I$ and $\forall j \in J\, (d^j_n = e)$. Since $im(p_n)$ is finite there is $J' \subseteq J$ cofinal with J (hence with I) and an element e' such that $\forall j \in J'\, (d^j_{n+1} = e')$. Then $e \leq e'$ since $e = d^j_n \leq_D d^j_{n+1} = e'$, and $e' \in X_{n+1}$, by construction.

Hence we can build a sequence $\{e^n\}_{n\in\omega}$ such that $e^n \in X_n$ and $e^n \leq_D e^{n+1}$. By completeness we have $\bigvee_{n\in\omega} e^n \in |A|$. We claim:

$$\bigvee_{i\in I} [d^i]_A = [\bigvee_{n\in\omega} e^n]_A\ .$$

In the first place we show that $\forall i \in I\, (d^i \leq_A \bigvee_{n\in\omega} e^n)$. By completeness and uniformity it is enough to prove:

$$\forall i \in I\, \forall m \in \omega\, (d^i_m \leq_A \bigvee_{n\in\omega} e^n)\ .$$

We observe:

$$\forall i \in I \, \forall m \in \omega \, \exists j \in I \, d^i \leq_A d^j \text{ and } d^j_m = e^m \ .$$

By uniformity, we have $d^i_m \leq_A d^j_m$, and $d^j_m = e^m \leq_D \bigvee e^n$. Finally we note:

$$\forall n \in \omega \, \exists i \in I \, e^n \leq_A d^i$$

as $\exists i \, (e^n = d^i_n \leq_D d^i)$. Given $[d]_A$ upper bound for $\{[d^i]_A\}_{i \in i}$ it is immediate to show $\bigvee_{n \in \omega} e^n \leq_A d$.

• Next let us prove that the quotient space is ω-algebraic. We claim:
(1) If $d \in \mathcal{K}(D) \cap |A|$ then $[d]_A$ is compact in $([A], \leq_A)$.
Suppose $[d]_A \leq_A \bigvee_{i \in I} x^i$, for $\{x^i\}_{i \in I}$ directed. Consider the chain $\{e^n\}_{n \in \omega}$ we have built above. Then $d \leq_A \bigvee_{n \in \omega} e^n$ Hence:

$$\exists m, p, j \, (d = d_m \leq_A (\bigvee_{n \in \omega} e^n)_m = e^p_m \leq_A e^p \leq_A d^j) \ .$$

(2) $\forall d \in |A| \, ([d]_A = \bigvee_{n \in \omega} [d_n]_A)$.
We observe $d_n \leq_D d_{n+1}$ implies $d_n \leq_A d_{n+1}$ and moreover, if $\forall n \in \omega \, d_n \leq_A e$ then by completeness $\bigvee_{n \in \omega} d_n \leq_A e$.

• To prove that $([A], \leq_A)$ is bifinite we consider the sequence $\{prj^n : A \to A\}_{n \in \omega}$ where prj^n is the function realized by the projection p_n. □

Corollary 15.6.20 *All morphisms in the full subcategory of separated, complete, uniform per's are Scott continuous.*

PROOF. Consider $f : A \to B$ and $\{[d^i]_A\}_{i \in I}$ directed in $[A]$. The existence of $\bigvee_{i \in I} f([d^i]_A)$ is guaranteed by the monotonicity of f and theorem 15.6.19. It remains to prove:

$$f(\bigvee_{i \in I} [d^i]_A) \leq_B \bigvee_{i \in I} f([d^i]_A) \ .$$

Suppose $\phi \Vdash f$ and consider the chain $\{e^n\}_{n \in \omega}$ built in theorem 15.6.19. Then we have $\phi(\bigvee_{n \in \omega} e^n) \cong \bigvee_{n \in \omega} \phi e^n$. Also, since $\forall n \, \exists i \in I \, (e^n \leq_A d^i)$, we have, by monotonicity $\forall n \in \omega \, \exists i \in I \, (\phi e^n \leq_B \phi d^i)$. Hence we can conclude $[\bigvee_{n \in \omega} \phi e^n]_B \leq_B \bigvee_{i \in I} f([d^i]_A)$. □

Domain equations can be solved in the category of cuper's, by an adaptation of the traditional approach based on injection-projection pairs [AP90]. In the following we follow a more direct path that exposes an interesting metric structure on the space of cuper's [Ama91c].

Definition 15.6.21 *Define a* closeness *function* $c : cuper^2 \to \omega \cup \{\infty\}$ *as follows:*

$$c(A,B) = \begin{cases} max\{n \mid A_n = B_n\} & \text{if } A \neq B \\ \infty & \text{otherwise} \end{cases} \ .$$

The distance $d : cuper^2 \to R$ *is defined as*

$$d(A,B) = \begin{cases} 2^{-c(A,B)} & \text{if } A \neq B \\ 0 & \text{otherwise} \end{cases} .$$

The space *cuper* resembles spaces of infinite labelled trees [AN80]. Roughly speaking these spaces are compact if the collection of distinct objects up to the n-th level is finite.

Proposition 15.6.22 *(1)* $(cuper, d)$ *is a metric space.*
(2) The space is an ultra-metric*, that is* $d(A,C) \leq max\{d(A,B), d(B,C)\}$.
(3) The space is (Cauchy) complete.

PROOF. The first point is left to the reader. For the second point observe that:

$$A_n = B_n \text{ and } B_m = C_m \Rightarrow A_k = C_k \quad \text{where } k = min\{n,m\} .$$

For the third point, let $\{A^i\}_{i<\omega}$ be a Cauchy sequence, that is:

$$\forall \epsilon > 0 \, \exists n_\epsilon \, \forall i,j \geq n_\epsilon \, (d(A^i, A^j) < \epsilon) .$$

We build $A = lim_{i<\omega} A^i$ by stages. We note that:

$$\forall n > 0 \, \exists k_n \, \forall i \geq k_n \, A^i_n \text{ is constant} .$$

Let $B^i = A^{k_i}_i$. We observe that $\{B^i\}_{i<\omega}$ is a chain of cuper's with respect to inclusion. Let $B = \bigcup_{i<\omega} B^i$. We claim that $\underline{B} = lim_{i<\omega} A^i$. To this end it is enough to check:

$$\forall i \, \forall \alpha \, B^i = (B(\alpha))_i .$$

In other terms the completion operation does not add new approximating elements. This can be shown by induction on α (cf. proof lemma 15.6.10). □

An operator f over a metric space (X,d) is *contractive* if there is a constant c such that $0 \leq c < 1$ and

$$\forall x,y \, d(f(x), f(y)) \leq c \, d(x,y) .$$

A well-known result known as Banach's theorem states that contractive operators over a complete metric space have a unique fixed point (exercise!). It turns out that exponent and product type constructors are contractive. It follows that the related recursive type equations have a unique solution in cuper up to equality. This fact is applied in exercise 15.7.5.

Proposition 15.6.23 *Let* $d((A,B),(A',B')) = max\{d(A,A'), d(B,B')\}$. *Then:*
(1) $d(B^A, B'^{A'}) \leq (1/2) d((A,B),(A',B'))$.
(2) $d(A \times B, A' \times B') \leq (1/2) d((A,B),(A',B'))$.

PROOF HINT. We note that: $A_k = A'_k$ and $B_k = B'_k \Rightarrow (B^A)_{k+1} = (B'^{A'})_{k+1}$. The factor $(1/2)$ comes form definition 15.6.21 and the properties of D_∞ models. □

15.7 Interpretation of Subtyping

We present an application of the category **cuper** to the development and interpretation of a theory for the subtyping of recursive types. Let us start with an intuitive explanation of what subtyping is. Various theories of *subtyping* have been proposed in the literature on software engineering (see, e.g., [Car88, Lis88]). Their principal aim is to support a certain cycle of software development where programs evolve over time as they are restructured and new functionalities are added. Such theories support an incremental design of software systems and establish under which conditions the programmer is allowed to *reuse* previously created modules.

Such reuse may require the introduction of explicit or implicit *coercions* whose effect on the semantics of the program has to be clearly understood by the programmer. A formalization of this concept in the context of *typed languages* can be given in two steps:

- Introduce a relation of subtype denoted by $\leq$. If σ and τ are types, the intuitive interpretation of $\sigma \leq \tau$ (read as σ is a subtype of τ) is: every σ-value can be coerced to a τ-value.
- Specifiy nature and use of such coercions.

In other terms the two basic questions in the design of a typed λ-calculus with subtypes are whether two types are in the subtype relation, and whether a term has a type.

In the approach to be formalized next we take the view that σ is a subtype of τ if for every term M of type σ, say $M : \sigma$, and for every possible choice of a run time code d for M (henceforth we will say that d is a realizer for M), there is a unique term $N : \tau$ (up to semantic equivalence) that has d among its realizers. This approach is inspired by model-theoretical considerations [BL88] as one can give a precise mathematical meaning to our informal statements in the framework of per-models. For the time being let us anticipate the pragmatic consequences of our view of subtyping and coercions:

- Coercions are uniquely determined.
- Coercions do not produce run-time code, hence there is no need for recompilation.
- The specific "implementation" of a data-type becomes relevant, as subtyping is not invariant under isomorphism. For instance the types $\sigma \times \sigma' \to \tau$ and $\sigma \to (\sigma' \to \tau)$ are isomorphic but they are incomparable with respect to the subtyping relation.

Definition 15.7.1 (interpretation of subtyping) *Let $\mathcal{T}$ be a type structure (cf. definition 15.3.1). We write $\mathcal{T} \models \sigma \leq \tau$ if for any η, $[\![\sigma]\!]\eta \subseteq [\![\tau]\!]\eta$.*

Remark 15.7.2 *(1) In the semantic framework developed for type assignment systems we have that $\mathcal{T} \models \sigma \leq \tau$ iff $\mathcal{T} \models \lambda x.x : \sigma \to \tau$. (2) Let A, B be per's, if*

$A \subseteq B$ then there is a unique morphism $c : A \to B$ in per that has the identity (formally the combinator skk) among its realizers. We refer to this morphism as the coercion morphism *from A to B. Incidentally the vice versa also holds: if $c : A \to B$ is a coercion morphism then $A \subseteq B$.*

In order to discuss the impact of this interpretation of subtyping on language design we consider a simply typed λ-calculus with recursive types, the $\lambda\mu_{\leq}$-calculus for short. The language of types is defined as follows:

$$\begin{array}{lll} tv & ::= & t \mid s \mid \ldots \\ \sigma & ::= & tv \mid \bot \mid \top \mid \sigma \to \tau \mid \mu tv.\sigma \ . \end{array}$$

Here $\bot$ and $\top$ are two constant types that denote the least and greatest type in the subtyping relation, respectively. The type $\mu t.\sigma$ is intended to denote the "least" solution of the equation $t = \sigma(t)$. The language of terms is defined as follows:

$$\begin{array}{lll} v & ::= & x \mid y \mid \ldots \\ M & ::= & v \mid \lambda v : \sigma.M \mid MM \mid \mathit{fold}_{\mu tv.\sigma} M \mid \mathit{unfold}_{\mu tv.\sigma} M \ . \end{array}$$

Besides the usual rules for the simply typed λ-calculus we have rules for *folding* and *unfolding* recursive types:

$$\frac{\Gamma \vdash M : \sigma[\mu t.\sigma/t]}{\Gamma \vdash \mathit{fold}_{\mu t.\sigma} M : \mu t.\sigma} \quad \frac{\Gamma \vdash M : \mu t.\sigma}{\Gamma \vdash \mathit{unfold}_{\mu t.\sigma} M : \sigma[\mu t.\sigma/t]} \ .$$

Following our informal discussion on subtyping we want to define a formal theory to derive when $\sigma \leq \tau$ and enrich the typing system with the following rule

$$(\mathit{Sub}) \quad \frac{\Gamma \vdash M : \sigma \quad \sigma \leq \tau}{\Gamma \vdash M : \tau} \ .$$

We introduce in figure 15.3 a formal theory for deriving subtyping judgments on recursive types. The theory is composed of two groups of rules:

(1) The first group defines the least congruence induced by the the rules (μ-$\bot$), (*fold*), and ($\mu_{\downarrow}$). In the ($\mu_{\downarrow}$) rule the condition $\sigma \downarrow t$ is read as t is *contractive* in σ and means that σ can be rewritten by unfolding into a type of the shape $\sigma_1 \to \sigma_2$. For instance $\mu s.(t \to s) \downarrow t$ but $\mu s.t \not\downarrow t$. The rules for type equivalence are inspired by classical results on regular languages (see, e.g., [Sal66]). The ($\mu_{\downarrow}$) rule should be regarded as a syntactic version of Banach's theorem (cf. section 15.6).

(2) The second group of rules is used to derive proper inequalities. The basic judgment has the shape $\Delta \vdash \sigma \leq \tau$, where $\Delta \equiv t_1 \leq s_1, \ldots, t_n \leq s_n$, t_i, s_i are type variables, and $n \geq 0$. The rule ($\to$) resembles the one introduced for filter models in chapter 3. The intuition for the premise of the rule (μ) is that the following holds: for all per's A, B, if $A \subseteq B$ then $[\![\sigma]\!][A/t] \subseteq [\![\tau]\!][B/s]$.

Rules for equality

$$(\mathit{refl})\ \frac{}{\sigma=\sigma} \qquad (\mathit{sym})\ \frac{\sigma=\tau}{\tau=\sigma}$$

$$(\mathit{tr})\ \frac{\sigma=\tau\ \ \tau=\rho}{\sigma=\rho} \qquad (\to_=)\ \frac{\sigma=\sigma\ \ \tau=\tau'}{\sigma\to\tau=\sigma'\to\tau'}$$

$$(\mu_=)\ \frac{\sigma=\tau}{\mu t.\sigma=\mu t.\tau} \qquad (\mu\text{-}\bot)\ \frac{}{\mu t.t=\bot}$$

$$(\mathit{fold})\ \frac{}{\mu t.\sigma=\sigma[\mu t.\sigma/t]} \qquad (\mu_\downarrow)\ \frac{\sigma[\tau/t]=\tau\ \ \sigma[\tau'/t]=\tau'\ \ \sigma\downarrow t}{\tau=\tau'}\ .$$

Rules for subtyping

$$(\mathit{eq})\ \frac{\sigma=\tau}{\Delta\vdash\sigma\le\tau} \qquad (\mathit{tr})\ \frac{\Delta\vdash\sigma\le\tau\ \ \Delta\vdash\tau\le\rho}{\Delta\vdash\sigma\le\rho}$$

$$(\mathit{Asmp})\ \frac{t\le s\in\Delta}{\Delta\vdash t\le s}$$

$$(\bot)\ \frac{}{\Delta\vdash\bot\le\sigma} \qquad (\top)\ \frac{}{\Delta\vdash\sigma\le\top}$$

$$(\to)\ \frac{\Delta\vdash\sigma'\le\sigma\ \ \Delta\vdash\tau\le\tau'}{\Delta\vdash\sigma\to\tau\le\sigma'\to\tau'} \qquad (\mu)\ \frac{\Delta,t\le s\vdash\sigma\le\tau\ \ t\notin FV(\tau),\ s\notin FV(\sigma)}{\Delta\vdash\mu t.\sigma\le\mu s.\tau}$$

Figure 15.3: Subtyping recursive types

Exercise 15.7.3 *Derive the following judgments:*

$$\begin{array}{ll}\mu t.(s\to t) & =\mu t.(s\to(s\to t))\\ \mu t.(t\to(t\to t)) & =\mu t.((t\to t)\to t)\\ \mu s.(\top\to s) & \le\bot\to(\mu s.(s\to s))\ .\end{array}$$

Next we interpret the $\lambda\mu_{\le}$-calculus in cuper's.

Definition 15.7.4 *The type interpretation is parametric in $\eta : Tvar \to cuper$ and is defined as follows:*

$$\begin{array}{ll}[\![\bot]\!]\eta & =\{(\bot_D,\bot_D)\}\\ [\![\top]\!]\eta & =D\times D\\ [\![\sigma\to\tau]\!]\eta & =[\![\tau]\!]\eta^{[\![\sigma]\!]\eta}\\ [\![\mu t.\sigma]\!]\eta & =Fix(\lambda A.[\![\sigma]\!]\eta[A/t])\end{array}$$

$$Fix(f) = \begin{cases} x & \text{if } f \text{ is contractive and } f(x) = x \\ \{(\bot_D, \bot_D)\} & \text{if } f = id \\ \text{undefined} & \text{otherwise} \end{cases} .$$

Exercise 15.7.5 *Verify that the type interpretation is always defined (cf. proposition 15.6.23).*

Definition 15.7.6 *We write $t_1 \leq s_1, \ldots, t_n \leq s_n \models \sigma \leq \tau$ if for all type environments η, if $\eta(t_i) \subseteq \eta(s_i)$ for $i = 1, \ldots, n$ then $[\![\sigma]\!]\eta \subseteq [\![\tau]\!]\eta$.*

Theorem 15.7.7 *If $\Delta \vdash \sigma \leq \tau$ then $\Delta \models \sigma \leq \tau$.*

PROOF HINT. By induction on the length of the derivation. We have already observed that rule $(\mu_\downarrow)$ is a syntactic version of Banach's theorem. The only rule that deserves an additional comment is (μ). If f is contractive or the identity and C is the least cuper then the Cauchy sequence $\{f^n(C)\}_{n<\omega}$ converges to $Fix(f)$. Suppose f, g are contractive or the identity, the semantic reading of the rule goes as follows:

$$\frac{\forall A, B (A \subseteq B \Rightarrow f(A) \subseteq g(B))}{Fix(f) \subseteq Fix(g)} .$$

From the premises we can prove by induction $f^n(C) \subseteq g^n(C)$. From this we can draw the conclusion $Fix(f) \subseteq Fix(g)$. □

Exercise 15.7.8 *Prove that the following inequality holds in the cuper's interpretation but is not derivable in the system (with empty context) $\alpha \to \alpha' \leq \beta \to \top$. It is shown in [AC93] that the system extended with the inequality above is complete with respect to a modified interpretation.*

The *term interpretation* follows the interpretation of system F in the category of per's defined in section 15.2. The constants *fold* and *unfold* are interpreted by the identity, as recursive equations are solved up to equality. More results on this theory of subtyping can be found in [AC93]. Two important points that hint to the practical relevance of the theory sketched above are:

(1) It is decidable if $\phi \vdash \sigma \leq \tau$.

(2) There is an algorithm that decides if a term is typable, and if this is the case the algorithm returns the least type that can be assigned to the term.

Chapter 16

Functions and Processes

The functional view of computation finds perhaps its most serious limitation in the analysis of concurrent systems (cf. chapter 9). The challenge is then to cope with the problems offered by concurrent systems while retaining some of the mathematically brilliant ideas and techniques developed in the pure functional setting.

In this chapter we introduce a simple extension of CCS known as π-calculus. The π-calculus is a rather minimal calculus whose initial purpose was to represent the notion of name or reference in a concurrent computing setting. It turns out that the π-calculus allows for simple encodings of various functional and concurrent models. It can then be used as a privileged tool to understand in which sense functional computation can be embedded in a concurrent model.

Section 16.1 is dedicated to the introduction of some basic theory of the π-calculus. In section 16.2 we illustrate the expressive power of the π-calculus by encoding into it a concurrent functional language, the $\lambda_{||}$-calculus for short, that can be regarded as the kernel of *concurrent* extensions of the ML programming language such as LCS, CML and FACILE where an integration of functional and concurrent programming is attempted.

16.1 π-calculus

In chapter 9 we have presented a calculus of processes, CCS, in which interaction arises as *rendez-vous* synchronization on communication channels. This computation paradigm is enhanced in the π-calculus (see [MPW92], after [AZ84, EN86]) by allowing:

- Channel names as transmissible values.
- The generation of new channels.

Because of these essential features the development of the π-calculus theory along the lines known for CCS (labelled transition system and related bisimulation) leads to a series of complications which can be hard to appreciate for a beginner.

For this reason we follow a different approach. We present first the π-calculus as a *programming language*. Technically this means to specify abstractly how a π-calculus program can be evaluated and to explain how this evaluation can be implemented. Once a reasonably clear implementation model has been sketched we introduce a notion of observation as the capability of a process to *commit* to a certain communication and we derive a notion of *barbed equivalence* on processes.

Barbed equivalence is a natural relation by which two π-terms can be compared [MS92]. Unfortunately it is difficult to relate two processes using this approach, as we always have to work with arbitrary contexts. This motivates the quest for a characterization of barbed equivalence which is better suited to mechanical verification. Towards this end, we introduce a labelled transition system and a related notion of *π-bisimulation*. A central result, whose proof we present here, says that π-bisimulation and barbed equivalence coincide. As an application of this characterization we show the decidability of equivalence for a special class of *finite control* processes.

The Language. We suppose that there is a countable collection of *channel names* that we denote with $a, b, \ldots$ *Processes* are specified by the following grammar:

$$\begin{array}{rl} n & ::= a \mid b \mid \ldots \\ P & ::= 0 \mid \overline{n}n.P \mid n(n).P \mid \nu n\, P \mid (P \mid P) \mid [n = n]P \mid (\gamma.P + \cdots + \gamma.P) \mid A(\vec{n})\ . \end{array}$$

- 0 is the process which is terminated and that can be garbage collected. Usually we omit writing 0, e.g. $\overline{a}b$ stands for $\overline{a}b.0$
- $\overline{a}b.P$ is the process that sends the channel name b on the channel a and becomes P.
- $a(b).P$ is the process that receives a channel name, say c, on the channel a and becomes $P[b/c]$. The formal parameter b is bound in $a(b).P$, in general bound names can be renamed.
- $\nu a\, P$ is the process that creates a new name different from all the existing ones and becomes P. The name a is bound in $\nu a\, P$. We denote with $FV(P)$ the collection of names occurring free in P.
- $(P \mid P)$ is the parallel composition of two processes.
- $[a = b]P$ is the matching construct. If the match holds then execute P else terminate.
- $\gamma_1.P + \cdots + \gamma_n.P$ is a *guarded sum*, where all alternative processes commit on an input/output action. The prefix γ is an abbreviation for an input/output guard, i.e. $\gamma ::= \overline{n}n \mid n(n)$.
- We denote with $A, B, \ldots$ agent identifiers. For every agent identifier there is a unique defining equation $A(a_1, \ldots, a_n) = P$ where all free names in P are included in $\{a_1, \ldots, a_n\}$ and all occurrences of an agent identifier in P are preceded by

$$\frac{}{(\overline{a}b.P + P') \mid (a(c).Q + Q') \to P \mid Q[b/c]} \qquad \frac{}{[a = a]P \to P}$$

$$\frac{P \to Q}{D[P] \to D[Q]} \text{ where } D ::= [\,] \mid D \mid P \mid \nu n\, D \qquad \frac{P \equiv P' \quad P' \to Q' \quad Q' \equiv Q}{P \to Q}$$

Figure 16.1: Reduction for the π-calculus

an input/output prefix. When writing processes guarded sum has priority over parallel composition.

Structural Equivalence. The basic computation rule in π-calculus is:

$$\overline{a}b.P \mid a(c).Q \to P \mid Q[b/c] \tag{16.1}$$

Unlabelled reductions like those in rule 16.1 represent internal communications and correspond to the τ-transitions in CCS. The reduction rule 16.1 is not sufficient to represent all possible internal communications. In order to have a greater flexibility we define a relation ≡, called *structural equivalence*, which is the smallest congruence on processes generated by the following equations:

- Renaming: $c(a).P \equiv c(b).P[b/a]$, $\nu a\, Q \equiv \nu b\, Q[b/a]$, for $b \notin FV(c(a).P)$ and $b \notin FV(\nu a\, Q)$. We denote with $\equiv_\alpha$ the congruence that identifies terms differing only by the name of their bound variables.
- Parallel composition is an associative and commutative operator with 0 as identity.
- The order of the guards in the sum is irrelevant. By convention whenever we write $\gamma.P + Q$ we intend that Q denotes the rest of the guard, if there is any.
- Restriction commutations: $\nu a\, P \mid Q \equiv \nu a\, (P \mid Q)$, for $a \notin FV(Q)$.
- Equation unfolding: any agent identifier can be replaced by its definition.

Remark 16.1.1 (standard form) *Every term without matching is structurally equivalent to a term:*

$$\nu a_1 \ldots \nu a_k\, (Q_1 \mid \cdots \mid Q_m)$$

where Q_i is a guard, namely $Q_i \equiv \gamma_{i,1}.P_{i,1} + \cdots + \gamma_{i,n_i}.P_{i,n_i}$, for $i = 1, \ldots, m$ and $k, m, n_i \geq 0$ (conventionally take the parallel composition equal to 0 if $m = 0$).

Reduction. The reduction relation is presented in figure 16.1. In the first place, the reduction rule 16.1 is generalized in order to take into account guarded sums. Second, it is assumed that rewriting is modulo structural equivalence, and third, reduction can be performed in certain contexts D (note that it is not possible to reduce under an input/output guard, a bit like in the weak λ-calculus where it is not possible to reduce under λ-abstraction, see section 8.3). There is also a rule taking care of matching. In order to understand the role of the various rules we invite the reader to consider the following examples.

- Channel Transmission: a process sends on the channel b a channel name a which allows interaction with a process receiving on a.

$$\nu a\,(\nu b\,(\overline{b}a \mid b(c).\overline{c}e) \mid a(d).R') \rightarrow^{+} \nu a\,\nu b\,R'[e/d] \tag{16.2}$$

- Scope Intrusion: when receiving a channel under the scope of a restriction one has to avoid name clashes (on a in the example).

$$\overline{b}a \mid \nu a\,(b(c).Q \mid S) \rightarrow \nu a'\,(Q[a'/a][a/c] \mid S[a'/a]) \quad a' \text{ fresh} \tag{16.3}$$

- Scope Extrusion: when transmitting a restricted name, the scope of restriction has to be enlarged to the receiving process, this phenomenon is called scope extrusion (in the example a is the extruded name).

$$\nu a\,(\overline{b}a.P \mid R) \mid b(c).Q \rightarrow \nu a\,(P \mid R \mid Q[a/c]) \quad a \notin FV(\nu c\,Q) \tag{16.4}$$

Implementation. In this section we define an abstract machine which addresses two implementation problems: substitution and new name generation. These problems are specific of the π-calculus as opposed to CCS.

In order to implement substitution we can import the ideas already developed for environment machines (cf. chapter 8), hence reduction is defined on closures which are pairs of code and environment. Name generation requires a new idea. In the π-calculus reduction rules, name generation is treated implicitly via α-renaming and structural equivalence, in an implementation this is not admissible.

We describe an *abstract* machine as a term rewriting system modulo an associative and commutative operator representing parallel composition. Guarded sum, matching, and agent definitions are omitted in the following discussion. The machine can be extended to deal with these features without particular difficulties.

- *Channels* are represented as strings.
- *Process code* syntax differs from process syntax for the insertion of a commitment operator $\succ$. This operator is used to represent the fact that the evaluation of the prefix is terminated and the process is ready to commit on a communication.

$$C ::= 0 \mid \overline{n}n.C \mid n(n).C \mid \nu n\,C \mid (C \mid C) \mid \overline{n}n \succ C \mid n(n) \succ C\ .$$

$(0, \rho, \theta)$	$\rightarrow 1$
$(C \mid C', \rho, \theta)$	$\rightarrow (C, \rho, \theta 0) \| \| (C', \rho, \theta 1)$
$(\nu a\, C, \rho, \theta)$	$\rightarrow (C, \rho[ch(\theta)/a], \theta 0)$
$(\overline{a}b.C, \rho, \theta)$	$\rightarrow (\overline{a'}b' \succ C, \rho, \theta)$ if $\rho(a) = a'$ and $\rho(b) = b'$
$(a(b).C, \rho, \theta)$	$\rightarrow (a'(b) \succ C, \rho, \theta)$ if $\rho(a) = a'$
$(\overline{a}b \succ C, \rho, \theta) \| \| (a(c) \succ C', \rho', \theta')$	$\rightarrow (C, \rho, \theta) \| \| (C', \rho'[b/c], \theta')$

Figure 16.2: An environment machine for the π-calculus

- An *environment* ρ is a total function mapping channel names to channel names. Initially the environment is the identity function. The substitution operation is not carried out but it is recorded in the environment. The *actual* value of a channel name is obtained by application of the environment function.
- A *channel generator* is a string in $\{0,1\}^*$, we denote with θ a generic string and with ϵ the empty string. We suppose that there is an injective function ch that associates to every string θ a channel name $ch(\theta)$.
- A *process descriptor* is a triple (C, ρ, θ).
- We suppose that there is an associative and commutative operator $\|$ on process descriptors having 1 as identity. This is the only structural equivalence on which we rely.
- The process P is *compiled* into (P, id, ϵ). Initially all names in P are distinct from a name $ch(\theta)$, for any θ.

With the conventions above, an environment machine to reduce π-terms is described in figure 16.2 as a finite collection of term rewriting rules.

Exercise 16.1.2 *Reduce* $(\nu a\, \overline{b}a.a(a).\overline{a}b.0 \mid a(c).\overline{c}d.d(c).0, id, \epsilon)$.

Exercise 16.1.3 ** (1) The machine in figure 16.2 solves at once the substitution and the name generation problem. Describe a simpler machine which handles the name generation problem only, leaving substitution as a meta-operation. (2) Formulate a theorem that relates reduction in the π-calculus to reductions in the abstract machine specified in (1).*

There are other implementation problems that relate to concurrent languages in general and that will not be studied here. For instance, we may note that the machine described in figure 16.2 reduces modulo associativity and commutativity. Algebraic manipulations are needed in order to bring in a contiguous position two process descriptors committed on dual communications. Moreover the selection

of the term to be reduced next is non-deterministic. In practice we need an efficient and distributed way to perform communications. This task may include:

- The definition of a scheduler to order the jobs execution on a processor.
- The introduction of data structures to know which process wants to communicate on which channel.
- The execution of non-trivial protocols that guarantee a coherent selection of communications, while avoiding deadlock (see, e.g., [BS83]).

Barbed Equivalence. We now turn to the issue of stating when two processes are equivalent. We postulate that what can be observed of a process is its capability of committing (engaging) on an input/output communication on a visible (i.e. non-restricted) channel. From this a notion of process equivalence is derived as follows.

Definition 16.1.4 (commitment) *A relation of immediate commitment $P \downarrow \beta$ where $\beta ::= n \mid \overline{n}$ is defined as follows:*

$$\begin{array}{ll} P \downarrow c & \text{if } P \equiv \nu\vec{c}(c(a).P + P' \mid Q) \quad c \notin \{\vec{c}\} \\ P \downarrow \overline{c} & \text{if } P \equiv \nu\vec{c}(\overline{c}d.P + P' \mid Q) \quad c \notin \{\vec{c}\} \ . \end{array}$$

Moreover, define $\rightarrow^*$ as the reflexive and transitive closure of the reduction relation $\rightarrow$. Then a weak commitment *relation $P \downarrow_* \beta$ is defined as:*

$$P \downarrow_* \beta \text{ if } \exists P' (P \rightarrow^* P' \quad \text{and} \quad P' \downarrow \beta) \ .$$

Definition 16.1.5 (barbed (bi-)simulation) *A binary relation S between processes is a* (strong) barbed simulation *if PSQ implies:*

$$\begin{array}{l} \forall P' (P \rightarrow P' \ \Rightarrow \ \exists Q' (Q \rightarrow Q' \text{ and } P'SQ')) \text{ and} \\ \forall \beta (P \downarrow \beta \ \Rightarrow \ Q \downarrow \beta) \ . \end{array}$$

S is a barbed bisimulation *if S and S^{-1} are barbed simulations. The largest barbed bisimulation is denoted with $\overset{\bullet}{\sim}$. By replacing everywhere $\rightarrow$ by $\rightarrow^*$ and $\downarrow$ by $\downarrow_*$ one obtains the notion of* weak *barbed bisimulation. The largest weak barbed bisimulation is denoted with $\overset{\bullet}{\approx}$.* [1]

The relation $\overset{\bullet}{\sim}$ (or $\overset{\bullet}{\approx}$) fails to be a congruence, in particular $P \overset{\bullet}{\sim} P'$ does not imply $P \mid Q \overset{\bullet}{\sim} P' \mid Q$ (already in CCS, $a.b \overset{\bullet}{\sim} a.c$ does *not* imply $a.b \mid \overline{a} \overset{\bullet}{\sim} a.c \mid \overline{a}$). This motivates the introduction of the following definition.

[1] The adjective barbed relates to a pictorial representation of the reductions and commitments of a process. In this representation the commitments are the barbs and the internal reductions are the wires connecting the barbs.

Definition 16.1.6 (barbed equivalence) *We define a relation $\sim$ of strong barbed equivalence and a relation $\approx$ of weak barbed equivalence as follows:*

$$P \sim P' \quad \text{if } \forall Q\,(P \mid Q \stackrel{\bullet}{\sim} P' \mid Q)$$
$$P \approx P' \quad \text{if } \forall Q\,(P \mid Q \stackrel{\bullet}{\approx} P' \mid Q)\ .$$

Exercise 16.1.7 ** Which operators of the π-calculus preserve $\sim$ and $\approx$? Hint: theorem 16.1.20 can be helpful as it provides a characterization of strong barbed equivalence.*

Polyadic π-calculus. We introduce some additional concepts and notations for the π-calculus. So far we have assumed that each channel may transmit exactly one channel name. In practice it is more handy to have a calculus where tuples of channel names can be transmitted at once. This raises the problem of enforcing some sort discipline on channels, as emitting and receiving processes have to transmit and accept, respectively, a tuple of the same length. A simple sort discipline can be defined as follows. Every channel is supposed to be labelled by its *sort*. Sorts are used to constraint the arity of a channel. A channel of sort $Ch(s_1,\ldots,s_n)$ can carry a tuple $z_1,\ldots,z_n$, where z_i has sort s_i, for $i = 1,\ldots,n$. For instance, if $n = 0$ then the channel can be used only for synchronization (as in CCS), and if the sort is $Ch(o)$ then the channel can only transmit some ground data of type o.

$$\text{Simple sorts} \quad s ::= o \mid Ch(s_1,\ldots,s_n) \quad (n \geq 0)$$

The syntax for processes is extended in the obvious way:

$$P ::= \overline{n}(n,\ldots,n).P \mid n(n,\ldots,n).P \mid \cdots$$

Well-formed processes have to respect the sort associated to the channel names. For instance, $\overline{a}(b_1,\ldots b_n).P$ is well formed if P is well formed, a has sort $Ch(s_1,\ldots,s_n)$ and b_i has sort s_i, for $i = 1,\ldots,n$. *Mutatis mutandis*, reduction is defined as in figure 16.1. We call the resulting π-calculus *polyadic*.

Exercise 16.1.8 *(1) Define a translation from the polyadic to the monadic π-calculus. Hint translation: $\langle\overline{c}(a_1,\ldots,a_n).P\rangle = \nu b\,\overline{c}b.\overline{b}a_1\ldots\overline{b}a_n.\langle P\rangle$. (2) Check that τ-reduction is adequately simulated.*

Labelled Transition System. The aim is to define a labelled transition system (lts) (cf. section 9.2) for the π-calculus which describes not only the computations that a process can perform autonomously (the τ transitions) but also the computations that the process can perform with an appropriate cooperation from the environment.

Definition 16.1.9 (actions) *We postulate that a process can perform five kinds of actions α:*

$$\alpha ::= \tau \mid nn \mid \overline{n}n \mid n \mid \overline{n}\ .$$

We can provide the following intuition for the meaning of each action:

- The τ action corresponds to internal reduction as defined in figure 16.1.
- The cd and $\overline{c}d$ actions are complementary and they correspond, respectively, to the input and the output on channel c of a "global" channel name d.
- The c and $\overline{c}$ actions are also complementary and they correspond, respectively, to the input and the output on channel c of a "new" channel.

The notions of "global" and "new" are intended as relative to a given collection of channels which is visible to the environment. To represent this collection we introduce next the notion of context. It is possible to define the lts without referring to contexts as shown later in figure 16.4. At first, we prefer to stick to a more redundant notation which allows for an intuitive explanation of the rules.

Definition 16.1.10 (context) *A* context Γ *is a finite, possibly empty, set of channel names. We write* $c_1, \ldots, c_n$ $(n \geq 0)$ *for the set* $\{c_1, \ldots, c_n\}$*, and* Γ, c *for the set* $\Gamma \cup \{c\}$ *where* $c \notin \Gamma$*.*

To consider a process in a context we write $\Gamma \vdash P$, it is always intended that the context contains all channel names free in P. We are now ready to define an lts as an inference system for judgments of the shape $(\Gamma \vdash P) \stackrel{\alpha}{\to} (\Gamma' \vdash P')$ to be read as the process P in the context Γ can make an action α and become P' in the context Γ'. The actions τ, cd and $\overline{c}d$ leave the context unchanged whereas the actions c and $\overline{c}$ enrich the context with a new channel.

In figure 16.3 the only "structural rule" is α-renaming. In order to keep the system finitely branching we suppose that the collection of channel names Ch is linearly well-ordered and we let *fst* be a function that returns the least element in a non-empty set of channel names. In practice we pick the first name that does not occur in the current context (and hence is not free in the process at hand). The symmetric version of the rules (*sync*), ($sync_{ex}$), and (*comp*) are omitted.

To some extent all that matters in the computation of the transitions are the *distinctions* between channel names. In particular note that the choice of the new names is completely arbitrary. We invite the reader to carry on the following exercise which is useful in the proof of the following propositions.

Exercise 16.1.11 *(1) Let* σ *be an injective substitution on channel names. Relate transitions of* $\Gamma \vdash P$ *and* $\sigma\Gamma \vdash \sigma P$*. (2) Relate the transitions of* $\Gamma \vdash P$ *and* $\Gamma' \vdash P$ *for* $FV(P) \subseteq \Gamma \subseteq \Gamma'$*.*

Definition 16.1.12 (π-bisimulation) *A binary relation* S *on processes is a (strong)* π*-simulation if whenever* PSQ *and* $\Gamma = FV(P \mid Q)$ *the following holds:*

$$\forall P'(\Gamma \vdash P \stackrel{\alpha}{\to} \Gamma' \vdash P') \quad \Rightarrow \quad \exists Q'(\Gamma \vdash Q \stackrel{\alpha}{\to} \Gamma' \vdash Q' \text{ and } P'SQ') \ .$$

The relation S *is a* π*-bisimulation if* S *and* S^{-1} *are* π*-simulations. We denote with* $\sim_\pi$ *the greatest* π*-bisimulation.*

$$(out) \qquad \frac{}{\Gamma \vdash \overline{c}d.P \xrightarrow{\overline{c}d} \Gamma \vdash P}$$

$$(in) \qquad \frac{d \in \Gamma}{\Gamma \vdash c(a).P \xrightarrow{cd} \Gamma \vdash P[d/a]}$$

$$(out_{ex}) \qquad \frac{\Gamma, c \vdash P \xrightarrow{\overline{d}_c} \Gamma, c \vdash P' \quad c = fst(Ch \backslash \Gamma)}{\Gamma \vdash \nu c\, P \xrightarrow{\overline{d}} \Gamma, c \vdash P'}$$

$$(in_{ex}) \qquad \frac{a = fst(Ch \backslash \Gamma)}{\Gamma \vdash c(a).P \xrightarrow{c} \Gamma, a \vdash P}$$

$$(sync) \qquad \frac{\Gamma \vdash P \xrightarrow{\overline{d}_c} \Gamma \vdash P' \qquad \Gamma \vdash Q \xrightarrow{d_c} \Gamma \vdash Q'}{\Gamma \vdash P \mid Q \xrightarrow{\tau} \Gamma \vdash P' \mid Q'}$$

$$(sync_{ex}) \qquad \frac{\Gamma \vdash P \xrightarrow{\overline{d}} \Gamma, c \vdash P' \qquad \Gamma \vdash Q \xrightarrow{d} \Gamma, c \vdash Q'}{\Gamma \vdash P \mid Q \xrightarrow{\tau} \Gamma \vdash \nu c (P' \mid Q')}$$

$$(\nu) \qquad \frac{(\Gamma, c \vdash P) \xrightarrow{\alpha} (\Gamma, c, \Gamma' \vdash P') \quad c = fst(Ch \backslash \Gamma) \quad c \text{ not in } \alpha}{(\Gamma \vdash \nu c\, P) \xrightarrow{\alpha} (\Gamma, \Gamma' \vdash \nu c\, P')}$$

$$(comp) \qquad \frac{\Gamma \vdash P \xrightarrow{\alpha} \Gamma' \vdash P'}{\Gamma \vdash P \mid Q \xrightarrow{\alpha} \Gamma' \vdash P' \mid Q}$$

$$(match) \qquad \frac{}{\Gamma \vdash [c = c]P \xrightarrow{\tau} \Gamma \vdash P}$$

$$(sum) \qquad \frac{(\Gamma \vdash \gamma_i.P_i) \xrightarrow{\alpha} (\Gamma' \vdash P')}{(\Gamma \vdash \gamma_1.P_1 + \cdots + \gamma_n.P_n) \xrightarrow{\alpha} (\Gamma' \vdash P')}$$

$$(fix) \qquad \frac{\Gamma \vdash P[\vec{c}/\vec{a}] \xrightarrow{\alpha} \Gamma' \vdash P' \quad A(\vec{a}) = P}{\Gamma \vdash A(\vec{c}) \xrightarrow{\alpha} \Gamma' \vdash P'}$$

$$(rename) \qquad \frac{P \equiv_\alpha P' \quad \Gamma \vdash P' \xrightarrow{\alpha} \Gamma' \vdash Q' \quad Q' \equiv_\alpha Q}{\Gamma \vdash P \xrightarrow{\alpha} \Gamma' \vdash Q}$$

Figure 16.3: A labelled transition system for the π-calculus

Definition 16.1.13 *Let Pr be the collection of processes. We define a function $\mathcal{F} : \mathcal{P}(Pr \times Pr) \to \mathcal{P}(Pr \times Pr)$ by $P\,\mathcal{F}(S)\,Q$ if:*

$$\begin{array}{l} \forall \alpha, P', \Gamma, \Gamma' (\Gamma = FV(P \mid Q) \text{ and } \Gamma \vdash P \stackrel{\alpha}{\to} \Gamma' \vdash P') \\ \Rightarrow \quad \exists Q' (\Gamma \vdash Q \stackrel{\alpha}{\to} \Gamma' \vdash Q' \text{ and } P' S Q') \end{array}$$

and symmetrically.

Exercise 16.1.14 *Let $\sim^0 = Pr^2$, $\sim^{\kappa+1} = \mathcal{F}(\sim^\kappa)$, and $\sim^\lambda = \bigcap_{\kappa<\lambda} \sim^\kappa$, for λ limit ordinal. Prove that (cf. proposition 9.2.8): (1) $\mathcal{F}$ is monotonic. (2) S is a π-bisimulation iff $S \subseteq \mathcal{F}(S)$. (3) If $\{X_i\}_{i\in I}$ is a codirected set, then $\mathcal{F}(\bigcap_{i\in I} X_i) = \bigcap_{i\in I} \mathcal{F}(X_i)$. (4) The greatest π-bisimulation $\sim_\pi$ exists and coincides with $\sim^\omega$.*

Proposition 16.1.15 *Let σ be an injective substitution on names. Then for any processes P, Q, $P \sim_\pi Q$ iff $\sigma P \sim_\pi \sigma Q$.*

PROOF HINT. We show that the following relation is a π-bisimulation:
$\{(P,Q) \mid \exists \sigma \text{ injective on } FV(P \mid Q) \text{ such that } \sigma P \sim_\pi \sigma Q\}$. □

Exercise 16.1.16 *In the definition of π-bisimulation we consider transitions with respect to a context $\Gamma = FV(P \mid Q)$. This requirement can be relaxed. Consider a sharpened definition of the functional $\mathcal{F}$, say $\mathcal{F}_\sharp$, where the condition "$\Gamma = FV(P \mid Q)$" is replaced by the condition "$\Gamma \supseteq FV(P \mid Q)$". Let $\sim_\sharp$ be the greatest fixpoint of the functional $\mathcal{F}_\sharp$. Check that $\sim_\pi = \sim_\sharp$. Hint: $\sim_\pi \subseteq \mathcal{F}_\sharp(\sim_\pi)$.*

Exercise 16.1.17 *(1) Show that all structurally equivalent processes are π-bisimilar. (2) Which operators preserve π-bisimulation ? Hint: π-bisimulation is not a preserved by the input prefix, that is $P \sim_\pi Q$ does* not *imply $a(b).P \sim_\pi a(b).Q$. (3) Define the notion of weak π-bisimulation (cf. definition 9.2.14).*

We hint to a presentation of the labelled transition system which does not use contexts. We suppose that the actions are redefined as follows:

$$\alpha ::= \tau \mid nn \mid \overline{n}n \mid n(n) \mid \overline{n}(n) \ .$$

This differs from definition 16.1.9 because the *new* name b that is being received or emitted is explicitly indicated in $a(b), \overline{a}(b)$ (which replace, respectively, the actions $a, \overline{a}$). The name b is bound in these actions. More generally we define the following functions on actions, where *fn* stands for free names, *bn* stands for bound names, and n for names, where $n(\alpha) = bn(\alpha) \cup fn(\alpha)$ and:

$$\begin{array}{llll} fn(\tau) = \emptyset & fn(\overline{a}(b)) = fn(a(b)) = \{a\} & fn(\overline{a}b) = fn(ab) = \{a,b\} \\ bn(\tau) = \emptyset & bn(\overline{a}(b)) = bn(a(b)) = \{b\} & bn(\overline{a}b) = bn(ab) = \emptyset \ . \end{array}$$

The labelled transition system is defined in figure 16.4, where the symmetric version of the rules (*sync*), ($sync_{ex}$), and (*comp*) are omitted. Comparing with

$$(in)\quad \frac{}{a(b).P \xrightarrow{ac} [c/b]P} \qquad (out)\quad \frac{}{\overline{a}b.P \xrightarrow{\overline{a}b} P}$$

$$(in_{ex})\quad \frac{}{a(b).P \xrightarrow{a(b)} P} \qquad (out_{ex})\quad \frac{P \xrightarrow{\overline{a}b} P' \quad a \neq b}{\nu b\, P \xrightarrow{\overline{a}(b)} P'}$$

$$(sync)\quad \frac{P \xrightarrow{\overline{a}b} P' \quad Q \xrightarrow{ab} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad (sync_{ex})\quad \frac{P \xrightarrow{\overline{a}(b)} P' \quad Q \xrightarrow{a(b)} Q'}{P \mid Q \xrightarrow{\tau} \nu b\,(P' \mid Q')}$$

$$(\nu)\quad \frac{P \xrightarrow{\alpha} P' \quad a \notin n(\alpha)}{\nu a\, P \xrightarrow{\alpha} \nu a\, P'} \qquad (comp)\quad \frac{P \xrightarrow{\alpha} P' \quad bn(\alpha) \cap FV(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$$

$$(sum)\quad \frac{\gamma_i.P_i \xrightarrow{\alpha} P'}{\gamma_1.P_1 + \cdots + \gamma_n.P_n \xrightarrow{\alpha} P'} \qquad (match)\quad \frac{}{[a = a]P \xrightarrow{\tau} P}$$

$$(fix)\quad \frac{P[\vec{b}/\vec{a}] \xrightarrow{\alpha} P' \quad A(\vec{a}) = P}{A(\vec{b}) \xrightarrow{\alpha} P'} \qquad (rename)\quad \frac{P \equiv P' \quad P' \xrightarrow{\alpha} Q' \quad Q' \equiv Q}{P \xrightarrow{\alpha} Q}$$

Figure 16.4: A labelled transition system without contexts

the system in figure 16.3 we note that the rules are in bijective correspondence with those of the new system. Name clashes in the new system are avoided by inserting suitable side conditions on the rules (ν) and (*comp*). In a sense we trade contexts against side conditions. The definition of bisimulation can be adapted to the lts without contexts as follows.

Definition 16.1.18 *Let Pr be the collection of processes. We define an operator $\mathcal{F} : \mathcal{P}(Pr \times Pr) \to \mathcal{P}(Pr \times Pr)$ as:*

$$\begin{array}{ll} P\,\mathcal{F}(S)\,Q & \textit{if } \forall P' \forall \alpha\,(bn(\alpha) \cap FV(Q) = \emptyset \textit{ and } P \xrightarrow{\alpha} P') \\ & \Rightarrow\ \exists Q'\,(Q \xrightarrow{\alpha} Q' \textit{ and } P'\,S\,Q') \textit{ (and symmetrically)} \end{array}$$

where the transitions are computed in the lts defined in figure 16.4. A relation S is a bisimulation if $S \subseteq \mathcal{F}(S)$. We define $\sim_{\pi'} = \bigcup\{S \mid S \subseteq \mathcal{F}(S)\}$.

The condition $bn(\alpha) \cap FV(Q) = \emptyset$ is used to avoid name clashes (cf. rule (*comp*)). As expected, the two definitions of bisimulation turn out to be the same.

Exercise 16.1.19 ** For all processes P, Q, $P \sim_\pi Q$ iff $P \sim_{\pi'} Q$.*

Characterization of Barbed Equivalence. The definition of π-bisimulation is technically appealing because the check of the equivalence of two processes can be performed "locally" that is without referring to an arbitrary parallel context as in the definition of barbed equivalence. On the other hand the definition of π-bisimulation is quite intensional and clearly contains a certain number of arbitrary choices: the actions to be observed, the selection of new names,... The following result, first stated in [MS92], shows that strong π-bisimulation and barbed equivalence are two presentations of the same notion, and it justifies, a posteriori, the choice of the actions specified in definition 16.1.9 (this choice is not obvious, for instance the "late" π-bisimulation first studied in [MPW92], which is based on a different treatment of the input action, is strictly stronger than barbed equivalence).

Theorem 16.1.20 *Strong barbed equivalence and strong π-bisimulation coincide.*

PROOF. We first outline the proof for a CCS-like calculus following the notation in section 9.2. CCS can be seen as a π-calculus in which the transmitted names are irrelevant. Formally, we could code $a.P$ as $a(b).P$ and $\overline{a}.P$ as $\nu b\overline{a}b.P$, where $b \notin FV(P)$.

- $P \sim_\pi Q \ \Rightarrow\ P \sim Q$. We observe that:

(1) $P \sim_\pi Q \ \Rightarrow\ P \stackrel{\bullet}{\sim} Q$.

(2) $P \sim_\pi Q \ \Rightarrow\ P \mid R \sim_\pi Q \mid R$, for any R.

Hence, $P \sim_\pi Q$ implies $P \mid R \stackrel{\bullet}{\sim} Q \mid R$, for any R, that is $P \sim Q$.

• $P \sim Q \quad \Rightarrow \quad P \sim_\pi Q$. This direction is a bit more complicated. We define a collection of tests $R(n, L)$ depending on $n \in \omega$ and a finite set L of channel names, and show by induction on n that:

$$\exists L\,(L \supseteq FV(P \mid Q) \text{ and } (P \mid R(n,L)) \overset{\bullet}{\sim} (Q \mid R(n,L))) \quad \Rightarrow \quad P \sim^n Q.$$

If the property above holds then we can conclude the proof by observing:

$$\begin{aligned} P \overset{\bullet}{\sim} Q \quad &\Rightarrow \forall R(P \mid R \overset{\bullet}{\sim} Q \mid R) \\ &\Rightarrow \forall n \in \omega\,(P \mid R(n,L) \overset{\bullet}{\sim} Q \mid R(n,L)) \quad \text{with } L = FV(P \mid Q) \\ &\Rightarrow \forall n \in \omega\,(P \sim^n Q) \\ &\Rightarrow P \sim^\omega Q \\ &\Rightarrow P \sim_\pi Q \quad \text{by exercise 16.1.14 .} \end{aligned}$$

We use an internal sum operator $\oplus$ which is a derived n-ary operator defined as follows:

$$P_1 \oplus \cdots \oplus P_n \equiv \nu a\,(a.P_1 \mid \cdots \mid a.P_n \mid \overline{a}) \quad a \notin FV(P_1 \mid \cdots \mid P_n)\,.$$

We note that $P_1 \oplus \cdots \oplus P_n \xrightarrow{\tau} P_i$ for $i = 1, \ldots, n$. We suppose that the collection of channel names Ch has been partitioned in two infinite well-ordered sets Ch' and Ch''. In the following we have $L \subseteq_{fin} Ch''$. We also assume to have the following sequences of distinct names in Ch':

$$\begin{aligned} &\{a_n \mid n \in \omega\} \\ &\{b_n^\beta \mid n \in \omega \text{ and } \beta \in \{\tau\} \cup \{a, \overline{a} \mid a \in Ch''\}\} \\ &\{b_n'^\beta \mid n \in \omega \text{ and } \beta \in \{a, \overline{a} \mid a \in Ch''\}\}\,. \end{aligned}$$

Commitments on these names permit to control the execution of certain parallel contexts $R(n, L)$ which we define by induction on $n \in \omega$ as follows:

$$\begin{aligned} R(0,L) &= a_0, \quad \text{and for } n > 0 \\ R(n,L) &= a_n \oplus (b_n^\tau \oplus R(n-1,L)) \oplus \\ &\quad \oplus \{b_n^\alpha \oplus (\alpha.(b_n'^\alpha \oplus R(n-1,L))) \mid \alpha \in L \cup \overline{L}\}\,. \end{aligned}$$

We suppose $n > 0$, $FV(P \mid Q) \subseteq L$, $(P \mid R(n,L)) \overset{\bullet}{\sim} (Q \mid R(n,L))$, and $P \xrightarrow{\alpha} P'$. We proceed by case analysis on the action α to show that Q can match the action α. We observe that the parallel contexts $R(n, L)$ can either perform internal reductions (which always cause the loss of a commitment) or offer a communication α.

$\alpha \equiv \tau$ Then:

$$(P \mid R(n,L)) \xrightarrow{\tau} (P \mid (\overline{b}_n^\tau \oplus R(n-1,L)))\,.$$

To match this reduction up to barbed bisimulation we have:

$$(Q \mid R(n,L)) \xrightarrow{\tau} (Q \mid (\overline{b}_n^\tau \oplus R(n-1,L)))\,.$$

We take two steps on the left hand side:

$$(P \mid (\overline{b}_n^{\tau} \oplus R(n-1,L))) \stackrel{\tau}{\rightarrow}{}^2 (P' \mid R(n-1,L)) \ .$$

Again this has to be matched by (we have to lose the $\overline{b}_n^{\tau}$ commitment and $R(n-1,L)$ cannot reduce without losing a commitment):

$$(Q \mid (\overline{b}_n^{\tau} \oplus R(n-1,L))) \stackrel{\tau}{\rightarrow}{}^2 (Q' \mid R(n-1,L)) \ .$$

We observe $Q \stackrel{\tau}{\rightarrow} Q'$. We can conclude by applying the inductive hypothesis.

$\alpha \equiv \overline{a}$ The case $\alpha \equiv a$ is symmetric. We may suppose $a \in L$.

$$(P \mid R(n,L)) \stackrel{\tau}{\rightarrow} (P \mid (\overline{b}_n^a \oplus a.(\overline{b'}_n^a \oplus R(n-1,L)))) \ .$$

To match this reduction up to barbed bisimulation we have:

$$(Q \mid R(n,L)) \stackrel{\tau}{\rightarrow} (Q \mid (\overline{b}_n^a \oplus a.(\overline{b'}_n^a \oplus R(n-1,L)))) \ .$$

We take three steps on the left hand side:

$$(P \mid (\overline{b}_n^a \oplus a.(\overline{b'}_n^a \oplus R(n-1,L)))) \stackrel{\tau}{\rightarrow}{}^3 (P' \mid R(n-1,L)) \ .$$

Again this has to be matched by:

$$(Q \mid (\overline{b}_n^a \oplus a.(\overline{b'}_n^a \oplus R(n-1,L)))) \stackrel{\tau}{\rightarrow}{}^3 (Q' \mid R(n-1,L)) \ .$$

We observe $Q \stackrel{\overline{a}}{\rightarrow} Q'$. We can conclude by applying the inductive hypothesis.

• Next we generalize the definitions in order to deal with the π-calculus. We assume to have the following sequences of distinct names in Ch':

$$\begin{array}{l}
\{b_n, b'_n \mid n \in \omega\} \\
\{c_n^{\beta} \mid n \in \omega \text{ and } \beta \in \{\tau, aa', a, \overline{a}a', \overline{a} \mid a, a' \in Ch''\}\} \\
\{c'^{\beta}_n \mid n \in \omega \text{ and } \beta \in \{aa', a, \overline{a}a', \overline{a} \mid a, a' \in Ch''\}\} \\
\{d_n^{\beta} \mid n \in \omega \text{ and } \beta \in \{a \mid a \in Ch''\}\} \\
\{e_n \mid n \in \omega\} \ .
\end{array}$$

The test $R(n,L)$ is defined by induction on n as follows. When emitting or receiving a name which is not in L, we work up to injective substitution to show that $P \sim^n Q$.

$R(0,L) = \overline{b}_0 \oplus \overline{b'}_0$, and for $n > 0$

$$\begin{array}{l}
R(n,L) = \overline{b}_n \oplus \overline{b'}_n \oplus \\
\quad (\overline{c}_n^{\tau} \oplus R(n-1,L)) \oplus \\
\quad \oplus\{\overline{c}_n^{\overline{a}a'} \oplus (\overline{a}a'.(\overline{c'}_n^{\overline{a}a'} \oplus R(n-1,L)) \mid a, a' \in L\} \oplus \\
\quad \oplus\{\overline{c}_n^{\overline{a}} \oplus \nu a''\,(\overline{a}a''.(\overline{c'}_n^{\overline{a}} \oplus R(n-1, L \cup \{a''\})) \mid a \in L\} \oplus \\
\quad \oplus\{\overline{c}_n^{aa'} \oplus a(a'').(\overline{c'}_n^{aa'} \oplus ([a'' = a']\overline{d}_n^{a'} \oplus R(n-1,L)) \mid a, a' \in L\} \oplus \\
\quad \oplus\{\overline{c}_n^{a} \oplus a(a'').(\overline{c'}_n^{a} \oplus (\oplus\{[a'' = a']\overline{d}_n^{a'} \mid a' \in L\} \oplus \overline{e}_n \oplus R(n-1, L \cup \{a''\})) \mid a \in L\} \ .
\end{array}$$

Here we pick a'' to be the first name in the well-ordered set $Ch''\backslash L$. In order to take into account the exchange of new names between the observed process and the test $R(n, L)$, we have to generalize the statement as follows.

$$\exists L, L'\,(L \supseteq FV(P \mid Q), L' \subseteq L \text{ and } \nu L'\,(P \mid R(n,L)) \overset{\bullet}{\sim} \nu L'\,(Q \mid R(n,L))))$$
$$\Rightarrow\ P \sim^n Q.$$

One can now proceed with an analysis of the possible actions of P mimicking what was done above in the CCS case. □

The proof technique presented here can be extended to the weak case as stated in the following exercise.

Exercise 16.1.21 ** Show that weak barbed bisimulation coincides with the ω-approximation of weak π-bisimulation, and that the latter coincides with π-bisimulation on image finite labelled transition systems (cf. definition 9.2.3, proposition 9.2.8, and exercise 16.1.19).*

Finite control processes. We restrict our attention to processes which are the parallel composition of a finite number of processes defined by a finite system of "regular" recursive equations (we also allow some channels to be restricted). W.l.o.g. we suppose that these equations have the following standard form:

$$A(c_1, \ldots, c_n) = \delta_1.A_1(\vec{c_1}) + \cdots + \delta_m.A_m(\vec{c_m})$$

where the rhs of the equation is taken to be 0 if $m = 0$, and δ_i is either a standard guard or the output of a new channel, say $\nu c\,\overline{d}c$, that we abbreviate as $\overline{d}(c)$. The parameters $\vec{c_j}$ are drawn from either $c_1, \ldots, c_n$ or the bound variable in the prefix δ_j. Our main goal is to show that bisimulation is *decidable* for this class of processes (the argument we give is based on [Dam94]). First, let us consider some processes that can be defined in the fragment of the π-calculus described above.

Example 16.1.22 *(1) The following process models a (persistent) memory cell (we write with $\overline{in}$ and we read with out):*

$$Mem(a) = in(b).Mem(b) + \overline{out}\ a.Mem(a)\ .$$

(2) The system $\nu a\,(G(a) \mid F(a))$ is composed of a new name generator $G(a)$ and a process $F(a)$ that forwards one of the last two names received:

$$\begin{array}{llll} G(a) & = \overline{a}(b).G(a) & F(a) & = a(c).F'(a,c) \\ F'(a,c) & = a(d).F''(a,c,d) & F''(a,c,d) & = \overline{a}c.F(a) + \overline{a}d.F(a)\ . \end{array}$$

Note that if we try to compute the synchronization tree associated to, say, the process $G(a)$ we may end up with an infinite tree in which an infinite number of labels occur. We need some more work to capture the regular behaviour of the process. To fix the ideas suppose that we want to compare two processes P_1, P_2. The process P_i, $i = 1, 2$ consists of the parallel composition of n processes. Each one of these processes is described by a system of m equations, of the shape $A(\vec{c}) = Q$. Always for the sake of simplicity, we suppose that each agent identifier A depends on k parameters. Then the state of the process P_1 is described by a vector:

$$P_1 \equiv \nu\vec{a}\,(A_{j_1}(\vec{c_1}) \mid \cdots \mid A_{j_n}(\vec{c_n}))$$

where $1 \leq j_h \leq m$. The element $A_{j_h}(\vec{c_h})$, for $1 \leq h \leq n$ determines the equation and the parameters being applied at the h-th component. Similarly we suppose that the state of the process P_2 is described by a vector:

$$P_2 \equiv \nu\vec{b}\,(B_{j_1}(\vec{d_1}) \mid \cdots \mid B_{j_n}(\vec{d_n}))\ .$$

The basic restriction that is satisfied by the processes P_i is that recursion does not go through parallel composition. This allows to bound the number of processes running in parallel (in our case the bound is n) and is exploited in proving the following result.

Proposition 16.1.23 *It can be decided if two processes having the structure of P_1 and P_2 above are bisimilar.*

PROOF. Suppose that we compare P_1 and P_2 by applying the definition of π-bisimulation. It is clear that at any moment of the computation each process may depend at most on nk distinct channel names. We may suppose that the free channel names in P_1 and P_2 form an initial segment in the ordering of the channel names (if this is not the case we can always apply an injective substitution). Moreover we identify the process $\nu c\,P$ with the process P whenever $c \notin FV(P)$. Hence the size of the vectors of restricted channels $\vec{a}$ and $\vec{b}$ is bound.

Next we select a set of channel names Δ which is the initial segment of the ordered channel names of cardinality $2nk+1$. There is a finite number of processes of the shape P_1 or P_2 which can be written using names in Δ. So we can find $Pr_1 \subset_{fin} Pr$ such that $P_1, P_2 \in Pr_1$, and if $\Gamma = FV(P \mid P')$ and $\Gamma \vdash P \xrightarrow{\alpha} \Gamma' \vdash P''$ then $P'' \in Pr_1$ up to renaming and elimination of useless restrictions. We observe:

$$P_1 \sim P_2 \ \text{ iff } \ \forall n \in \omega\,(P_1 \sim^n P_2) \ \text{ iff } \ \forall n \in \omega\,(P_1(\sim^n \cap (Pr_1 \times Pr_1))P_2)\ .$$

The sequence $Pr_1 \times Pr_1 \supseteq (\sim^1 \cap (Pr_1 \times Pr_1)) \supseteq \cdots$ converges in a finite number of steps since Pr_1 is finite. □

16.2 A Concurrent Functional Language

Programming languages that combine functional and concurrent programming, such as LCS [BGG91], CML [Rep91] and FACILE [GMP89, TLP+93], are starting to emerge and get applied. These languages are conceived for the programming of reactive systems and distributed systems. A main motivation for using these languages is that they offer integration of different computational paradigms in a clean and well understood programming model that allows formal reasoning about programs' behaviour.

We define a simply typed language, called $\lambda_{||}$, first presented in [ALT95], and inspired by previous work on the FACILE programming language [GMP89, TLP+93, Ama94] whose three basic ingredients are:

- A call-by-value λ-calculus extended with the possibility of parallel evaluation of expressions.
- A notion of channel and primitives to read-write channels in a synchronous way; communications are performed as side effects of expression evaluation.
- The possibility of dynamically generating new channels during execution.

The $\lambda_{||}$-calculus should be regarded as a bridge between programming languages such as FACILE and CML [Rep91] and theoretical calculi such as the π-calculus. To this end it includes abstraction and application among its basic primitives. Benefits of having a direct treatment of abstraction and application include: (i) A handy and well-understood functional fragment is available, this simplifies the practice of programming. (ii) The distinction between sequential reduction and inter-process communication makes more efficient implementations possible. (iii) It is possible to reduce to a minimum the primitives which have to be added to the sequential language, e.g. there is no need of pre-fixing and recursion, and all bindings can be understood as either λ-bindings or ν-bindings. In a slightly different formulation, the latter can be actually reduced to the former, we keep both binders though to simplify the comparison with the π-calculus.

We start by fixing some notation for the $\lambda_{||}$-calculus, ignoring typing issues for the time being. There is a universe of expressions $e, e', \ldots$ inductively generated by the following operators: λ-abstraction $(\lambda x.e)$, application (ee'), parallel composition $(e \mid e')$, restriction $(\nu x\, e)$, output $(e!e')$, and input $(e?)$.

The evaluation of an expression follows a call-by-value order, if the evaluator arrives at an expression of the form $c!v$ or $c?$ (where c is a channel and v is a value) then it is stuck till a synchronization with a parallel expression trying to perform a dual action occurs. As a programming example consider the following functional F that takes two functions, evaluates them in parallel on the number 3 and transmits the product of their outputs on a channel c (we suppose to have natural numbers with the relative product operation $\times$):

$$F \equiv \lambda f.\lambda g.\nu y\,(y!(f3) \mid y!(g3) \mid c!(y? \times y?)) \; .$$

In order to implement the parallel evaluation of $f3$ and $g3$ a local channel y and two processes $y!(f3)$ and $y!(g3)$ are created. Upon termination of the evaluation of, say, $f3$ the value is transmitted to the third process $c!(y? \times y?)$. When both values are received their product is computed and sent on the channel c.

Our first task is to provide the $\lambda_{||}$-calculus with a natural (operational) notion of equivalence. To this end we define the relations of reduction and commitment and build on top of them the notions of barbed bisimulation and equivalence following what was done in section 16.1 for the π-calculus. Our second task is that of showing that there is an adequate translation of the $\lambda_{||}$-calculus into the π-calculus. This serves two goals:

- The encoding of the call-by-value λ-calculus and the transmission of higher-order processes gives a substantial example of the expressive power of the π-calculus.
- It elucidates the semantics of the $\lambda_{||}$-calculus.

A Concurrent λ-calculus. We formally present the $\lambda_{||}$-calculus, define its semantics and illustrate its expressive power by some examples.

- Types are partitioned into values types and *one* behaviour type.

$$\begin{array}{ll} \sigma ::= o \mid (\sigma \multimap \sigma) \mid Ch(\sigma) \mid (\sigma \multimap b) & \text{(value type)} \\ b & \text{(behaviour type)} \\ \alpha ::= \sigma \mid b & \text{(value or behaviour type)} \ . \end{array}$$

- An infinite supply of variables $x^\sigma, y^\sigma, \ldots$, labelled with their type, is assumed for any value type σ. We reserve variables $f, g, \ldots$ for functional types $\sigma \multimap \alpha$. Moreover, an infinite collection of constants $c^\sigma, d^\sigma, \ldots$ is given where σ is either a ground type o or a channel type $Ch(\sigma')$, for some value type σ'. In particular there is a special constant $*^o$. We denote with $z, z', \ldots$ variables or constants.

$$\begin{array}{ll} v & ::= x \mid y \mid \ldots \\ e & ::= c^\sigma \mid v^\sigma \mid \lambda v^\sigma.e \mid ee \mid e!e \mid e? \mid \nu v^{Ch(\sigma)}\, e \mid 0 \mid (e \mid e) \ . \end{array}$$

- Well-typed expressions are defined in figure 16.5. All expressions are considered up to α-renaming. Parallel composition has to be understood as an associative and commutative operator, with 0 as identity. Note that expressions of type behaviour are built up starting with the constant 0, for instance $(\lambda x : o.0)(c^{Ch(\sigma)}!d^\sigma) : b$.

Expressions having a value type are called value expressions and they return a result upon termination. Expressions having type b are called behaviour expressions and they *never* return a result. In particular their semantics is determined only by their interaction capabilities. Since we are in a call-by-value framework it does not make sense to allow behaviours as arguments of a function. The types'

$$(Asmp)\ \frac{}{z^{\sigma} : \sigma} \qquad (0)\ \frac{}{0 : b}$$

$$(\rightarrow_I)\ \frac{e : \alpha}{\lambda x^{\sigma}.e : \sigma \rightharpoonup \alpha} \qquad (\rightarrow_E)\ \frac{e : \sigma \rightharpoonup \alpha \quad e' : \sigma}{ee' : \alpha}$$

$$(!)\ \frac{e : Ch(\sigma) \quad e' : \sigma}{e!e' : o} \qquad (?)\ \frac{e : Ch(\sigma)}{e? : \sigma}$$

$$(\nu)\ \frac{e : \alpha}{\nu x^{Ch(\sigma)}\, e : \alpha} \qquad (|)\ \frac{e : b \quad e' : b}{e \mid e' : b}$$

Figure 16.5: Typing rules for the $\lambda_{||}$-calculus

grammar is restricted accordingly in order to avoid such pathologies. It should be remarked that the interaction capabilities of an expression are not reflected by its type.

Next we describe a rewriting relation (up to structural equivalence) which is supposed to represent abstractly the possible internal computations of a well-typed $\lambda_{||}$-expression. On top of this relation we build a notion of observation, and notions of barbed bisimulation and equivalence.

Definition 16.2.1 *A program is a closed expression of type b.* Values *are specified as follows:* $V ::= c \mid v \mid \lambda v.e$.

In the definition above, variables are values because evaluation may take place under the ν operator. In the implementation these variables can be understood as fresh constants (cf. abstract machine for the π-calculus).

Local evaluation contexts are standard evaluation contexts for call-by-value evaluation (cf. section 8.5). For historical reasons ! and ? are written here in infix and postfix notation, respectively. If one writes them in prefix notation then local evaluation contexts are literally call-by-value evaluation contexts.

$$E ::= [\,] \mid Ee \mid (\lambda v.e)E \mid E!e \mid z!E \mid E? \ .$$

Local evaluation contexts do not allow evaluation under restriction and parallel composition. In order to complete the description of the reduction relation we need to introduce a notion of *global* evaluation context C.

$$C ::= [\,] \mid (\nu v\, C) \mid (C \mid e) \ .$$

Consider the following equations: associativity and commutativity of the parallel composition, $e \mid 0 = e$, and the following laws concerning the the restriction operator ν,

$$(\beta) \quad E[(\lambda x.e)V] \to E[e[V/x]] \qquad (\tau) \quad E[z!V] \mid E'[z?] \to E[*] \mid E'[V]$$

$$(cxt) \quad \frac{e \to e'}{C[e] \to C[e']} \qquad (\equiv) \quad \frac{e \equiv e_1 \;\; e_1 \to e_1' \;\; e_1' \equiv e'}{e \to e'}$$

Figure 16.6: Reduction rules for the $\lambda_{||}$-calculus

$$\begin{array}{ll} (\nu_|) & \nu x\, e \mid e' \equiv_\nu \nu x\, (e \mid e') \quad x \notin FV(e') \\ (\nu_X) & \nu x\, \nu y\, e \equiv_\nu \nu y\, \nu x\, e \\ (\nu_E) & E[\nu x\, e] \equiv_\nu \nu x\, E[e] \quad x \notin FV(E),\ E[e] : b\ . \end{array}$$

We define the relation $\equiv$ as the least equivalence relation on $\lambda_{||}$-expressions that contains the equations above and is closed under global contexts, that is $e \equiv e'$ implies $C[e] \equiv C[e']$. It would be also sensible to ask closure under arbitrary contexts, we do not this to simplify the following comparison with the π-calculus.

Using the notion of local evaluation context two basic reduction rules are defined in figure 16.6. The rule (β) corresponds to local functional evaluation while the rule (τ) describes inter-process communication. The reduction relation describes the internal computation of a program, therefore it is assumed that E, E' have type b. The definition of the rewriting relation is extended to all global contexts by the (cxt) rule (figure 16.6).

The derivation tree associated to a one-step reduction of an expression has the following structure, up to structural equivalence: (i) at most one application of the (cxt) rule, and (ii) one application of one of the basic reduction rules (β) and (τ). We write $e \to_r e'$ if the rule applied in (ii) is $r \in \{\beta, \tau\}$. We observe that by means of structural equivalences it is always possible to display a behaviour expression as follows:

$$\nu x_1 \ldots \nu x_n\, (E_1[\Delta_1] \mid \cdots \mid E_m[\Delta_m])$$

where $n, m \geq 0$, if $m = 0$ then the process can be identified with 0, and $\Delta ::= (\lambda v.e)V \mid z!V \mid z?$. It is interesting to note that purely functional computations always terminate.

Proposition 16.2.2 *Let e be a program. Then all its reduction sequences not involving the communication rule (τ) are finite.*

PROOF. We outline three basic steps. First, we observe that it is enough to prove termination for a calculus having just one channel for every value type.

This allows elimination of restriction. Second, we translate types as follows: $\langle o \rangle = o$; $\langle b \rangle = o$; $\langle \sigma \rightharpoonup \alpha \rangle = \langle \sigma \rangle \to \langle \alpha \rangle$; $\langle Ch(\sigma) \rangle = \langle \sigma \rangle$. Third, we associate to the $\lambda_{||}$-operators $0, ?, !, |$ variables with a suitable types. For instance to $|$ we associate a variable $x_|$ with type $o \to (o \to o)$. It is then possible to translate $\lambda_{||}$ into a simply typed λ-calculus (which is known to be strongly normalizing, cf. theorem 2.2.9). In the translation every β-reduction in $\lambda_{||}$ induces a β-reduction in the translated term. From this one can conclude the termination of every β-reduction sequence in $\lambda_{||}$. □

A fixed point combinator. If we allow τ reductions, then a program in the $\lambda_{||}$-calculus may fail to terminate. Indeed behaviours can be recursively defined by means of a fixed point operator $Y : ((o \rightharpoonup b) \rightharpoonup b) \rightharpoonup b$. This is obtained by a simple simulation of the fixed point combinator for call-by-value (cf. section 8.2) $Y_V \equiv \lambda f.\omega_V\omega_V$ where $\omega_V \equiv \lambda x.f(\lambda w.xx)$. Being in a simply typed framework one expects problems in typing self-application. The way-out is to simulate self-application by a parallel composition of the function and the argument which communicate on a channel of type $o \rightharpoonup b$ (this exploits the fact that all behaviour expressions inhabit the same type). In the following $e!e'$ abbreviates $(\lambda w.0)(e!e')$.

$$Y_b \equiv \lambda f.\nu y\,(\omega_b \mid y!\lambda w.\omega_b) \quad \textit{where} \quad \omega_b \equiv (\lambda x.f(\lambda w.(x* \mid y!x)))y? \ .$$

Using Y_b one may for instance define a behaviour *replicator Rep* e, such that $Rep\ e \approx e \mid Rep\ e$, as follows: $Rep\ e \equiv Y_b(\lambda x^{o \rightharpoonup b}.(x* \mid e))$.

Barbed equivalence. It is easy to adapt the notion of barbed bisimulation and barbed equivalence to the $\lambda_{||}$-calculus. Having already defined the reduction relation it just remains to fix the relation of immediate commitment. The relation $e \downarrow \beta$ where e is a program (cf. definition 16.2.1), $\beta ::= \overline{c} \mid c$, and c is a constant, is defined as follows:

$$e \downarrow \overline{c} \ \text{ if } e \equiv C[E[c!V]] \qquad e \downarrow c \ \text{ if } e \equiv C[E[c?]] \ .$$

As usual let $\to^*$ be the reflexive and transitive closure of $\to$ and define a weak commitment relation $e \downarrow_* \beta$ as $e \downarrow_* \beta$ if $\exists e'\,(e \to^* e' \quad \text{and} \quad e' \downarrow \beta)$. The notions of barbed bisimulation and barbed equivalence are then derived in a mechanic way. A binary relation S between programs is a (weak) barbed simulation if eSf implies: (1) $\forall e'(e \to^* e'$ implies $\exists f'\,(f \to^* f'$ and $e'Sf'))$, and (2) $\forall \beta(e \downarrow_* \beta$ implies $f \downarrow_* \beta)$. S is a barbed bisimulation if S and S^{-1} are barbed simulations. We write $e \overset{\bullet}{\approx} f$ if $e\,S\,f$ for some S barbed bisimulation.

Definition 16.2.3 (congruent equivalence) *Let e, e' be well typed expressions of the $\lambda_{||}$-calculus. Then we write $e \approx e'$ if for all contexts P such that $P[e]$ and $P[e']$ are programs, $P[e] \overset{\bullet}{\approx} P[e']$.*

Remark 16.2.4 *(1) By construction $\approx$ is a congruence with respect to the operators of the calculus. (2) It is easy to prove that if $e : \alpha$ then $(\lambda x.e)V \approx e[V/x]$.*

The following exercises relate the $\lambda_{||}$-calculus to two previously introduced topics: environment machines and continuations, and consider a variant of the calculus based on asynchronous communication.

Exercise 16.2.5 ** Define an abstract machine that executes $\lambda_{||}$-programs by combining the abstract machines defined for the call-by-value λ-calculus (section 8.3) and for the π-calculus (section 16.1).*

Exercise 16.2.6 ** Extend the calculus with a control operator $\mathcal{C}$ (cf. section 8.5) defined according to the following typing and reduction rules:*

$$\frac{e : (\sigma \rightharpoonup b) \rightharpoonup b}{\mathcal{C}e : \sigma} \qquad \frac{}{E[\mathcal{C}e] \to e(\lambda x^{\sigma}.E[x])} \ .$$

The intuition is that the operator $\mathcal{C}$ catches the local evaluation context. Define a CPS *translation from the $\lambda_{||}$-calculus with control operator to the $\lambda_{||}$-calculus.*

Exercise 16.2.7 ** Consider a variant of the $\lambda_{||}$-calculus with an "asynchronous" output operator "$!_a$" (when speaking on the telephone we communicate synchronously, when sending a letter we communicate asynchronously). This calculus can be regarded as a restriction of the $\lambda_{||}$-calculus in which an output is always followed by the terminated process. Typing and reduction are defined as follows:*

$$\frac{e : Ch(\sigma) \quad e' : \sigma}{e!_a e' : b} \qquad \frac{}{E[z?] \mid z!_a V \to E[V]} \ .$$

This calculus can be regarded as a restriction of the $\lambda_{||}$-calculus by writing $e!_a e'$ as $(\lambda x : o.0)(e!e')$. Define a translation from the $\lambda_{||}$-calculus into the corresponding calculus having asynchronous output. Hint: it is convenient to suppose first that the target calculus has the control operator defined in the previous exercise. Then the idea is to translate an input with an asynchronous output: rather than receiving a value one transmits the local evaluation context. Symmetrically one translates an output with an input: rather than transmitting a value one receives the local evaluation context where the value has to be evaluated, say $\langle c!V \rangle = \mathcal{C}(\lambda g.(\lambda f.(f\langle V \rangle \mid g*))c?$, $\langle c? \rangle = \mathcal{C}(\lambda f.c!_a f)$.

Confluent reduction in the π-calculus. We introduce some additional concepts and notations for the π-calculus. As for the $\lambda_{||}$-calculus we assume a constant $*$ of sort o. We may omit writing the process 0. As usual, $\vec{a}$ stands for $a_1, \ldots, a_n$ $(n \geq 0)$. The process $!(c(\vec{a}).P)$ with free variables $\vec{b}$ stands for a recursively defined process $A(\vec{b})$ satisfying the equation $A(\vec{b}) = c(\vec{a}).(P \mid A(\vec{b}))$, where $\{\vec{a}\} \cap \{\vec{b}\} = \emptyset$. The operator $!$ is traditionally called *replication*. For the sake of simplicity we also assume the following equivalences concerning the replication

operator. The first allows for the unfolding of the replication operator and the second entails the garbage collection of certain deadlocked processes.

$$(\check{!}_1)\ \ \check{!}P \equiv P \mid \check{!}P \qquad (\check{!}_2)\ \ \nu x\, \check{!}(x(\vec{y}).P) \equiv 0 \tag{16.5}$$

We will consider structurally congruent two π-terms which are in the congruent closure of the equations 16.5. It is useful to identify certain special reductions which enjoy an interesting *confluence* property.

Definition 16.2.8 *Administrative reductions: We write $Q \rightarrow_{ad} Q'$ if for some context D with one hole, $Q \equiv D[\nu u\,(\overline{u}\vec{z} \mid u(\vec{x}).P)]$ and $Q' \equiv D[P[\vec{z}/\vec{x}]]$, where $u \notin FV(P \cup \{\vec{z}\})$.*

Beta reductions: We write $Q \rightarrow_{beta} Q'$ if for some context D with one hole, $Q \equiv D[\nu f\,(\overline{f}\vec{z} \mid \check{!}(f(\vec{x}).P) \mid P')]$ and $Q' \equiv D[\nu f\,(P[\vec{z}/\vec{x}] \mid \check{!}(f(\vec{x}).P) \mid P')]$, where $f \notin FV(P)$, and f cannot occur free in P' in input position, that is as $f(\vec{y}).P''$.

We note that administrative reductions always terminate. Moreover we observe the following confluence property.

Proposition 16.2.9 *Suppose $P \rightarrow P_1$ and $P \rightarrow_{ad,beta} P_2$. Then either $P_1 \equiv P_2$ or there is P' such that $P_1 \rightarrow_{ad,beta} P'$ and $P_2 \rightarrow P'$.*

PROOF HINT. By a simple analysis of the relative positions of the redexes. In particular, we note that if two *beta*-reductions superpose then they both refer to the same replicated receiving subprocess. □

Translation. We introduce a translation of the $\lambda_{||}$-calculus into the π-calculus and we discuss some of the basic properties of the translation. Notably, we produce an optimized translation to which the standard translation reduces by means of administrative reductions. The basic problem is that of finding a simulation of function transmission by means of channel transmission. The idea is that rather than transmitting a function one transmits a pointer to a function (a channel) and at the same time one "stores" the function by means of the *replication* operator. Let us consider the following reduction sequence in $\lambda_{||}$:

$$c!(\lambda x.e) \mid (\lambda f.(fn \mid fm))c? \rightarrow^{+} [n/x]e \mid [m/x]e\ .$$

Supposing that there is some translation $\lceil\ \rceil$ such that:

$$\lceil c!(\lambda x.e)\rceil = \nu f\,(\overline{c}f \mid \check{!}(f(x).\lceil e\rceil)) \qquad \lceil(\lambda f.(fn \mid fm))c?\rceil = c(f).(\overline{f}n \mid \overline{f}m)\ .$$

Then by parallel composition of the translations the following simulating reduction sequence in the π-calculus is obtained:

$$\nu f\,(\overline{c}f \mid \check{!}(f(x).\lceil e\rceil)) \mid c(f).(\overline{f}n \mid \overline{f}m) \ \rightarrow\ \nu f\,(\check{!}(f(x).\lceil e\rceil) \mid \overline{f}n \mid \overline{f}m) \ \rightarrow^{+}$$
$$\nu f\,(\check{!}(f(x).\lceil e\rceil) \mid [n/x]\lceil e\rceil \mid [m/x]\lceil e\rceil) \approx \lceil [n/x]e \mid [m/x]e\rceil\ .$$

Definition 16.2.10 (type translation) *A function* $\lceil\ \rceil$ *from* $\lambda_{||}$ *types into* π *sorts is defined as follows:*

$$\begin{array}{ll} \lceil o \rceil = o & \lceil Ch(\sigma) \rceil = Ch(\lceil \sigma \rceil) \\ \lceil \sigma \to \sigma' \rceil = Ch(\lceil \sigma \rceil, Ch(\lceil \sigma' \rceil)) & \lceil \sigma \to b \rceil = Ch(\lceil \sigma \rceil, Ch()) \ . \end{array}$$

In figure 16.7 a function $\lceil\ \rceil$ from well-typed $\lambda_{||}$-expressions into well-sorted π-processes is defined. It is possible to statically assign one out of three "colours" to each π-variable involved in the translation. The colours are used to make the functionality of a channel explicit and classify the possible reductions of translated terms. To this end, we suppose that in the expression e to be translated all variables of functional type $\sigma \rightharpoonup \alpha$ are represented with $f, g, \ldots$ Variables of channel or ground sort in the $\lambda_{||}$-term are represented by $x, y, \ldots$ and channels used for "internal book keeping" in the translation are represented by $u, t, v, \ldots$ Thus we suppose that π-variables are partitioned in three infinite sets: $u, t, w, \ldots$; $f, g, \ldots$; and $x, y, \ldots$ Furthermore, we let $r, r', \ldots$ ambiguously denote constants $(c, c', \ldots)$, variables $(x, y, \ldots)$, and variables $(f, g, \ldots)$.

The translation is parameterized over a (fresh) channel u. If $e : \sigma$ is a value expression then u has sort $Ch(\lceil \sigma \rceil)$ and it is used to transmit the value (or a pointer to the value) resulting from the evaluation of the expression e. If $e : b$ is a behaviour expression then u is actually of *no use*, we conventionally assign the sort $Ch()$ to the channel u (we choose to parameterize the behaviour expressions too in order to have a more uniform notation). Each rule using variables r actually stands for *two* rules, one in which r is replaced by a variable $x, y, \ldots$ or a constant and another where it is replaced by a variable $f, g, \ldots$ In the translation only the variables $x, y, \ldots$ can be instantiated by a constant. Note the use of polyadic channels in the translation of λ-abstraction.

As expected, reductions in the $\lambda_{||}$-calculus are implemented by several reductions in the π-calculus. The need for a finer description of the computation in the π-calculus relates to two aspects:

(1) In the π-calculus there is no notion of application. The implicit order of evaluation given by the relative positions of the expressions in the $\lambda_{||}$-calculus has to be explicitly represented in the π-calculus. In particular the "computation" of the evaluation context is performed by means of certain administrative reductions.

(2) In the π-calculus it is not possible to transmit functions. Instead, a pointer to a function which is stored in the environment by means of the replication operator is transmitted. (There is an analogy with graph reduction of functional languages, see [Bou93] for a discussion).

Before analysing the encoding of call-by-value we hint to the encoding of call-by-name, as defined in figure 8.6. The translation is given parametrically with respect to a fresh name a which should be interpreted as the channel on which

$$
\begin{array}{ll}
\lceil r \rceil u & = \overline{u} r \\
\lceil \lambda r.e \rceil u & = \nu f\,(\overline{u} f \mid \lceil f := \lambda r.e \rceil) \\
 & \text{where } \lceil f := \lambda r.e \rceil = \,!(f(r,w).\lceil e \rceil w) \\
\lceil e e' \rceil u & = \nu t\,\nu w\,(\lceil e \rceil t \mid t(f).(\lceil e' \rceil w \mid w(r).\overline{f}(r,u))) \\
\lceil e ! e' \rceil u & = \nu t\,\nu w\,(\lceil e \rceil t \mid t(x).(\lceil e' \rceil w \mid w(r).\overline{x} r.\overline{u} *)) \\
\lceil e ? \rceil u & = \nu t\,(\lceil e \rceil t \mid t(x).x(r).\overline{u} r) \\
\lceil \nu x\, e \rceil u & = \nu x\, \lceil e \rceil u \\
\lceil 0 \rceil u & = 0 \\
\lceil e \mid e' \rceil u & = \lceil e \rceil u \mid \lceil e' \rceil u
\end{array}
$$

Figure 16.7: Expression translation

the term will receive a pair consisting of (a pointer to) its next argument, and the channel name on which to receive the following pair.

$$
\begin{array}{ll}
[\![x]\!] a & = \overline{x} a \\
[\![\lambda x.M]\!] a & = a(x,b).[\![M]\!] b \\
[\![MN]\!] a & = \nu b\,\nu c\,([\![M]\!] b \mid \overline{b}(c,a) \mid \,!(c(d).[\![N]\!] d))\,.
\end{array}
$$

Exercise 16.2.11 ** Prove that* $[\![(\lambda x.M)N]\!] a \approx [\![M[N/x]]\!] a$.

For more results on this translation we refer to [San92, BL94] where a characterization of the equivalence induced by the π-calculus encoding on λ-terms can be found. Related work on the representation of (higher-order) processes in the π-calculus can be found in [Mil92, Ama93, San92, Tho93]. The following is a challenging programming exercise.

Exercise 16.2.12 ** Define a translation of the* $\lambda_{\sqcup}$*-calculus (section 8.4) in the* π*-calculus which simulates reduction.*

We now turn to a detailed analysis of the π-calculus encoding for call-by-value. This requires the introduction of some technical definitions.

Definition 16.2.13 *Given a process* P *in the* π*-calculus let* $\sharp P$ *be its normal form with respect to administrative reductions on channels coloured* $u, t, w, \ldots$ *A binary relation* R *between programs in* $\lambda_{||}$*-calculus and programs in the* π*-calculus is defined as follows, where* u *is some fresh channel,* V_i *are* λ*-abstractions and the substitution is iterated from left to right, as* V_j *may depend on* f_i *for* $i < j$.

$$
\begin{array}{l}
e[V_n/f_n] \cdots [V_1/f_1]\ R\ P \quad \textit{if} \\
\sharp P \equiv \sharp \nu f_1 \ldots \nu f_n\,(\lceil e \rceil u \mid \lceil f_1 := V_1 \rceil \mid \cdots \mid \lceil f_n := V_n \rceil)\,.
\end{array}
$$

The relative complexity of the definition of the relation R relates to the points (1-2) above. The translated term may need to perform a certain number of administrative reductions before a reduction corresponding to a reduction of the $\lambda_{||}$-calculus emerges. We get rid of these administrative reductions by introducing the notion of normal form $\sharp P$. A second issue concerns the substitution of a value for a variable which in the π-calculus is simulated by the substitution of a pointer to a value for a variable. Therefore, we have to relate, say, the term $e[V/f]$ with the term $\nu f.(\lceil e \rceil \mid \lceil f := V \rceil)$. It will be convenient to use the following abbreviations:

$$\begin{array}{lll} \nu\vec{f} & \text{stands for} & \nu f_1 \ldots \nu f_n \\ \lceil \vec{f := V} \rceil & \text{stands for} & \lceil f_1 := V_1 \rceil \mid \cdots \mid \lceil f_n := V_n \rceil \\ [\vec{V/f}] & \text{stands for} & [V_n/f_n] \cdots [V_1/f_1] \quad (n \geq 0) \; . \end{array}$$

In order to analyse the structure of $\sharp\nu\vec{f}(\lceil e \rceil u \mid \lceil \vec{f := V} \rceil)$ we define an *optimized* translation. The optimization amounts to pre-computing the initial administrative steps of the translation (a similar idea was applied in section 8.5). To this end, we define an open redex and an open evaluation context, as a redex and an evaluation context, respectively, in which a functional variable may stand for a value (cf. definition 16.2.1). For instance, fV is an open redex, and fE is an open evaluation context. In this way we can speak about redexes which arise only after a substitution is carried on.

We note that if $e[\vec{V/f}] \equiv E[\Delta]$ then $e \equiv E'[\Delta']$, where Δ', E' are open redex and evaluation context, respectively, and $\Delta'[\vec{V/f}] \equiv \Delta$, $E'[\vec{V/f}] \equiv E$. This remark is easily extended to the case where:

$$e[\vec{V/f}] \equiv \nu\vec{x}\,(E_1[\Delta_1] \mid \cdots \mid E_n[\Delta_n]) \; .$$

In the following definitions and proofs the reader may at first skip the part involving the input-output operators and concentrate on the λ-calculus fragment of the $\lambda_{||}$-calculus.

Definition 16.2.14 (open context translation) *The translation is defined on open contexts E such that $E \neq [\,]$. We assume that V is a λ-abstraction.*

$$\begin{array}{ll} \langle [u']e \rangle u & = \nu w\,(u'(f).\lceil e \rceil w \mid w(r).\overline{f}(r,u)) \\ \langle V[u'] \rangle u & = \nu f\,(\lceil f := V \rceil \mid u'(r).\overline{f}(r,u)) \\ \langle f[u'] \rangle u & = u'(r).\overline{f}(r,u) \\ \langle [u']?\rangle u & = u'(x).x(r).\overline{u}r \\ \langle [u']!e \rangle u & = \nu w\,(u'(x).(\lceil e \rceil w \mid w(r).\overline{x}r.\overline{u}*)) \\ \langle z![u'] \rangle u & = u'(r).\overline{z}r.\overline{u}* \\ \langle E[u']e \rangle u & = \nu t\,\nu w\,(\langle E[u'] \rangle t \mid t(f).\lceil e \rceil w \mid w(r).\overline{f}(r,u)) \\ \langle VE[u'] \rangle u & = \nu w\,\nu f\,(\lceil f := V \rceil \mid \langle E[u'] \rangle w \mid w(r).\overline{f}(r,u)) \\ \langle fE[u'] \rangle u & = \nu w\,(\langle E[u'] \rangle w \mid w(r).\overline{f}(r,u)) \end{array}$$

$$\begin{array}{ll}\langle E[u']?\rangle u & = \nu w\,(\langle E[u']\rangle w \mid w(x).x(r).\overline{u}r)\\ \langle E[u']!e\rangle u & = \nu t\,\nu w\,(\langle E[u']\rangle t \mid t(x).\lceil e\rceil w \mid w(r).\overline{x}r.\overline{u}*)\\ \langle z!E[u']\rangle u & = \nu t\,(\langle E[u']\rangle t \mid t(r).\overline{z}r.\overline{u}*)\quad .\end{array}$$

Lemma 16.2.15 (administrative reductions) *Suppose that E is an open evaluation context such that $E \neq [\,]$, and that V_j are λ-abstractions which may depend on f_i for $i < j$. Then:*

$$\nu\vec{f}\,(\lceil E[e]\rceil u \mid \lceil f := \vec{V}\rceil) \to^*_{ad} \nu\vec{f}\,\nu u'\,(\lceil e\rceil u' \mid \langle E[u']\rangle u \mid \lceil f := \vec{V}\rceil)\ .$$

Proof. By induction on the structure of the evaluation context. There are 12 cases to consider, following the context translation in the definition 16.2.14 above. We present two typical cases for illustration.

Case Ee_1.
$\nu\vec{f}\,(\lceil E[e]e_1\rceil u \mid \lceil f := \vec{V}\rceil) \equiv$
$\nu\vec{f}\,\nu t\,\nu w\,(\lceil E[e]\rceil t \mid t(f).\lceil e_1\rceil w \mid w(r).\overline{f}(r,u) \mid \lceil f := \vec{V}\rceil) \to^*_{ad}$ (by ind. hyp.)
$\nu\vec{f}\,\nu t\,\nu w\,\nu u'\,(\lceil e\rceil u' \mid \langle E[u']\rangle t \mid t(f).\lceil e_1\rceil w \mid w(r).\overline{f}(r,u) \mid \lceil f := \vec{V}\rceil) \equiv$
$\nu\vec{f}\,\nu u'\,(\lceil e\rceil u' \mid \langle E[u']e_1\rangle u \mid \lceil f := \vec{V}\rceil)\ .$

Case VE.
$\nu\vec{f}\,(\lceil VE[e]\rceil u \mid \lceil f := \vec{V}\rceil) \equiv$
$\nu\vec{f}\,\nu t\,\nu w\,\nu f\,(\lceil f := V\rceil \mid \overline{t}f \mid t(f).\lceil E[e]\rceil w \mid w(r).\overline{f}(r,u) \mid \lceil f := \vec{V}\rceil) \to_{ad}$
$\nu\vec{f}\,\nu t\,\nu w\,\nu f\,(\lceil f := V\rceil \mid \lceil E[e]\rceil w \mid w(r).\overline{f}(r,u) \mid \lceil f := \vec{V}\rceil) \to^*_{ad}$ (by ind. hyp.)
$\nu\vec{f}\,\nu t\,\nu w\,\nu f\,\nu u'\,(\lceil f := V\rceil \mid \lceil e\rceil u' \mid \langle E[u']\rangle w \mid w(r).\overline{f}(r,u) \mid \lceil f := \vec{V}\rceil) \equiv$
$\nu\vec{f}\,\nu u'\,(\lceil f := V\rceil \mid \lceil e\rceil u' \mid \langle VE[u']\rangle w \mid \lceil f := \vec{V}\rceil)\ .$

□

Remark 16.2.16 *(1) From the previous lemma 16.2.15 we can prove that if $e \equiv e'$ then $\sharp\lceil e\rceil u \equiv \sharp\lceil e'\rceil u$. (2) Lemma 16.2.15 immediately extends to a general behaviour expression $e \equiv \nu\vec{x}\,(E_1[\Delta_1] \mid \cdots \mid E_m[\Delta_m])$ as the expression translation distributes with respect to restriction and parallel composition.*

The following translation pre-computes the administrative reductions in an open redex and it is needed in the following proposition.

Definition 16.2.17 (open redex translation) *In the following open redex translation we assume that V is a λ-abstraction.*

$$\begin{array}{ll}\{(\lambda r'.e)r\}u & = \nu f\,(\lceil f := \lambda r'.e\rceil \mid \overline{f}(r,u))\\ \{(\lambda f'.e)V\}u & = \nu f\,\nu f'\,(\lceil f := \lambda f'.e\rceil \mid \lceil f' := V\rceil \mid \overline{f}(f',u))\\ \{fr\}u & = \overline{f}(r,u)\end{array}$$

$$\begin{array}{ll} \{fV\}u & = \nu f'\,(\lceil f' := V\rceil \mid \overline{f}(f', u)) \\ \{z?\}u & = z(r).\overline{u}r \\ \{z!r\}u & = \overline{z}r.\overline{u}* \\ \{z!V\}u & = \nu f'\,(\lceil f' := V\rceil \mid \overline{z}f'.\overline{u}*) \quad . \end{array}$$

Proposition 16.2.18 *The* administrative normal form *of the behaviour expression* $e \equiv \nu\vec{x}\,(E_1[\Delta_1] \mid \cdots \mid E_m[\Delta_m])$ *can be characterized as (supposing* $E_i \neq [\,]$*, for* $i = 1, \ldots, m$*, otherwise just drop the context translation):*

$$\sharp\lceil e\rceil u \equiv \nu\vec{x}\,\nu u_1 \ldots u_m\,(\{\Delta_1\}u_1 \mid \langle E_1[u_1]\rangle u \mid \cdots \mid \{\Delta_1\}u_m \mid \langle E_m[u_m]\rangle u) \ .$$

Proof hint. By remark 16.2.16 and the observation that translations of open evaluation contexts and redexes do not admit administrative reductions. □

With the help of the optimized translation described above, we derive the following lemma, which relates reductions and commitments modulo the relation R.

Lemma 16.2.19 *The following assertions relate reductions and commitments:*

(1) *If* eRP *and* $e \to e'$ *then* $\sharp P \to P'$ *and* $e'RP'$.

(2) *Vice versa, if* eRP *and* $\sharp P \to P'$ *then* $e \to e'$ *and* $e'RP'$.

(3) *Suppose* eRP. *Then* $e \downarrow \beta$ *iff* $\sharp P \downarrow \beta$.

Proof. (1) By analysis of the redex, following the open redex translation in definition 16.2.17. We consider only two cases which should justify the definition of the relation R. As a first case suppose eRP and:

$$e \equiv E[(\lambda f'.e)V][\vec{V}/\vec{f}] \to_\beta E[e[V/f']][\vec{V}/\vec{f}] \equiv e'$$

$$\sharp P \equiv \sharp\nu\vec{f}\,(\lceil E[(\lambda f'.e)V]\rceil u \mid \lceil f :\vec{=} V\rceil) \ .$$

Then $\sharp P \to_{beta} P'$, where:

$$P' \equiv \nu\vec{f}\,\nu f'\,\nu u'\,(\lceil e\rceil u' \mid \langle E[u']\rangle u \mid \lceil f' := V\rceil \mid \lceil f :\vec{=} V\rceil)$$

and observe $e'RP'$, since $e' \equiv E[e][V/f'][\vec{V}/\vec{f}]$ and by the administrative reduction lemma 16.2.15:

$$\sharp P' \equiv \sharp\nu\vec{f}\,\nu f'\,(\lceil E[e]\rceil u \mid \lceil f' := V\rceil \mid \lceil f :\vec{=} V\rceil) \ .$$

As a second case suppose eRP and:

$$e \equiv (E[c!f_i] \mid E'[c?])[\vec{V}/\vec{f}] \to_\beta (E[*] \mid E'[f_i])[\vec{V}/\vec{f}] \equiv e' \ .$$

$$\sharp P \equiv \sharp\nu\vec{f}\,(\lceil E[c!f_i]\rceil u \mid \lceil E'[c?]\rceil u \mid \lceil f :\vec{=} V\rceil) \ .$$

Then $\sharp P \rightarrow_\tau P'$, where:

$$P' \equiv \nu \vec{f}\, \nu u_1\, \nu u_2\, (\lceil * \rceil u_1 \mid \langle E[u_1] \rangle u \mid \lceil f_i \rceil u_2 \mid \langle E'[u_2] \rangle u \mid \lceil f := \vec{V} \rceil)$$

and observe $e' R P'$, since by the administrative reduction lemma 16.2.15:

$$\sharp P' \equiv \sharp \nu \vec{f}\, (\lceil E[*] \mid E'[f_i] \rceil u \mid \lceil f := \vec{V} \rceil)\ .$$

(2) Same analysis as in (1). (3) This follows by the definition of the relation R and by the characterization of the administrative reduction normal form. □

Theorem 16.2.20 *Let e, e' be programs in $\lambda_{||}$. Then $\lceil e \rceil u \stackrel{\bullet}{\approx} \lceil e' \rceil u$ iff $e \stackrel{\bullet}{\approx} e'$.*

PROOF. The previous lemma 16.2.19 allows to go back and forth between (weak) reductions and (weak) commitments "modulo R". Hence one can define the following relations and show that they are barbed bisimulations.

$$\begin{aligned} S &= \{(e, e') \mid \exists P, P'\, (e\, R\, P \stackrel{\bullet}{\approx} P'\, R^{-1}\, e')\} \\ S' &= \{(P, P') \mid \exists e, e'\, (P\, R^{-1}\, e \stackrel{\bullet}{\approx} e'\, R\, P')\}\ . \end{aligned}$$

□

Unfortunately, this result does not extend to barbed equivalence, as there are equivalent $\lambda_{||}$-terms whose π-calculus translations can be distinguished. The relationships between λ-calculus and π-calculus remain to be clarified. For instance it is not known whether there is a "natural" fully-abstract translation of the call-by-value λ-calculus into the π-calculus, or in another direction, whether there is a "reasonable" extension of the λ-calculus that would make the translation considered here fully-abstract.

Appendix A

Memento of Recursion Theory

In this memento, functions are always partial, unless otherwise specified. The symbol $\downarrow$ ($\uparrow$) is used for "is defined" ("is undefined"). A definition of the form "$f(x) \downarrow y$ iff P" has to be read "$f(x) \downarrow$ iff P, and P implies $f(x) = y$". We abbreviate $x_1, \ldots, x_n$ into $\vec{x}$. We also write, for two expressions s and t,

$$s \cong t \quad \text{iff} \quad (s \downarrow \text{ and } t \downarrow \text{ and } s = t) \text{ or } (s \uparrow \text{ and } t \uparrow).$$

A.1 Partial Recursive Functions

Partial recursive, or computable functions, may be defined in a number of equivalent ways. This is what Church's thesis is about: all definitions of computability turn out to be equivalent. Church's thesis justifies some confidence in "semi-formal" arguments, used to show that a given function is computable. These arguments can be accepted only if at any moment, upon request, the author of the argument is able to fully formalize it in one of the available axiomatizations. The most basic way of defining computable functions is by means of computing devices of which Turing machines are the most well known. A given Turing machine defines, for each n, a partial function $f : \omega^n \to \omega$. More mathematical presentations are by means of recursive program schemes, or by means of combinations of basic recursive functions.

Theorem A.1.1 (Gödel-Kleene) *For any n, the set of Turing computable functions from ω^n to ω is the set of partial recursive functions from ω^n to ω, where by definition the class of partial recursive (p.r.) functions is the smallest class containing:*

- $0 : \omega \to \omega$ *defined by* $0(x) = 0$.
- $succ : \omega \to \omega$ *(the successor function).*
- *Projections* $\pi_{n,i} : \omega^n \to \omega$ *defined by* $\pi_{n,i}(x_1, \ldots, x_n) = x_i$.

and closed under the following constructions:

• *Composition: If* $f_1 : \omega^m \to \omega, \ldots, f_n : \omega^m \to \omega$ *and* $g : \omega^n \to \omega$ *are p.r., then* $g \circ \langle f_1, \ldots, f_n \rangle : \omega^m \to \omega$ *is p.r..*

• *Primitive recursion: if* $f : \omega^n \to \omega$, $g : \omega^{n+2} \to \omega$ *are p.r., then so is* h *defined by:*

$$\begin{array}{ll} h(\vec{x}, 0) & = f(\vec{x}) \\ h(\vec{x}, y+1) & = g(\vec{x}, y, h(\vec{x}, y)) \ . \end{array}$$

• *Minimalisation: if* $f : \omega^{n+1} \to \omega$ *is p.r., so is* $g : \omega^n \rightharpoonup \omega$ *defined by* $g(\vec{x}) = \mu y.(f(\vec{x}, y) = 0)$, *where* $\mu y.P$ *means: the smallest* y *such that* P.

The source of partiality lies in minimalisation. The total functions obtained by the combinations of Gödel-Kleene, except minimalisation, are called primitive recursive. The partial recursive functions which are total are called the recursive functions. The set of partial recursive functions from ω^n to ω is called PR^n (we write PR for PR^1).

Lemma A.1.2 (encoding of pairs) *The following functions are recursive and provide inverse bijections between* $\omega \times \omega$ *and* ω.

$\langle \_, \_ \rangle : \omega \times \omega \to \omega$ *defined by:* $\langle m, n \rangle = 2^m(2n+1) - 1$.
$\pi_1 : \omega \to \omega$ *where* $\pi_1(n)$ *is the exponent of* 2 *in the prime decomposition of* $n+1$.
$\pi_2 : \omega \to \omega$ *defined by:* $\pi_2(n) = ((n+1)/2^{\pi_1(n)} - 1)/2$.

We say that a function $f : \omega \times \omega \to \omega$ is p.r. iff $f \circ \langle \pi_1, \pi_2 \rangle : \omega \to \omega$ is p.r.. Turing machines can also be coded by natural numbers (a Turing machine is determined by a finite control which can be described by a finite string on a finite alphabet which in turn can be represented by a natural number). We call:

T_n the Turing machine which has code n.
ϕ_n^m the partial function from ω^m to ω defined by T_n (we write ϕ_n for ϕ_n^1).
$W_n^m = dom(\phi_n^m)$ (we write W_n for W_n^1).

If $f = \phi_n^m$ ($W = W_n^m$), we say that n is an index of f (W).

Lemma A.1.3 (enumeration of PR**)** *The mapping* $\lambda n.\phi_n$ *is a surjection of* ω *onto* PR.

As a first consequence, there are total functions which are not recursive.

Exercise A.1.4 *Show that* f *defined by*

$$f(n) = \begin{cases} \phi_n(n) + 1 & \text{if } \phi_n(n) \downarrow \\ 0 & \text{otherwise} \end{cases}$$

is not recursive. (But g *defined by* $g(n) \downarrow \phi_n(n) + 1$ *iff* $\phi_n(n) \downarrow$ *is p.r, see following theorem A.1.6). Show that there exist recursive, non primitive recursive functions. Hint: For the last part use an enumeration* $\{\theta_n\}_{n \in \omega}$ *of the primitive recursive functions, and take* $\lambda x.\theta_x(x) + 1$.

The next theorem says that arguments of a partial recursive function can be frozen, uniformly.

Theorem A.1.5 (s-m-n) *For each m, n there is a total recursive $m+1$ ary function s_n^m (s for short) such that for all $\vec{x} = x_1, \ldots, x_m$, $\vec{y} = y_{m+1}, \ldots, y_{m+n}$ and p : $\phi_p^{m+n}(\vec{x}, \vec{y}) \cong \phi_{s(p,\vec{x})}^n(\vec{y})$.*

PROOF HINT. We can "prefix" to T_p instructions that input the frozen argument $\vec{x}$. □

Theorem A.1.6 (universal Turing machine) *There exists a Turing machine T_U computing, for any n, the function $\psi_U^n : \omega^{n+1} \to \omega$ defined by: $\psi_U^n(p, \vec{y}) = \phi_p^n(\vec{y})$.*

PROOF HINT. Informally, T_U decodes its first argument p into the machine T_p, and then acts as T_p on the remaining arguments. □

A.2 Recursively Enumerable Sets

The theory of computable functions can be equivalently be presented as a theory of computable predicates.

Definition A.2.1 (decidable and semi-decidable) *A subset W of ω^n is called* decidable, *or* recursive, *when its characteristic function χ defined by*

$$\chi(x) = \begin{cases} 0 & \text{if } x \in W \\ 1 & \text{otherwise} \end{cases}$$

is recursive. A subset W of ω^n is called primitive recursive, *when its characteristic function is primitive recursive. A subset W of ω^n is called* semi-decidable, *or* recursively enumerable *(r.e.), when its partial characteristic function χ_p ($\chi_p(x) \downarrow$ iff $x \in W$) is partial recursive.*

Clearly, every decidable set is semi-decidable. A central example of a recursive set is the following.

Proposition A.2.2 (convergence in t steps) *Given a Turing machine T computing the partial recursive function f, the set $\{(\vec{x}, y, t) \mid f(\vec{x}) \downarrow y$ in t steps of $T\}$ is recursive.*

PROOF. Given a Turing machine T computing f, the obvious informal algorithm is: perform at most t steps of T starting with input $\vec{x}$, and check whether result y has been reached. □

Remark A.2.3 *A more careful analysis shows that the characteristic function of* $\{(\vec{x}, y, t) \mid f(\vec{x}) \downarrow y \text{ in } t \text{ steps}\}$ *can be defined by means of primitive recursion only.*

There are a number of equivalent characterizations of recursive and recursively enumerable sets.

Proposition A.2.4 *The set* $W \subseteq \omega^n$ *is r.e. iff one of the following conditions holds:*

(1) $W = dom(f)$, *for some partial recursive function* f.

(2) *There exists a recursive set* $W' \subseteq \omega^{n+1}$ *such that* $W = \{\vec{x} \mid \exists y(\vec{x}, y) \in W'\}$.

(3) $W = \emptyset$ *or* $W = im(h)$, *for some recursive function* $h : \omega^n \to \omega$.

(4) $W = im(h)$, *for some partial recursive function* $h : \omega^n \to \omega$.

PROOF. (1) If $W = dom(f)$, then its partial characteristic function is $\mathbf{1} \circ f$, where $\mathbf{1}$ is constant 1.

(2) Let W be $\{\vec{x} \mid \exists y(\vec{x}, y) \in W'\}$. Then $W = dom(\lambda\vec{x}.\mu y.((\vec{x}, y) \in W'))$. Conversely, if $W = dom(f)$, take $W' = \{(\vec{x}, y) | f(\vec{x}) \downarrow \text{ in } y \text{ steps}\}$.

(3) If $W = dom(f) \neq \emptyset$, pick an element $\vec{a} \in W$. Define:

$$g(\vec{x}, y) = \begin{cases} \vec{x} & \text{if } f(\vec{x}) \downarrow \text{ in } y \text{ steps} \\ \vec{a} & \text{otherwise} \end{cases}$$

Then $W = im(g) = im(h)$ (where h is the composition of g with the encoding from ω^n to ω^{n+1}).

(4) If $W = im(h)$, we have by proposition A.2.2 that $\{(\vec{x}, y, t) \mid h(\vec{x}) \downarrow y \text{ in } t \text{ steps}\}$ is recursive. Thus $W = dom(\lambda\vec{x}.\mu z.(z = \langle y, t\rangle \text{ and } h(\vec{x}) \downarrow y \text{ in } t \text{ steps}))$. □

Remark A.2.5 *The encodings quoted among others in the proof of proposition A.2.4(3) "hide" a useful technique, known as* dovetailing*: the informal way of obtaining* h *is by trying the first step of* $f(1)$*, the first step of* $f(2)$*, the second step of* $f(1)$*, the first step of* $f(3)$*, the second step of* $f(2)$*, the third step of* $f(1)$*, the first step of* $f(4)\ldots$

Exercise A.2.6 *Show that if* $W \subseteq \omega^{n+1}$ *is r.e., then* $\{\vec{x} \mid \exists y, (\vec{x}, y) \in W\}$ *is r.e.. Hint: Consider a recursive* W' *such that* $W = \{(\vec{x}, y) | \exists z(\vec{x}, y, z) \in W'\}$ *is r.e. .*

Proposition A.2.7 $W \subseteq \omega^n$ *is recursive iff* W *and its complement* W^c *are semi-decidable.*

PROOF. If W is decidable, it is semidecidable, and W^c is decidable (with characteristic function $\neg \circ \chi$, where χ is the characteristic function of W). Conversely, if W and W^c are both semi-decidable, let W' and W'' be recursive and such that

$$W = \{\vec{x} \mid \exists y(\vec{x}, y) \in W'\} \quad W^c = \{\vec{x} \mid \exists y(\vec{x}, y) \in W''\} .$$

Let χ' and χ'' be the characteristic functions of W' and W'', respectively. Then

$$W = dom(\lambda \vec{x}.\mu y.(Y \circ \langle \chi', \chi'' \rangle(\vec{x}, y) = 0))$$

where Y is any recursive function restricting to the boolean union over $\{0, 1\}$. The function $\lambda \vec{x}.\mu y.(Y \circ \langle \chi', \chi'' \rangle(\vec{x}, y) = 0))$ is p.r. by construction, and moreover is total since $W \cup W^c = \omega^n$. □

The following is a useful characterization of partial recursive functions.

Proposition A.2.8 *A function f is p.r. iff its graph $\{(\vec{x}, y) \mid f(\vec{x}) \downarrow y\}$ is r.e.*

PROOF. If f is p.r., then by proposition A.2.2 $\{(\vec{x}, y, t) \mid f(\vec{x}) \downarrow y \text{ in } t \text{ steps}\}$ is recursive. We conclude by proposition A.2.4 (2) that $\{(\vec{x}, y) \mid f(\vec{x}) \downarrow y\}$ is r.e., since $f(\vec{x}) \downarrow y$ iff $f(\vec{x}) \downarrow y$ in t steps for some t.

Conversely, if $\{(\vec{x}, y) \mid f(\vec{x}) \downarrow y\}$ is r.e., let W' be a recursive set such that $f(\vec{x}) \downarrow y$ iff $(\vec{x}, y, t) \in W'$ for some t. Then f can be written as

$$\pi_1 \circ (\lambda \vec{x}.\mu z.(z = \langle y, t \rangle \text{ and } (\vec{x}, y, t) \in W'))$$

and thus is p.r.. □

Here is an example of a semi-decidable, non decidable predicate.

Proposition A.2.9 *(1) The set $K = \{x \mid x \in W_x\}$ is semi-decidable. (2) The set $\{x \mid x \notin W_x\}$ is not r.e. .*

PROOF. (1) We have $K = im(\lambda x.\phi_x(x)) = im(\psi_U \circ \langle id, id \rangle)$, thus K is r.e. by proposition A.2.4(4). (2) Suppose $\{x \mid x \notin W_x\} = dom(f)$ for some *PR* function. Let n be an index of f. We have: $\forall x\, (x \notin W_x \text{ iff } x \in W_n)$. We get a contradiction when taking $x = n$. □

Exercise A.2.10 *Show that $\{x | \phi_x \text{ is recursive}\}$ is not r.e.. Hint: Consider $g(x) = \phi_{f(x)}(x) + 1$, where f is a claimed enumeration of the recursive functions.*

A.3 Rice-Shapiro Theorem

We end up this memento with an important theorem, widely used in theoretical computer science. It gives evidence to the thesis: *computable implies continuous* (cf. theorem 1.3.1 and proposition 15.5.10). A partial function θ such that $dom(\theta)$ is finite is called finite. Clearly finite functions from ω to ω are computable. Partial functions may be ordered as follows:

$$f \leq g \text{ iff } \forall x\,(f(x)\downarrow y) \Rightarrow (g(x)\downarrow y)\ .$$

Theorem A.3.1 (Rice-Shapiro) *Let A be a subset of PR such that $A' = \{x|\phi_x \in A\}$ is r.e.. Then, for any partial recursive f, $f \in A$ iff there exists a finite function $\theta \leq f$ such that $\theta \in A$.*

PROOF. Let T be a Turing machine computing the partial characteristic function of $K = \{x \mid x \in W_x\}$.

($\Leftarrow$) Suppose $f \in A$, and $\forall \theta \leq f\,(\theta \notin A)$. Let g be the partial recursive function defined by $g(z,t)\downarrow y$ iff T starting with z does not terminate in less than t steps, and $f(t)\downarrow y$. One has, by definition of g:

$$\lambda t.g(z,t) = \begin{cases} f & \text{if } z \notin K \\ \theta & \text{if } z \in K, \text{ where } \theta \leq_{fin} f\ . \end{cases}$$

Thus our assumption entails $z \notin K$ iff $\lambda t.g(z,t) \in A$. Let s be a recursive function, given by theorem A.1.5, such that $g(z,t) \cong \phi_{s(z)}(t)$. The above equivalence can be rephrased as: $z \notin K$ iff $s(z) \in A'$. But the predicate on the right is r.e.: contradiction.

($\Rightarrow$) Suppose $f \notin A$ and $\theta \in A$, for some finite $\theta \leq f$. We argue as in the previous case, defining now g by

$$g(z,t)\downarrow y \text{ iff } (\theta(t)\downarrow \text{ or } z \in K) \text{ and } f(t)\downarrow y\ .$$

□

Corollary A.3.2 (Rice) *If $B \subseteq PR$, $B \neq \emptyset$ and $B \neq PR$, then $\{x \mid \phi_x \in B\}$ is undecidable.*

PROOF. Let A be as in the statement of Rice-Shapiro theorem, and let $\perp$ be the totally undefined function. If $\perp \in A$, then, by the theorem, A must be the whole of PR.

Now suppose that $\{x \mid \phi_x \in B\}$ is decidable. Then B and B^c both satisfy the conditions of the Rice-Shapiro theorem. Consider the totally undefined function $\perp$. We have: $\perp \in B$ or $\perp \in B^c$. We deduce that either $B = PR$ or $B^c = PR$: contradiction. □

Appendix B

Memento of Category Theory

Category theory has been tightly connected to abstract mathematics since the first paper on cohomology by Eilenberg and Mac Lane [EM45] which establishes its basic notions. This appendix is a pro-memoria for a few elementary definitions and results in this branch of mathematics. We refer to [ML71, AL91] for adequate introductions and wider perspectives.

In the mathematical practice, category theory is helpful in formalizing a problem, as it is a good habit to ask in which category we are working in, if a certain transformation is a functor, if a given subcategory is reflective,... Using category theoretical terminology, one can often express a result in a more modular and abstract way. A list of "prescriptions" for the use of category theory in computer science can be found in [Gog91].

Categorical logic is a branch of category theory that arises from the observation due to Lawvere that logical connectives can be suitably expressed by means of universal properties. In this way one represents the models of, say, intuitionistic propositional logic, as categories with certain closure properties where sentences are interpreted as objects and proofs as morphisms (cf. chapter 4).

The tools developed in categorical logic begin to play a central role in the study of programming languages. A link between these two apparently distant topics is suggested by:

- The role of (typed) λ-calculi in the work of Landin, McCarthy, Strachey, and Scott on the foundations of programming languages.
- The Curry-Howard correspondence between systems of natural deduction and typed λ-calculi.
- The categorical semantics of typed λ-calculi along the lines traced by Lambek and Scott.

The basic idea in this study is to describe in the categorical language the "models" of a given programming languages. For instance, in the case of the simply typed λ-calculus the models correspond to the cartesian closed categories (cf. chapter 4).

This approach has been fairly successful in describing data types by means of universal properties. At present it is unclear if such program will be successful on a larger variety of programming languages features. It is however a recognized fact that ideas from categorical logic play a central role in the study of *functional* languages. Moreover promising attempts to describe categorically other features of programming languages such as modules, continuations, local variables,... are actively pursued.

B.1 Basic Definitions

A category may be regarded as a directed labelled graph endowed with a partial operation of composition of edges which is associative and has an identity.

Definition B.1.1 (category) *A category* $\mathbf{C}$ *is a sextuple* $(Ob, Mor, dom, cod, id, comp)$ *where* Ob *is the class of objects,* Mor *is the class of morphisms and:*

$$\begin{array}{ll} dom : Mor \to Ob & cod : Mor \to Ob \\ id : Ob \to Mor & comp : Comp \to Mor \end{array}$$

where $Comp = \{(f, g) \in Mor \times Mor \mid dom(f) = cod(g)\}$. *Moreover:*

$$\begin{array}{ll} id \circ f = f \circ id = f & \textit{(identity)} \\ f \circ (g \circ h) = (f \circ g) \circ h & \textit{(associativity)} \end{array}$$

where $f \circ g$ *is a shorthand for* $comp(f, g)$, *we write* $f \circ g$ *only if* $(f, g) \in Comp$, *and we omit to write the object to which* id *is applied in (identity).*

Let $\mathbf{C}$ be a category, $a, b \in Ob$, then

$$\mathbf{C}[a, b] = \{f \in Mor \mid dom(f) = a \text{ and } cod(f) = b\}$$

is the homset from a to b. We also write $f : a \to b$ for $f \in \mathbf{C}[a, b]$, and $a \in \mathbf{C}$ for $a \in Ob$. When confusion may arise we decorate the components $Ob, Mor, \ldots$ of a category with its name, hence writing $Ob_{\mathbf{C}}, Mor_{\mathbf{C}}, \ldots$ A category $\mathbf{C}$ is *small* if $Mor_{\mathbf{C}}$ is a set, and it is *locally* small if for any $a, b \in \mathbf{C}$, $\mathbf{C}[a, b]$ is a set.

Example B.1.2 (basic categories) *We just specify objects and morphisms. The operation of composition is naturally defined. The verification of the identity and associativity laws is immediate:*

- *Sets and functions,* **Set**.
- *Sets and partial functions,* **pSet**.
- *Sets and binary relations.*
- *Every pre-order* $(P, \leq)$ *induces a category* $\mathbf{P}$ *with* $\sharp\mathbf{P}[a, b] = 1$ *if* $a \leq b$ *and* $\sharp\mathbf{P}[a, b] = 0$ *otherwise.*

- *Any set with just an identity morphism for each object (this is the* discrete *category).*
- *Every monoid induces a category with one object and its elements as morphisms.*
- *Posets (or pre-orders) and monotonic functions.*
- *Groups and homomorphisms.*
- *Topological spaces and continuous functions,* **Top**.
- *Directed unlabelled graphs and transformations that preserve domain and codomain of edges.*

Definition B.1.3 (dual category) *Let* $\mathbf{C}$ *be a category. We define the dual category* $\mathbf{C}^{op}$ *as follows:*

$$\begin{array}{ll} Ob_{\mathbf{C}^{op}} = Ob_{\mathbf{C}} & \mathbf{C}^{op}[a,b] = \mathbf{C}[b,a] \\ id^{op} = id & f \circ^{op} g = g \circ f \ . \end{array}$$

Remark B.1.4 (dual property) *Given a property* P *for a category* $\mathbf{C}$ *and relative theorems it often makes sense to consider a* dual property P^{op} *to which correspond* dual theorems. *This idea can be formalized using the notion of dual category as follows: given a property* P *for a category* $\mathbf{C}$ *we say that* $\mathbf{C}$ *has property* P^{op} *if* $\mathbf{C}^{op}$ *has property* P.

Example B.1.5 (categories built out of categories) *(1) A* subcategory *is any sub-graph of a given category closed under composition and identity.*

(2) *If* $\mathbf{C}$ *and* $\mathbf{D}$ *are categories the* product category $\mathbf{C} \times \mathbf{D}$ *is defined by:*

$$Ob_{\mathbf{C}\times\mathbf{D}} = Ob_{\mathbf{C}} \times Ob_{\mathbf{D}}, \quad (\mathbf{C} \times \mathbf{D})[(a,b),(a',b')] = \mathbf{C}[a,a'] \times \mathbf{D}[b,b'] \ .$$

(3) *If* $\mathbf{C}$ *is a category and* $a \in \mathbf{C}$, *the* slice category $\mathbf{C} \downarrow a$ *is defined as:* $\mathbf{C} \downarrow a = \bigcup_{b \in \mathbf{C}} \mathbf{C}[b,a]$, $(\mathbf{C} \downarrow a)[f,g] = \{h \mid g \circ h = f\}$.

Definition B.1.6 (terminal object) *An object* a *in a category* $\mathbf{C}$ *is* terminal *if* $\forall b \in \mathbf{C}\, \exists! f : b \to a$. *We denote a terminal object with* 1 *and with* $!_b$ *the unique morphism from* b *to* 1.

Definition B.1.7 (properties of morphisms) *Let* $\mathbf{C}$ *be a category.*

- *A morphism* $f : a \to b$ *is a* mono *if* $\forall h,k (f \circ h = f \circ k \Rightarrow h = k)$.
- *A morphism* f *is* epi *if it is mono in* $\mathbf{C}^{op}$, *i.e.* $\forall h,k (h \circ f = k \circ f \Rightarrow h = k)$.
- *A morphism* $f : a \to b$ *is a* split *mono if there is a morphism* $g : b \to a$ *(called split epi) such that* $g \circ f = id$.
- *A morphism* $f : a \to b$ *is an* iso *if there is an inverse morphism* $g : b \to a$ *such that* $g \circ f = id$ *and* $f \circ g = id$. *We write* $a \cong b$ *if there is an iso between* a *and* b.

Exercise B.1.8 *Prove the following properties: (1) Each object has a unique identity morphism, (2) The inverse of an iso is unique, (3) If* $g \circ f = id$ *then* g *is an epi and* f *is a mono, (4)* f *epi and split mono implies* f *iso, (5)* f *mono and epi does not imply* f *iso, (6) The terminal object is uniquely determined up to isomorphism.*

B.2 Limits

The notions of cone and limit of a diagram are presented. The main result explains how to build limits of arbitrary diagrams by combining limits of special diagrams, namely *products* and *equalizers*.

Definition B.2.1 (diagram) *Let* $\mathbf{C}$ *be a category and* $I = (Ob_I, Mor_I)$ *a graph. A diagram in* $\mathbf{C}$ *over* I *is a graph morphism* $D : I \to \mathbf{C}$.

We often represent a diagram D as a pair $(\{d_i\}_{i \in Ob_I}, \{f_u\}_{u \in Mor_I})$.

Definition B.2.2 (category of cones) *Let* $\mathbf{C}$ *be a category and* $D : I \to \mathbf{C}$ *be a diagram. We define the category of cones* $Cones_{\mathbf{C}}D$ *as follows:*

$$Cones_{\mathbf{C}}D = \{(c, \{h_i\}_{i \in Ob_I}) \mid \forall u \in Mor_I\, (f_u : d_i \to d_j \Rightarrow h_j = f_u \circ h_i)\}$$
$$Cones_{\mathbf{C}}D[(c, \{h_i\}_{i \in Ob_I}), (d, \{k_i\}_{i \in Ob_I})] = \{g : c \to d \mid \forall i \in Ob_I\, (h_i = k_i \circ g)\}\ .$$

Definition B.2.3 (limit) *Let* $\mathbf{C}$ *be a category and* $D : I \to \mathbf{C}$ *be a diagram.* D *has a limit if the category* $Cones_{\mathbf{C}}D$ *has a terminal object.*

By the properties of terminal objects it follows that limits are determined up to isomorphism in $Cones_{\mathbf{C}}D$. Hence we may improperly speak of a limit as an object of the category $Cones_{\mathbf{C}}D$. We denote this object by $lim_{\mathbf{C}}D$. Also we say that the category $\mathbf{C}$ has *I-limits* if all diagrams indexed over I have limits.

Example B.2.4 (special limits) *We specialize the definition of limit to some recurring diagrams:*

- *If* $I = \emptyset$ *then the limit is a terminal object.*
- *If* I *is a discrete graph (no morphisms) then a diagram over* I *in* $\mathbf{C}$ *is just a family of objects* $\{a_i\}_{i \in Ob_I}$. *In this case a limit is also called a* product *and it is determined by a couple* $(c, \{\pi_i : c \to a_i\}_{i \in Ob_I})$ *such that for any cone* $(d, \{f_i : c \to a_i\}_{i \in Ob_I})$ *there exists a unique* $z : d \to c$ *such that* $\forall i \in Ob_I\, (f_i = \pi_i \circ z)$. *We write* c *as* $\Pi_{i \in Ob_I} a_i$, *and* z *as* $\langle f_i \rangle$, *which is an abbreviation for* $\langle f_i \rangle_{i \in Ob_I}$.
- Equalizers *are limits of diagrams over a graph* I *with two nodes, say* x, y, *and two edges from* x *to* y. *If the image of the diagram is a pair of morphisms* $f, g : a \to b$ *then an equalizer (or limit) is a pair* $(c, e : c \to a)$ *with properties (i)* $f \circ e = g \circ e$, *and (ii) if* $(c', e' : c' \to a)$ *and* $f \circ e' = g \circ e'$ *then* $\exists! z : c' \to c\, (e \circ z = e')$.
- Pullbacks *are limits of diagrams over a graph* I *with three nodes* x, y, z, *one edge from* x *to* z, *and one edge from* y *to* z. *If the image of the diagram is a pair of morphisms* $f : a \to d$, $g : b \to d$ *then a pullback (or limit) is a pair* $(c, \{h : c \to a, k : c \to b\})$ *with properties (i)* $f \circ h = g \circ k$, *and (ii) if* $(c', \{h' : c' \to a, k' : c' \to b\})$ *and* $f \circ h' = g \circ k'$ *then* $\exists! z : c' \to c\, (h \circ z = h'$ *and* $k \circ z = k')$.

The notions of *cocone*, *initial object*, and *colimit* are dual to the notions of cone, terminal object, and limit, respectively. We spell out the definition of coproduct which is often needed.

Definition B.2.5 (coproduct) *The colimit of a family of objects $\{a_i\}_{i\in Ob_I}$ is called coproduct. It is determined by a couple $(c, \{inj_i : a_i \to c\}_{i\in Ob_I})$ such that for any cocone $(d, \{f_i : a_i \to d\}_{i\in Ob_I})$ there exists a unique $z : c \to d$ such that $\forall i \in Ob_I\,(f_i = z \circ inj_i)$. We write c as $\Sigma_{i\in Ob_I} a_i$, and z as $[f_i]_{i\in Ob_I}$, or simply $[f_i]$.*

Exercise B.2.6 *Show that a category with terminal object and pullbacks has binary products and equalizers.*

Theorem B.2.7 (existence of I-limits) *Let* **C** *be a category and I be a graph, then* **C** *has I-limits if (1)* **C** *has equalizers, (2)* **C** *has all products indexed over Ob_I and Mor_I. In particular a category with equalizers and finite products has all finite limits.*

PROOF. Let $D : I \to \mathbf{C}$ be a diagram. We define:

$$P = \Pi_{i\in Ob_I} D(i) \qquad Q = \Pi_{u\in Mor_I} cod(D(u)) \ .$$

Next we define $f, g : P \to Q$, and $e : L \to P$ as follows:

- Let p and q denote the projections of P and Q, respectively.
- f is the unique morphism such that $D(u) \circ p_{dom(u)} = q_u \circ f$, for any $u \in Mor_I$.
- g is the unique morphism such that $p_{cod(u)} = q_u \circ g$, for any $u \in Mor_I$.
- e is the equalizer of f and g.

We claim that $(L, \{p_i \circ e\}_{i\in Ob_I})$ is a limit of the diagram D. The proof of this fact takes several steps.

(1) $(L, \{p_i \circ e\}_{i\in Ob_I}) \in Cones_{\mathbf{C}} D$. We have to show $D(u) \circ p_{dom(u)} \circ e = p_{cod(u)} \circ e$, for any $u \in Mor_I$. We observe:

$$\begin{aligned} D(u) \circ p_{dom(u)} \circ e &= q_u \circ f \circ e \\ &= q_u \circ g \circ e \\ &= p_{cod(u)} \circ e \ . \end{aligned}$$

(2) Let $(F, \{l_i\}_{i\in Ob_I}) \in Cones_{\mathbf{C}} D$. Then there is a uniquely determined morphism $\langle l_i \rangle : F \to P$ such that $p_i \circ \langle l_i \rangle = l_i$, for any $i \in Ob_I$. We claim $f \circ \langle l_i \rangle = g \circ \langle l_i \rangle$. This follows from the observation that for any $u \in Mor_I$:

$$\begin{aligned} q_u \circ f \circ \langle l_i \rangle &= D(u) \circ p_{dom(u)} \circ \langle l_i \rangle \\ &= D(u) \circ l_{dom(u)} \\ &= l_{cod(u)} \\ &= p_{cod(u)} \circ \langle l_i \rangle \\ &= q_u \circ g \circ \langle l_i \rangle \ . \end{aligned}$$

(3) Hence there is a unique $z : F \to L$ such that $e \circ z = \langle l_i \rangle$. We verify that $z : (F, \{l_i\}_{i \in Ob_I}) \to (L, \{p_i \circ e\}_{i \in Ob_I})$ is in $Cones_{\mathbf{C}} D$ by checking $p_i \circ e \circ z = l_i$, for any $i \in Ob_I$. This follows by:

$$p_i \circ e \circ z = p_i \circ \langle l_i \rangle = l_i \ .$$

(4) Finally suppose $z' : (F, \{l_i\}_{i \in Ob_I}) \to (L, \{p_i \circ e\}_{i \in Ob_I})$ in $Cones_{\mathbf{C}} D$. Then $z' : (F, \langle l_i \rangle) \to (L, e)$ as $p_i \circ e \circ z' = l_i$, for any $i \in Ob_I$ implies $e \circ z' = \langle l_i \rangle$. Hence $z = z'$. □

Exercise B.2.8 *Study the existence of (co-)limits in the categories introduced in example B.1.2.*

B.3 Functors and Natural Transformations

A functor is a morphism between categories and a natural transformation is a morphism between functors. The main result presented here is that there is a full and faithful functor from any category $\mathbf{C}$ to the category of set-valued functors over $\mathbf{C}^{op}$.

Definition B.3.1 (functor) *Let* $\mathbf{C}$, $\mathbf{D}$ *be categories, a functor* $F : \mathbf{C} \to \mathbf{D}$ *is a morphism between the underlying graphs that preserves identity and composition, that is:*

$$F_{Ob} : Ob_{\mathbf{C}} \to Ob_{\mathbf{D}} \qquad F_{Mor} : Mor_{\mathbf{C}} \to Mor_{\mathbf{D}}$$
$$F_{Mor}(id_a) = id_{F_{Ob}(a)} \quad F_{Mor}(f \circ g) = F_{Mor}(f) \circ F_{Mor}(g)$$

where if $f : a \to b$ *then* $F_{Mor}(f) : F_{Ob}(a) \to F_{Ob}(b)$.

In the following we omit the indices Ob and Mor in a functor. By a *contravariant* functor $F : \mathbf{C} \to \mathbf{D}$ we mean a functor $F : \mathbf{C}^{op} \to \mathbf{D}$.

Exercise B.3.2 *Show that small categories and functors form a category.*

Definition B.3.3 (hom-functor) *Let* $\mathbf{C}$ *be a locally small category. We define the hom-functor* $\mathbf{C}[\_,\_] : \mathbf{C}^{op} \times \mathbf{C} \to \mathbf{Set}$ *as follows:*

$$\mathbf{C}[\_,\_](a,b) = \mathbf{C}[a,b] \quad \mathbf{C}[\_,\_](f,g) = \lambda h.g \circ h \circ f \ .$$

Given an object c in the category $\mathbf{C}$ we denote with $\mathbf{C}[\_,c] : \mathbf{C}^{op} \to \mathbf{Set}$ and $\mathbf{C}[c,\_] : \mathbf{C} \to \mathbf{Set}$ the contravariant and covariant functors over $\mathbf{C}$ obtained by restricting the hom-functor to the first and second component, respectively.

Exercise B.3.4 *Suppose* $F : \mathbf{C} \to \mathbf{D}$ *is a functor,* $D : I \to \mathbf{C}$ *is a diagram and* $(a, \{l_i\}_{i \in Ob_I}) \in Cones_{\mathbf{C}} D$. *Show that* $(Fa, \{Fl_i\}_{i \in Ob_I}) \in Cones_{\mathbf{D}}(F \circ D)$.

Definition B.3.5 (limit preservation) *Suppose $F : \mathbf{C} \to \mathbf{D}$ is a functor, and $D : I \to \mathbf{C}$ is a diagram. We say that F preserves the limits of the diagram D if:*

$$(a, \{l_i\}_{i \in Ob_I}) \in lim_{\mathbf{C}} D \;\Rightarrow\; (Fa, \{Fl_i\}_{i \in Ob_I}) \in lim_{\mathbf{D}}(F \circ D) \; .$$

Proposition B.3.6 *Let $\mathbf{C}$ be a locally small category and c be an object in $\mathbf{C}$. Then the covariant hom-functor $\mathbf{C}[c, \_] : \mathbf{C} \to \mathbf{Set}$ preserves limits.*

PROOF. Let $D = (\{d_i\}_{i \in Ob_I}, \{f_u\}_{u \in Mor_I})$ be a diagram and $(a, \{l_i\}_{i \in Ob_I}) \in lim_{\mathbf{C}} D$. Then $(\mathbf{C}[c, a], \{\lambda h.l_i \circ h\}_{i \in Ob_I}) \in Cones_{\mathbf{Set}}(\mathbf{C}[c, \_] \circ D)$. We suppose $(X, \{g_i\}_{i \in Ob_I}) \in Cones_{\mathbf{Set}}(\mathbf{C}[c, \_] \circ D)$, that is:

$$\forall u \in Mor_I \, \forall x \in X \, (f_u : d_i \to d_j \;\Rightarrow\; (f_u \circ g_i(x) = g_j(x))) \; .$$

Then $\forall x \in X \, (c, \{g_i(x)\}_{i \in Ob_I}) \in Cones_{\mathbf{Set}} D$. Hence there is a unique $h(x) : c \to a$ such that $g_i(x) = l_i \circ h(x)$, for any $i \in Ob_I$. We can then build a unique $z : X \to C[c, a]$ such that $l_i \circ z = g_i$, for any $i \in Ob_I$. This z is defined by $z(x) = h(x)$. □

Definition B.3.7 (natural transformation) *Let $F, G : \mathbf{C} \to \mathbf{D}$ be functors. A natural transformation $\tau : F \to G$ is a family $\{\tau_a : Fa \to Ga\}_{a \in Ob_{\mathbf{C}}}$ such that for any $f : a \to b$, $\tau_b \circ Ff = Gf \circ \tau_a$. A natural isomorphism τ is a natural transformation such that τ_a is an isomorphism, for any a.*

Exercise B.3.8 *Given $\mathbf{C}, \mathbf{D}$ categories show that the functors from $\mathbf{C}$ to $\mathbf{D}$, and their natural transformations form a category. We denote this new category with $\mathbf{D}^{\mathbf{C}}$. It can be shown that $\mathbf{D}^{\mathbf{C}}$ is actually an* exponent *in the sense of cartesian closed categories (see section B.7).*

Definition B.3.9 (category of pre-sheaves) *Given a category $\mathbf{C}$ the category of* pre-sheaves *over $\mathbf{C}$ is the category $\mathbf{Set}^{\mathbf{C}^{op}}$ of contravariant set-valued functors and natural transformations.*

Another important operation involving natural transformations is the composition with a functor.

Proposition B.3.10 *If $G : \mathbf{B} \to \mathbf{C}$, $F, F' : \mathbf{C} \to \mathbf{C}'$ and $\delta : F \to F'$, then $\delta G : F \circ G \to F' \circ G$ is natural, where δG is defined by set theoretical composition, i.e. $(\delta G)_a = \delta_{Ga}$. Likewise, if $H : \mathbf{C}' \to \mathbf{B}'$, $F, F' : \mathbf{C} \to \mathbf{C}'$ and $\delta : F \to F'$, then $H\delta : H \circ F \to H \circ F'$ is natural, where $(H\delta)_a = H(\delta_a)$.*

The composition of natural transformations and functors extends to a notion of *horizontal* composition of natural transformations (in contrast to the *vertical* one given by the "∘").

Proposition B.3.11 *If* $F, F' : \mathbf{C} \to \mathbf{C}'$, $G, G' : \mathbf{C}' \to \mathbf{C}''$, $\delta : F \to F'$, $\epsilon : G \to G'$, *then*

$$(\epsilon F') \circ (G\delta) = (G'\delta) \circ (\epsilon F) : G \circ F \to G' \circ F' \ .$$

We write $\epsilon\delta$ *for the common value of both sides of this equation.*

Exercise B.3.12 *Show the following so called* interchange law *(originally stated by Godement), for all* $\delta, \delta', \epsilon, \epsilon'$ *of appropriate types* $(\epsilon' \circ \epsilon)(\delta' \circ \delta) = (\epsilon'\delta') \circ (\epsilon\delta)$.

Definition B.3.13 (full and faithful functor) *A functor* $F : \mathbf{C} \to \mathbf{D}$ *is* full *if* $\forall a, b \forall h : Fa \to Fb \exists f : a \to b (Ff = h)$, *and* faithful *if it is injective on each hom-set* $\mathbf{C}[a, b]$.

Theorem B.3.14 (Yoneda) *For any category* $\mathbf{C}$ *there is a full and faithful functor* $Y : \mathbf{C} \to \mathbf{Set}^{\mathbf{C}^{op}}$ *from* $\mathbf{C}$ *into the related category of pre-sheaves, called Yoneda embedding, and defined as follows:*

$$Y(c) = \mathbf{C}[\_, c] \qquad Y(f) = \lambda h.f \circ h \ .$$

PROOF HINT. The key to the proof that Y is full resides in the following lemma where we take F as $h_d = \mathbf{C}[\_, d]$. □

Lemma B.3.15 (Yoneda's lemma) *For any functor* $F : \mathbf{C}^{op} \to \mathbf{Set}$ *and any object* $c \in \mathbf{C}$, *the following isomorphism holds in* $\mathbf{Set}$, *where* $h_c = \mathbf{C}[\_, c]$ *and* $\mathbf{Nat}[h_c, F]$ *are the natural transformations from* h_c *to* F*:*

$$Fc \cong \mathbf{Nat}[h_c, F] \ .$$

PROOF. We define $i : Fc \to \mathbf{Nat}[h_c, F]$ with inverse $j : \mathbf{Nat}[h_c, F] \to Fc$ as follows:

$$i(x) = \lambda d.\lambda l : d \to c.(Fl)(x) \qquad j(\tau) = \tau_c(id_c) \ .$$

First we verify that $i(x)$ is a natural transformation as:

$$(Ff)((i(x)_a)(l)) = (Ff)((Fl)(x)) = F(l \circ f)(x) = (i(x)_b)(l \circ f) \ .$$

Next we verify that j is the inverse of i:

$$\begin{aligned} j(i(x)) &= i(x)_c(id_c) = (Fid_c)(x) = (id)(x) = x \\ i(j(\tau)) &= \lambda d.\lambda l : d \to c.(Fl)(\tau_c(id_c)) = \lambda d.\lambda l : d \to c.\tau_d(l) = \tau \end{aligned}$$

as by applying the naturality of τ to $l : d \to c$ one gets $(Fl)(\tau_c(id_c)) = \tau_d(l)$. □

B.4 Universal Morphisms and Adjunctions

A universal morphism is a rather simple abstraction of a frequent mathematical phenomenon.

Example B.4.1 *(1) Given a signature Σ consider the category of Σ-algebras and morphisms. If A is a Σ-algebra denote with $|A|$ its carrier (which is a set). There is a well known construction which associates to any set X the* free Σ-*algebra $\Sigma(X)$ and which is characterized by the following property:*

$$\exists u : X \to |\Sigma(X)| \, \forall A \, \forall f : X \to |A| \, \exists! f' : \Sigma(X) \to A \, (f' \circ u = f) \; .$$

(2) Consider the category of metric spaces and continuous morphisms and the full subcategory of complete metric spaces. The Cauchy completion associates to any metric space (X, d) a complete metric space (X_c, d_c) which is characterized by:

$$\exists u : (X, d) \to (X_c, d_c) \, \forall (Y, d') \text{ complete } \forall f : (X, d) \to (Y, d')$$
$$\exists! f' : (X_c, d_c) \to (Y, d') \, (f' \circ u = f) \; .$$

Definition B.4.2 (universal morphism) *Let $F : \mathbf{C} \to \mathbf{D}$ be a functor and d an object in $\mathbf{D}$. Then the couple $(c_d, u : d \to Fc_d)$ is* universal *from d to F (and we also write $(c_d, u) : d \to F$) if:*

$$\forall c \, \forall f : d \to Fc \, \exists! f' : c_d \to c \, (Ff' \circ u = f) \; .$$

Exercise B.4.3 *(1) Show that if $(c_d, u) : d \to F$ and $(c'_d, u') : d \to F$ then $c_d \cong c_{d'}$. (2) Explicit the dual notion of co-universal. (3) Verify that the previous examples B.4.1 fit the definition of universal morphism.*

The notion of *adjunction* is a fundamental one, and it has several equivalent characterizations. In particular, an adjunction arises whenever there is a "uniform" way of determining a universal morphism (cf. proposition B.4.6(3) and theorem B.4.8).

Definition B.4.4 (adjunction) *An* adjunction *between two categories $\mathbf{C}, \mathbf{D}$ is a triple (L, R, τ), where $L : \mathbf{D} \to \mathbf{C}$, and $R : \mathbf{C} \to \mathbf{D}$ are functors and $\tau : \mathbf{C}[L\_, \_] \to \mathbf{D}[\_, R\_]$ is a natural isomorphism. We say that L is the* left adjoint*, R is the* right adjoint*, and we denote this situation by $L \dashv R$.*

Exercise B.4.5 *With reference to example B.4.1, define the "free algebra" and "Cauchy completion" functors. Verify that they are left adjoints to the respective forgetful functors.*

In the following we develop some properties of adjunctions in the special case in which **C** and **D** are poset categories and therefore the functors L and R are monotonic functions. Let us first observe that the triple (L, R, τ) is an adjunction iff

$$\forall c, d (Ld \leq c \text{ iff } d \leq Rc) .$$

A pair of monotonic functions satisfying this property is also known as Galois connection.

Proposition B.4.6 *Let* **C**, **D** *be poset categories. Then:*

(1) *Every component of an adjunction determines the other.*

(2) *The following conditions are equivalent for* $R : \mathbf{C} \to \mathbf{D}$, *and* $L : \mathbf{D} \to \mathbf{C}$: *(a)* $\forall c, d (Ld \leq c \text{ iff } d \leq Rc)$, *and (b)* $L \circ R \leq id_C$, $id_D \leq R \circ L$.

(3) *The pair* $(c_d, d \leq Fc_d)$ *is universal from* d *to* $F : \mathbf{C} \to \mathbf{D}$ *if:*

$$\forall c (d \leq Fc \Rightarrow c_d \leq c) . \tag{B.1}$$

If $\forall d (c_d, d \leq Fc_d) : d \to F$ *then* F *has a left adjoint* L *where* $L(d) = c_d$.

(4) *Vice versa, if* $L \dashv R$, $R : \mathbf{C} \to \mathbf{D}$, *and* $L : \mathbf{D} \to \mathbf{C}$ *then* $\forall d (Ld, d \leq (R \circ L)d)$ *is universal from* d *to* R, *and symmetrically* $\forall c (Rc, (L \circ R)c \leq c)$ *is co-universal from* L *to* c.

PROOF. (1) We note that if $L \dashv R$ and $L \dashv R'$ then $d \leq Rc$ iff $Ld \leq c$ iff $d \leq R'c$. For $d = Rc$ we get $Rc \leq Rc$ iff $Rc \leq R'c$. Hence $Rc \leq R'c$, and symmetrically $R'c \leq Rc$.

(2) Concerning the equivalence of the statements: $(a) \Rightarrow (b)$ $L(Rc) \leq c$ iff $Rc \leq Rc$. $(b) \Rightarrow (a)$ $Ld \leq c$ implies $d \leq R(Ld) \leq Rc$, and $d \leq Rc$ implies $Ld \leq L(Rc) \leq c$.

(3) Condition B.1 follows from definition B.4.2. If $d \leq Fc_d$ then, by condition B.1, $c_d \leq c$, that is $Ld \leq c$. By hypothesis $\forall d (d \leq F(Ld))$. Hence, $Ld \leq c$ implies $d \leq F(Ld) \leq Fc$.

(4) Direct application of the characterizations (2-3). □

Exercise B.4.7 *Generalize point (1) of proposition B.4.6 to arbitrary categories: if* $L \dashv R$ *and* $L \dashv R'$ *then* R *and* R' *are naturally isomorphic.*

The following theorem connects adjunctions with universal morphisms and generalizes points (2-4) of proposition B.4.6.

Theorem B.4.8 *An adjunction* (L, R, τ) *determines:*

(1) *A natural transformation* $\eta : id_D \to R \circ L$, *called unit, such that for each object* $d \in \mathbf{D}$, (Ld, η_d) *is universal from* d *to* R, *for each* $f : Ld \to c$, $\tau(f) = R(f) \circ \eta_d : d \to Rc$.

(2) *A natural transformation* $\epsilon : L \circ R \to id_C$, *called counit, such that for each object* $c \in \mathbf{C}$, (Rc, ϵ_c) *is co-universal from* L *to* c, *and for each* $g : d \to Rc$, $\tau^{-1}(g) = \epsilon_c \circ L(g) : Ld \to c$.

(3) *Moreover the following equations hold:*

$$(R\epsilon) \circ (\eta R) = id_R \quad (\epsilon L) \circ (L\eta) = id_L \ .$$

Exercise B.4.9 *Show that an adjunction* $L \dashv R$ *is completely determined by (i) functors* $L : \mathbf{D} \to \mathbf{C}$ *and* $R : \mathbf{C} \to \mathbf{D}$, *(ii) natural transformations* $\epsilon : L \circ R \to id$, $\eta : id \to R \circ L$ *such that* $(\epsilon L) \circ (L\eta) = id_L$ *and* $(R\epsilon) \circ (\eta R) = id_R$.

Exercise B.4.10 *Let* $\mathbf{C}$ *be a category. Show:*

(1) $\mathbf{C}$ *has a terminal object iff the unique functor* $! : \mathbf{C} \to \mathbf{1}$ *has a right adjoint.*

(2) $\mathbf{C}$ *has a binary products iff the diagonal functor* $\Delta : \mathbf{C} \to \mathbf{C} \times \mathbf{C}$ *has a right adjoint, where* $\Delta(a) = (a, a)$ *and* $\Delta(f) = (f, f)$.

(3) *Given a graph* I *consider the category* $[\mathbf{I} \to \mathbf{C}]$ *of graphs and natural transformations (observe that the definition of natural transformation does not require* I *to be a category). Define a generalized diagonal functor* $\Delta_I : \mathbf{C} \to [\mathbf{I} \to \mathbf{C}]$ *and show that* $\mathbf{C}$ *has limits of* I*-indexed diagrams iff the functor* Δ_I *has a right adjoint.*

(4) *Show that the left adjoint of the inclusion functor from complete metric spaces to metric spaces builds the Cauchy completion. Analogously show that the left adjoint to the forgetful functor from* Σ*-algebras to* **Set** *builds the free-algebra.*

The definition of adjunction hides some redundancy, the following characterizations show different ways of optimizing it. An adjunction $L \dashv R$ is determined by (i) a functor $L : \mathbf{D} \to \mathbf{C}$, (ii) a function $R : Ob_{\mathbf{C}} \to Ob_{\mathbf{D}}$, and one of the following conditions:

(1) Bijections $\tau_{d,c} : \mathbf{D}[Ld, c] \to \mathbf{C}[d, Rc]$ for all c, d, such that for all f, g of appropriate types: $\tau(f) \circ g = \tau(f \circ L(g))$. Hint: R is uniquely extended to a functor by setting $Rh = \tau(h \circ \tau^{-1}(id))$.

(2) Functions $\tau_{d,c} : \mathbf{D}[Ld, c] \to \mathbf{C}[d, Rc]$ for all c, d, and morphisms $\epsilon_c : L(Rc) \to c$ (ϵ for short) for all c, such that for all f, g of appropriate types $\epsilon \circ L(\tau(f)) = f$, $g = \tau(\epsilon \circ Lg)$. Hint: τ is proved bijective by setting $\tau^{-1}(g) = \epsilon \circ Lg$. The naturality is also a consequence.

(3) Morphisms $\epsilon_c : L(Rc) \to c$, for all c, such that for all $c, d, f \in \mathbf{C}[Ld, c]$, there exists a unique morphism, written $\tau(f)$, satisfying $\epsilon \circ L(\tau(f)) = f$. Hint: The naturality of the ϵ follows. Another way of saying this is that (Rc, ϵ_c) are co-universal from L to c.

B.5 Adjoints and Limits

Given a functor F, the existence of a left (right) adjoint implies the preservation of limits (colimits). First consider the situation in **Poset**.

Proposition B.5.1 *Let* $\mathbf{C}, \mathbf{D}$ *be poset categories. If there is an adjunction* $L \dashv R$, $R : \mathbf{C} \to \mathbf{D}$, *and* $L : \mathbf{D} \to \mathbf{C}$ *then* R *preserves glb's (and* L *lub's).*

PROOF. We suppose $X \subseteq \mathbf{C}$, and $\exists \bigwedge X$. Also we assume $\forall c \in X\, (d \leq Rc)$. Then $\forall c \in X\, (Ld \leq c)$. Hence $Ld \leq \bigwedge X$, that implies $d \leq R(\bigwedge X)$. □

The following theorem generalizes the previous proposition.

Theorem B.5.2 *If the functor* $R : \mathbf{C} \to \mathbf{D}$ *has a left adjoint then* R *preserves limits (and* L *colimits).*

Vice versa one may wonder if the existence of limits helps in the construction of an adjunction. Consider again the situation in **Poset**.

Proposition B.5.3 *Let* $\mathbf{C}, \mathbf{D}$ *be poset categories. Suppose there is* $R : \mathbf{C} \to \mathbf{D}$ *and* $\mathbf{C}$ *has all glb's. Then* R *has a left adjoint iff* R *preserves glb's.*

PROOF. ($\Rightarrow$) This follows by B.5.1. ($\Leftarrow$) Define $L(d) = \bigwedge_{\mathbf{C}}\{c' \mid d \leq Rc'\}$. Then $d \leq Rc$ implies $L(d) = \bigwedge_{\mathbf{C}}\{c' \mid d \leq Rc'\} \leq c$. On the other hand, if $L(d) \leq c$ then $d \leq \bigwedge_{\mathbf{C}}\{Rc' \mid d \leq Rc'\} = R(\bigwedge_{\mathbf{C}}\{c' \mid d \leq Rc'\}) = R(L(d)) \leq R(c)$. □

There are several results which generalize the previous proposition. We present just one of them. Given a functor $R : \mathbf{C} \to \mathbf{D}$, where $\mathbf{C}$ is small and has all limits the following *Solution Set Condition* is enough to establish the existence of a left adjoint:

$$\begin{array}{l} \forall d \in \mathbf{D}\, \exists \{(c_i, w_i : d \to Rc_i)\}_{i \in I} \quad (I \text{ set}), \\ \forall c' \in \mathbf{C}\, \forall f : d \to Rc' \exists i \in I\, \exists f' : c_i \to c' \\ f = Rf' \circ w_i\ . \end{array}$$

This can be understood as a weakening of the universal condition in the definition B.4.2 of a universal morphism. Given an object $d \in \mathbf{D}$ we can find a set of objects and morphisms that commute (not in a unique way) with every morphism $f : d \to Rc'$.

Exercise B.5.4 *Show that if* R *has a left adjoint then the solution set condition is always satisfied (cf. [BW85]).*

Theorem B.5.5 (Freyd) *Let* $\mathbf{C}$ *be a category with all limits. Then a functor* $R : \mathbf{C} \to \mathbf{D}$ *has a left adjoint iff* R *preserves all limits and satisfies the Solution Set Condition.*

B.6 Equivalences and Reflections

The notion of functor isomorphism is often too strong to express the idea that two categories enjoy the "same properties" (e.g. existence of limits). The following weaker notion of equivalence is more useful.

Definition B.6.1 (equivalence of categories) *A functor $F : \mathbf{C} \to \mathbf{D}$ is an equivalence of categories if there is a functor $G : \mathbf{D} \to \mathbf{C}$ such that $F \circ G \cong id_{\mathbf{D}}$, and $G \circ F \cong id_{\mathbf{C}}$, via natural isomorphisms.*

Theorem B.6.2 *The following properties of a functor $F : \mathbf{C} \to \mathbf{D}$ are equivalent:*

(1) *F is an equivalence of categories.*

(2) *F is part of an adjoint (F, G, η, ϵ) such that η and ϵ are natural isomorphisms.*

(3) *F is full and faithful and $\forall d \in \mathbf{D}\, \exists c \in \mathbf{C}\, (d \cong Fc)$.*

Exercise B.6.3 *Give examples of equivalent but not isomorphic pre-orders.*

Exercise B.6.4 *Show that any adjunction cuts down to an equivalence between the full subcategory whose objects are those at which the counity and the unity, respectively, are iso.*

Exercise B.6.5 *(1) Let $L \dashv R$ be an adjunction where $L : \mathbf{D} \to \mathbf{C}$, $R : \mathbf{C} \to \mathbf{D}$. If $\mathbf{C}', \mathbf{D}'$ are full subcategories of $\mathbf{C}, \mathbf{D}$, respectively, $\forall a \in \mathbf{D}'\, La \in \mathbf{C}'$, and $\forall b \in \mathbf{C}'\, Rb \in \mathbf{D}'$, then the adjunction $L \dashv R$ restricts to an adjunction between $\mathbf{C}'$ and $\mathbf{D}'$. The same holds of equivalences. (2) If $L \dashv R$ is an equivalence between two categories $\mathbf{C}, \mathbf{D}$, if $\mathbf{D}'$ is a full subcategory of $\mathbf{D}$ closed under isomorphic objects, then the equivalence cuts down to an equivalence between $\mathbf{C}'$ and $\mathbf{D}'$ where $\mathbf{C}'$ is the full subcategory of $\mathbf{C}$ whose collection of objects is $\{a \mid Ra \in \mathbf{D}'\}$, which is equal to $\{a \mid \exists b \in \mathbf{D}'\, a \cong Lb\}$.*

Exercise B.6.6 *Suppose that $L \dashv R$ is an adjunction with counity ϵ such that ϵ_d is a mono for all d. Show the following equivalences: (1) ϵ is iso at d iff d is isomorphic to Lc for some c. (2) η is iso at c iff c is isomorphic to Rd for some d . Show the same properties under the assumption that η_c is epi, for all c.*

Reflection is a condition weaker than equivalence. The following proposition illustrates the idea in the poset case.

Proposition B.6.7 (poset reflection) *Let $\mathbf{C}, \mathbf{D}$ be poset categories. Suppose there is an adjunction $L \dashv R$, $R : \mathbf{C} \to \mathbf{D}$, $L : \mathbf{D} \to \mathbf{C}$, where R is an inclusion. Then for any $X \subseteq \mathbf{C}$,*

$$\exists \bigwedge_{\mathbf{D}} X \quad \Rightarrow \quad \exists \bigwedge_{\mathbf{C}} X \text{ and } \bigwedge_{\mathbf{C}} X = \bigwedge_{\mathbf{D}} X \ .$$

PROOF. We set $c = L(\bigwedge_{\mathbf{D}} X)$, and show $c = \bigwedge_{\mathbf{C}} X = \bigwedge_{\mathbf{D}} X$. For any $x \in X$, $\bigwedge_{\mathbf{D}} X \leq x$ implies, by the adjunction hypothesis, $c = L(\bigwedge_{\mathbf{D}} X) \leq x$. Hence $c \leq \bigwedge_{\mathbf{D}} X$. On the other hand, suppose $c' \in \mathbf{C}$ is a lower bound for X, then $c' \leq \bigwedge_{\mathbf{D}} X$, and therefore $Lc' \leq c$. It is enough to observe that $Lc' = c'$. By the adjuction condition, $c' \leq c'$ implies $Lc' \leq c'$, and $Lc' \leq Lc'$ implies $c' \leq Lc'$. □

Definition B.6.8 *If* $\mathbf{C}$ *is a subcategory of* $\mathbf{D}$ *we denote with* $Incl : \mathbf{C} \to \mathbf{D}$ *the inclusion functor. We say that* $\mathbf{C}$ *is a* reflective subcategory *of* $\mathbf{D}$ *if there is* L *such that* $L \dashv Incl$. L *is also called the* reflector *functor.*

The point (4) of the following theorem generalizes the previous example.

Theorem B.6.9 *For an adjunction* (L, R, η, ϵ) *the following holds:*

(1) R *is faithful iff every component* $\epsilon_c : L(Rc) \to c$ *is an epi.*

(2) R *is full iff every component* $\epsilon_c : L(Rc) \to c$ *is a split mono (i.e. it has a left inverse).*

(3) *Hence* R *is full and faithful iff* $\epsilon_c : L(Rc) \to c$ *is an iso.*

(4) *If* $R : \mathbf{C} \to \mathbf{D}$ *is the inclusion functor then for any diagram* $D : I \to \mathbf{C}$*:*

$$\exists lim_{\mathbf{D}} D \quad \Rightarrow \quad \exists lim_{\mathbf{C}} D \text{ and } lim_{\mathbf{C}} D \cong lim_{\mathbf{D}} D \ .$$

Exercise B.6.10 *Show that the full sub-category of Hausdorff topological spaces is reflective in the category of topological spaces and continuous morphisms, and that the full subcategory of posets is reflective in the category of preorders and monotonic morphisms. On the other hand show that the ideal completion of a poset to a directed complete poset does* not *provide a left adjoint to the inclusion of directed complete posets into the category of posets and monotonic morphisms.*

B.7 Cartesian Closed Categories

Cartesian closure formalizes the idea of closure of a category under function space. Chapter 4 provides some intuition for the genesis of the notion, several equivalent definitions, e.g. 4.2.5, and examples. We recall that a CCC is a category with finite products and such that the functor $\_ \times A : \mathbf{C} \to \mathbf{C}$ has a right adjoint, for any object A. In the following, we present small categories and presheaves as examples of CCC's.

Example B.7.1 *The category of small categories and functors is cartesian closed. The exponent object* $\mathbf{D}^{\mathbf{C}}$ *is given by the category of functors and natural transformations. Then we define:*

$$\begin{array}{ll} ev(F, A) = FA & ev(\delta, f) = Gf \circ \delta_A = \delta_B \circ Ff \quad (\delta : F \to G, f : A \to B) \\ \Lambda(F)AB = F(A, B) & \Lambda(F)fg = F(f, g) \\ \Lambda(F)Af = F(id_A, f) & \Lambda(F)fB = F(f, id_B) \ . \end{array}$$

Example B.7.2 (presheaves) *Our next example of a CCC is* $\mathbf{Set}^{\mathbf{C}^{op}}$*, for any category* $\mathbf{C}$*. The cartesian structure is built* pointwise*, but this does not work for exponents (try to take* $(F \Rightarrow G)A = \mathbf{Set}[FA, GA]$*, how does one define* $(F \Rightarrow G)$ *on morphisms?). The solution is to use Yoneda lemma B.3.15. For* $F, G : \mathbf{C}^{op} \to \mathbf{Set}$ *we define:*

$$(F \Rightarrow G) = \lambda c.Nat[\mathbf{C}[\_, c] \times F, G] \ .$$

Exercise B.7.3 *If* $\mathbf{C}$ *is a preorder, we can recover a pointwise definition of* $F \Rightarrow G$*. Define* $\mathbf{C}_{\restriction A}$ *as the full subcategory of* $\mathbf{C}$ *with objects those* B *such that* $B \leq A$*. Given* $F : \mathbf{C}^{op} \to \mathbf{Set}$*, define* $F_{\restriction A} : (\mathbf{C}_{\restriction A})^{op} \to \mathbf{Set}$ *by restriction. Then show* $(F \Rightarrow G)A = \mathbf{Set}[F_{\restriction A}, G_{\restriction A}]$*.*

Exercise B.7.4 *Let* $\mathbf{C}$ *be a CCC which has an initial object* 0*. Then show that for any* A*: (i)* $0 \times A \cong 0$*, (ii)* $\mathbf{C}[A, 0] \neq \emptyset$ *implies* $A \cong 0$ *(thus* $\mathbf{C}[A, 0]$ *has at most one element). If furthermore* $\mathbf{C}$ *has finite limits, show that, for any* A*, the unique morphism from* 0 *to* A *is mono. Hints:* $\mathbf{C}[0 \times A, B] \cong \mathbf{C}[0, B^A]$ *and consider in particular* $B = 0 \times A$*. Consider also* $!^{op} \circ \pi_1$*. Suppose* $f : A \to 0$*. Then consider* $\pi_2 \circ \langle f, id \rangle$*.*

Exercise B.7.5 *Let* $\mathbf{C}$ *be a CCC, and* 0 *be an object such that the natural transformation* $\mu : \lambda x.x \to \lambda x.(x \Rightarrow 0) \Rightarrow 0$ *defined by* $\mu = \Lambda(ev \circ \langle \pi_2, \pi_1 \rangle)$ *is iso. Show that* 0 *is initial and that* $\mathbf{C}$ *is a preorder (this is an important negative fact: there is no nontrivial categorical semantics of classical logic, thinking of* 0 *as absurdity and of* $(x \Rightarrow 0) \Rightarrow 0$ *as double negation). Hints: (i)* $0 \Rightarrow 0 \cong 1$*, indeed* $1 \cong (1 \Rightarrow 0) \Rightarrow 0$*, and* $(1 \Rightarrow A) \cong A$*, for any* A*. (ii) for any* A*:*

$$\begin{array}{lll} \mathbf{C}[0, A] & \cong \mathbf{C}[0, (A \Rightarrow 0) \Rightarrow 0] & \cong \mathbf{C}[0 \times (A \Rightarrow 0), 0] \\ & \cong \mathbf{C}[A \Rightarrow 0, 0 \Rightarrow 0] & \cong \mathbf{C}[A \Rightarrow 0, 1] \end{array}$$

(iii) for any A, B*:* $\mathbf{C}[A, B] \cong \mathbf{C}[A, (B \Rightarrow 0) \Rightarrow 0] \cong \mathbf{C}[A \times (B \Rightarrow 0), 0]$*.*

B.8 Monads

The notion of monad (or triple) is an important category-theoretical notion, we refer to [BW85, ML71] for more information and to chapter 8 for several applications of this notion in computer science.

Definition B.8.1 (monad) *A monad over a category* $\mathbf{C}$ *is a triple* (T, η, μ) *where* $T : \mathbf{C} \to \mathbf{C}$ *is a functor,* $\eta : id_{\mathbf{C}} \to T$*,* $\mu : T^2 \to T$ *are natural transformations and the following equations hold:*

$$\mu_A \circ \eta_{TA} = id_{TA} \quad \mu_A \circ T\eta_A = id_{TA} \quad \mu_A \circ \mu_{TA} = \mu_A \circ T\mu_A \ .$$

Exercise B.8.2 *Show that if* $\mathbf{C}$ *is a poset then a monad can be characterized as a closure, i.e. a monotonic function* $T : \mathbf{C} \to \mathbf{C}$*, such that* $id \leq T = T \circ T$*.*

Definition B.8.3 (category of T-algebras) *Given a monad (T, η, μ) a T-algebra is a morphism $\alpha : Td \to d$ satisfying the following conditions:*

$$(T_\eta) \quad \alpha \circ \eta_d = id_d \qquad (T_\mu) \quad \alpha \circ T\alpha = \alpha \circ \mu_d \ .$$

The category $\mathbf{Alg}_T$ *has T-algebras as objects and*

$$\mathbf{Alg}_T[\alpha : Td \to d, \beta : Td' \to d'] = \{f : d \to d' \mid \beta \circ Tf = f \circ \alpha\} \ .$$

Exercise B.8.4 *With reference to example B.4.1 and exercise B.4.5, show that Σ-algebras are exactly the algebras for the monad associated with the adjunction $T_\Sigma \dashv$ Forget, where T_Σ is the "free algebra" functor and Forget is the forgetful functor (cf. exercise B.4.5).*

Exercise B.8.5 *Consider the powerset functor $\mathcal{P} : \mathbf{Set} \to \mathbf{Set}$ with $\eta(x) = \{x\}$ and $\mu(X) = \bigcup X$. (1) Show that these data define a monad. (2) Show that the category of complete lattices and functions preserving arbitrary glb's is isomorphic to the category of algebras for this monad. Hint: show that a complete lattice can be presented as a set X equipped with an operation $\bigwedge : \mathcal{P}X \to X$ such that $\bigwedge\{x\} = x$ and $\bigwedge\{\bigwedge X_j \mid j \in J\} = \bigwedge(\bigcup_j X_j)$ for any indexed family of subsets X_j).*

Definition B.8.6 (Kleisli category) *Given a monad (T, η, μ) over the category* $\mathbf{D}$*, the Kleisli category* $\mathbf{K}_T$ *is defined as:*

$$\begin{array}{ll} \mathbf{K}_T = \mathbf{D} & \mathbf{K}_T[d, d'] = \mathbf{D}[d, Td'] \\ id_d = \eta_d : d \to Td & f \circ g = \mu_{d''} \circ Tf \circ g \quad \text{for } g : d \to d', f : d' \to d'' \text{ in } \mathbf{K}_T \ . \end{array}$$

Theorem B.8.7 *(1) Every adjunction (L, R, η, ϵ) gives rise to a monad:*

$$T(L \dashv R) = (R \circ L, \eta, R\epsilon L) \ .$$

(2) *Given a monad (T, η, ϵ) over the category* $\mathbf{D}$*, consider the category of T-algebras* $\mathbf{Alg}_T$*. We can build an adjunction $(L^T, R^T, \eta^T, \epsilon^T)$ as follows:*

$$\begin{array}{ll} L^T(d) = \mu_d : T^2 d \to Td & L^T(f : d \to d') = T(f) \\ R^T(\alpha : Td \to d) = d & R^T(g : \alpha \to \beta) = g \\ \eta^T = \eta & \epsilon^T(\alpha : Td \to d) = \alpha \ . \end{array}$$

Moreover the monad induced by this adjunction is again (T, η, ϵ).

(3) *Given a monad (T, η, ϵ) over the category* $\mathbf{D}$*, consider the Kleisli category* $\mathbf{K_T}$ *then we can build an adjunction $(L^{K_T}, R^{K_T}, \eta^{K_T}, \epsilon^{K_T})$ as follows:*

$$\begin{array}{ll} L^{K_T}(d) = d & L^{K_T}(f : d \to d') = \eta_{d'} \circ f \\ R^{K_T}(d) = Td & R^{K_T}(f : d \to Td') = \mu_{d'} \circ Tf \\ \eta^{K_T} = \eta & \epsilon_d^{K_T} = id_{Td} \ . \end{array}$$

Moreover the monad induced by this adjunction is again (T, η, ϵ).

Given a monad T the Kleisli adjunction and the T-algebra adjunction can be shown to be initial and final, respectively, in a suitable category of adjunctions generating the monad T.

Bibliography

[ABL86] R. Amadio, K. Bruce, and G. Longo. The finitary projection model and the solution of higher order domain equations. In *Proc. Conference on Logic in Computer Science (LICS), Boston*, pages 122–130. IEEE, 1986.

[abPC79] M. Barr (appendix by P.H. Chu). *$\star$-autonomous categories*. Lecture Notes in Mathematics, 752. Springer-Verlag, 1979.

[Abr91a] S. Abramsky. A domain equation for bisimulation. *Information and Computation*, 92:161–218, 1991.

[Abr91b] S. Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51:1–77, 1991.

[AC93] R. Amadio and L. Cardelli. Subtyping recursive types. *Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. Also appeared as TR 62 DEC-SRC and TR 133 Inria-Lorraine. Short version appeared in Proc. POPL 91, Orlando.

[AC94] R. Amadio and P.-L. Curien. Selected domains and lambda calculi. Technical Report 161, INRIA-Lorraine, 1994. Lecture Notes.

[ACCL92] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1-4:375–416, 1992.

[Acz88] P. Aczel. *Non-well founded sets*. CSLI Lecture Notes 14, 1988.

[AHMP95] A. Avron, F. Honsell, I. Mason, and R. Pollak. Using typed lambda calculus to implement formal systems on a machine. *J. of Automated Reasoning*, 1995. To appear.

[AJ92] S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. In *Proc. 12th FST-TCS, Springer Lect. Notes in Comp. Sci. 652*, 1992.

[AJM95] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. Imperial College, 1995.

[AL87] R. Amadio and G. Longo. Type-free compiling of parametric types. In M. Wirsing, editor, *Formal Description of Programming Concepts-III*, pages 377–398. IFIP, North Holland, 1987. Presented at the IFIP Conference on Formal Description of Programming Concepts, Ebberup (DK), 1986.

[AL91] A. Asperti and G. Longo. *Categories, types, and structures*. MIT Press, 1991.

[ALT95] R. Amadio, L. Leth, and B. Thomsen. From a concurrent λ-calculus to the π-calculus. In *Proc. Foundations of Computation Theory 95, Dresden*. Springer Lect. Notes in Comp. Sci. 965, 1995. Expanded version appeared as ECRC-TR-95-18, available at `http://protis.univ-mrs.fr/~amadio/`.

[Ama89] R. Amadio. Formal theories of inheritance for typed functional languages. Technical report, Dipartimento di Informatica, Università di Pisa, 1989. TR 28/89.

[Ama91a] R Amadio. Bifinite domains: Stable case. In Pitt &al., editor, *Proc. Category Theory in Comp. Sci. 91, Paris*, pages 16–33. Springer Lect. Notes in Comp. Sci. 530, 1991. Full version appeared as TR 3-91, Liens Paris.

[Ama91b] R. Amadio. Domains in a realizability framework. In Abramsky and Maibaum, editors, *Proc. CAAP 91, Brighton*, pages 241–263. Springer Lect. Notes in Comp. Sci. 493, 1991. Extended version appeared as TR 19-90, Liens Paris.

[Ama91c] R. Amadio. Recursion over realizability structures. *Information and Computation*, 91(1):55–85, 1991. Preliminary version appeared as TR 1/89, Dipartimento di Informatica, Università di Pisa.

[Ama93] R. Amadio. On the reduction of Chocs bisimulation to π-calculus bisimulation. In *Proc. CONCUR 93, Hildesheim*, pages 112–126. Springer Lect. Notes in Comp. Sci. 715, 1993. Also appeared as Research Report Inria-Lorraine 1786, October 1992.

[Ama94] R. Amadio. Translating Core Facile. Technical Report ECRC-94-3, ECRC, Munich, 1994. Available at `http://protis.univ-mrs.fr/~amadio/`.

[Ama95] R. Amadio. A quick construction of a retraction of all retractions for stable bifinites. *Information and Computation*, 116(2):272–274, 1995. Also appeared as RR 1785 Inria-Lorraine, 1992.

[AN80] A. Arnold and M. Nivat. Metric interpretation of infinite trees and semantics of non-deterministic recursive programs. *Theoretical Computer Science*, 11:181–205, 1980.

[AP90] M. Abadi and G. Plotkin. A per model of polymorphism and recursive types. In *Proc. LICS*, 1990.

[App92] A. Appel. *Compiling with continuations*. Cambridge University Press, Cambridge, 1992.

[AZ84] E. Astesiano and E. Zucca. Parametric channels via label expressions in CCS. *Theoretical Computer Science*, 33:45–64, 1984.

[Bar84] H. Barendregt. *The lambda calculus; its syntax and semantics*. North-Holland, 1984.

[Bar91a] H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1:125–154, 1991.

[Bar91b] M. Barr. *-autonomous categories and linear logic. *Mathematical Structures in Computer Science*, 1(2):159–178, 1991.

[BC85] G. Berry and P.-L. Curien. The kernel of the applicative language cds: Theory and practice. *Proc. French-US Seminar on the Applications of Algebra to Language Definition and Compilation*, 1985. Cambridge University Press.

[BCD83] H. Barendregt, M. Coppo, and M. Dezani. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48:931–940, 1983.

[BE93] A. Bucciarelli and T. Ehrhard. A theory of sequentiality. *Theoretical Computer Science*, 113:273–291, 1993.

[BE94] A. Bucciarelli and T. Ehrhard. Sequentiality in an extensional framework. *Information and Computation*, 110:265–296, 1994.

[Ber78] G. Berry. Stable models of typed lambda-calculi. In *Proc. ICALP*, Springer Lect. Notes in Comp. Sci. 62, 1978.

[Ber79] G. Berry. *Modèles complètement adéquats et stables des lambda-calculs typés*. PhD thesis, Université Paris VII (Thèse d'Etat), 1979.

[Ber91] S. Berardi. Retractions on dI-domains as a model for type:type. *Information and Computation*, 94:204–231, 1991.

[Bet88] I. Bethke. *Notes on partial combinatory algebras*. PhD thesis, Amsterdam, 1988.

[BGG91] B. Berthomieu, D. Giralt, and J.-P. Gouyon. LCS users' manual. Technical report 91226, LAAS/CNRS, 1991.

[BH76] B. Banaschewski and R. Herrlich. Subcategories defined by implications. *Houston J. Math*, 2:149–171, 1976.

[Bie95] G. Bierman. What is a categorical model of intuitionistic linear logic? In *Typed Lambda Calculi and Applications*. Springer Lect. Notes in Comp. Sci. 902, 1995.

[BL88] K. Bruce and G. Longo. A modest model of records, inheritance, and bounded quantification. In *Proc. LICS*, 1988.

[BL94] G. Boudol and C. Laneve. The discriminating power of multiplicites in the λ-calculus. Technical report, Research Report 2441, INRIA-Sophia-Antipolis, 1994.

[Bla72] A. Blass. Degrees of indeterminacy of games. *Fundamenta Mathematicae*, LXXVII:151–166, 1972.

[Bla92] A. Blass. A game semantics for linear logic. *Annals of Pure and Applied Logic*, 56:183–220, 1992.

[Bou93] G. Boudol. Some chemical abstract machines. In *Springer Lect. Notes in Comp. Sci. 803: Proceedings of REX School*. Springer-Verlag, 1993.

[Bou94] G. Boudol. Lambda calculi for (strict) parallel functions. *Information and Computation*, 108:51–127, 1994.

[BS83] G. Buckley and A. Silberschatz. An effective implementation for the generalized input-output construct of CSP. *Transactions on Programming Languages and Systems*, 5(2):223–235, 1983.

[BTG91] V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic strong normalization. *Theoretical Computer Science*, 83(3-28), 1991.

[Buc93] A. Bucciarelli. Another approach to sequentiality: Kleene's unimonotone functions. In *Proc. MFPS, Springer Lect. Notes in Comp. Sci. 802*, 1993.

[BW85] M. Barr and C. Wells. *Toposes, triples and theories*. Springer-Verlag, 1985.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[CCF94] R. Cartwright, P.-L. Curien, and M. Felleisen. Fully abstract semantics for observably sequential languages. *Information and Computation*, pages 297–401, 1994.

[CCM87] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Sci. of Comp. Programming*, 8:173–202, 1987.

[CDC80] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980.

[CDHL82] M. Coppo, M. Dezani, F. Honsell, and G. Longo. Extended type structures and filter lambda models. In *Proc. Logic Coll. 1982*, Lolli et al. (ed.), North-Holland, 1982.

[CF58] H. Curry and R. Feys. *Combinatory Logic, vol. 1.* North Holland, 1958.

[CF92] R. Cartwright and M. Felleisen. Observable sequentiality and full abstraction. *Proc. POPL*, 1992.

[CGW88] T. Coquand, C. Gunter, and G. Winskel. Domain theoretic models of polymorphism. *Information and Computation*, 81:123–167, 1988.

[CH88] T. Coquand and G. Huet. A calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[CH94] P.-L. Curien and T. Hardin. Yet yet a counterexample for λ+SP. *Journal of Functional Programming*, 4(1):113–115, 1994.

[CHR92] P.-L. Curien, T. Hardin, and A. Rios. Strong normalisation of substitutions. In *Mathematical Foundations of Computer Science*, volume 629 of *Springer Lect. Notes in Comp. Sci.*, 1992.

[CO88] P.-L. Curien and A. Obtulowicz. Partiality, cartesian closedness and toposes. *Information and Computation*, 80:50–95, 1988.

[Coq89] T. Coquand. Categories of embeddings. *Theoretical Computer Science*, 68:221–237, 1989.

[CPW96] P.-L. Curien, G. Plotkin, and G. Winskel. Bistructures, bidomains and linear logic. *Milner's Festschrift*, 1996. MIT Press, to appear. Preliminary version by G. Plotkin and G. Winskel in Proc ICALP 94, Springer Lect. Notes in Comp. Sci.

[Cur86] P.-L. Curien. *Categorical combinators, sequential algorithms and functional programming.* Pitman, 1986. Revised edition, Birkhäuser, 1993.

[Cur91] P.-L. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.

[Dam94] M. Dam. On the decidability of process equivalence for the π-calculus. SICS report RR 94:20, 1994. Available at `ftp.sics.se`.

[Dan90] V. Danos. *La logique linéaire appliquée à l'étude de divers processus de normalisation (principalement le λ-calcul).* PhD thesis, Université Paris VII, 1990.

[dB80] N. G. de Bruijn. A survey of the project Automath. *Curry Festschrift, Hindley-Seldin (eds)*, pages 589–606, 1980. Academic Press.

[DF92] O. Danvy and A. Filinski. Representing control: a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(361-391), 1992.

[DR93] M. Droste and Göbel R. Universal domains and the amalgamation property. *Mathematical Structures in Computer Science*, 3:137–160, 1993.

[Dro89] M. Droste. Event structures and domains. *Theoretical Computer Science*, 68:37–48, 1989.

[Ehr93] T. Ehrhard. Hypercoherences: a strongly stable model of linear logic. *Mathematical Structures in Computer Science*, 3:365–385, 1993.

[Ehr96] T. Ehrhard. A relative PCF-definability result for strongly stable functions. LMD, Marseille, 1996.

[EM45] S. Eilenberg and S. MacLane. General theory of natural equivalences. *Trans. Am. Math. Soc.*, 58:231–241, 1945.

[EN86] U. Engberg and M. Nielsen. A calculus of communicating systems with label passing. Technical report, DAIMI PB-208, Computer Science Department, Aarhus, Denmark, 1986.

[Eng81] E. Engeler. Algebras and combinators. *Algebra Universalis*, 13:389–392, 1981.

[FFKD87] M. Felleisen, D. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52:205–237, 1987.

[FMRS92] P. Freyd, P. Mulry, G. Rosolini, and D. Scott. Extensional pers. *Information and Computation*, 98:211–227, 1992.

[Fri73] H. Friedman. Equality between functionals. In *Proc. Logic Colloquium, Springer Lect. Notes in Mathematics 453*, 1973.

[Fri78] H. Friedman. Classically and intuitionistically provably recursive functions. In *Higher set theory, Springer Lect. Notes in Mathematics 699*, 1978.

[GD62] A. Gleason and R. Dilworth. A generalized Cantor theorem. In *Proc. Amer. Math. Soc. 13*, 1962.

[GHK+80] G. Gierz, K. Hofmann, K. Keimel, J. Lawson, M. Mislove, and D. Scott. *A compendium of continuous lattices*. Springer-Verlag, 1980.

[Gir72] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[Gir86] J.-Y. Girard. The system F of variable types, fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

[Gir87] J-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GJ90] C. Gunter and A. Jung. Coherence and consistency in domains. *J. of Pure and Applied Algebra*, 63:49–66, 1990.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[GMP89] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.

[GN91] H. Geuvers and M. Nederhof. Modular proof of strong normalization for the calculus of constructions. *Journal of Functional Programming*, 1(2):155–189, 1991.

[Gog91] J. Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1:49–68, 1991.

[Gri90] T. Griffin. A formulae-as-types notion of control. In *Proc. POPL*, San Francisco, 1990.

[Har89] T. Hardin. Confluence results for the pure strong categorical combinatory logic CCL: lambda-calculi as subsytems of CCL. *Theoretical Computer Science*, 65:291–342, 1989.

[Hen50] L. Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15:81–91, 1950.

[HF87] C. Haynes and D. Friedman. Embedding continuations in procedural objects. *Transactions on Programming Languages and Systems*, 9(4):397–402, 1987.

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of ACM*, 40:143–184, 1993.

[Hin69] R. Hindley. The principal type schema of an object in combinatory logic. *Trans. Amer. Math. Society*, 1969.

[Hin83] R. Hindley. The completeness theorem for typing lambda-terms. *Theoretical Computer Science*, 22:1–17, 1983.

[HO80] G. Huet and D. Oppen. Equations and rewrite rules: a survey. *in Formal Language Theory: Perspectives and Open Problems, R. Book ed., Academic Press*, pages 349–405, 1980.

[HO94] J. Hyland and L. Ong. On full abstraction for PCF. Cambridge Univ., 1994.

[How80] W. Howard. The formulas-as-types notion of construction. *Curry Festschrift, Hindley-Seldin (eds)*, pages 479–490, 1980. Academic Press. Manuscript circulated since 1969.

[HS86] R. Hindley and J. Seldin. *Introduction to combinators and lambda-calculus*. London Mathematical Society, 1986.

[Hyl76] M. Hyland. A syntactic characterization of the equality in some models of lambda calculus. *J. London Math. Soc.*, 2:361–370, 1976.

[Hyl82] M. Hyland. The effective topos. *The Brouwer Symposium*, 1982. Troelstra and Van Dalen (eds.).

[Hyl88] M. Hyland. A small complete category. *Annals of Pure and Applied Logic*, 40:135–165, 1988.

[Hyl90] M. Hyland. First steps in synthetic domain theory. In *Proc. Category Theory 90*. Springer-Verlag, 1990.

[Jec78] T. Jech. *Set Theory*. Academic Press, 1978.

[JMS91] B. Jacobs, E. Moggi, and T. Streicher. Relating models of impredicative type theories. In *Proc. CTCS 91, Paris*, Springer Lect. Notes in Comp. Sci. 530, 1991.

[Joh82] P. Johnstone. *Stone Spaces*. Cambridge Studies in Adv. Math. 3, 1982.

[JT93] A. Jung and J. Tiuryn. A new characterization of lambda-definability. In *Proc. Conference on Typed lambda-calculus and applications, Springer Lect. Notes in Comp. Sci. 664*, 1993.

[Jun88] A. Jung. *Cartesian closed categories of domains*. CWI Tracts, Amsterdam, 1988.

[Jun90] A. Jung. The classification of continuous domains. In *Proc. LICS*, 1990. Philadelphia.

[Kle45] S. Kleene. On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic*, 10:109–124, 1945.

[Kle78] S. Kleene. Recursive functionals and quantifiers of finite types revisited I. In *Proc. General Recursion Theory II, Fenstad et al. (eds), North-Holland*, 1978.

[Kle80] S. Kleene. Recursive functionals and quantifiers of finite types revisited II. In *Proc. The Kleene Symposium, Barwise et al. (eds), North-Holland*, 1980.

[Kle82] S. Kleene. Recursive functionals and quantifiers of finite types revisited III. In *Proc. Patras Logic Symposium, North Holland*, 1982.

[Kle85] S. Kleene. Unimonotone functions of finite types (recursive functionals and quantifiers of finite types revisited IV). In *Proc. Symposia in Pure Mathematics 42*, 1985.

[Klo85] J.W. Klop. *Combinatory Reduction Systems.* PhD thesis, Utrecht university, 1985.

[KP93] G. Kahn and G. Plotkin. Concrete domains. *Theoretical Computer Science*, 121:187–277, 1993. Appeared as TR IRIA-Laboria 336 in 1978.

[Kri91] J.-L. Krivine. *Lambda-calcul, types et modèles.* Masson, 1991.

[Lam92a] F. Lamarche. Games, additives and correctness criteria. Manuscript, LIENS, Paris, 1992.

[Lam92b] F. Lamarche. Sequentiality, games and linear logic. Manuscript, LIENS, Paris, 1992.

[Lam94] F. Lamarche. From Chu spaces to cpo's. 1994.

[Lan64] P. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[Lan66] P. Landin. The next 700 programming languages. *Communications ACM*, 3, 1966.

[Lev78] J.-J. Levy. *Réductions correctes et optimales dans le λ-calcul.* PhD thesis, Université Paris VII, 1978.

[Lis88] B. Liskov. Data abstraction and hierarchy. In *Proc. OOPSLA*, Sigplan Notices 23-5, 1988.

[LM84] G. Longo and E. Moggi. Cartesian closed categories of enumerations and effective type structures. In *Symposium on Semantics of Data Types*, Springer Lect. Notes in Comp. Sci. 173, 1984.

[LM92] G. Longo and E. Moggi. Constructive natural deduction and its modest interpretation. *Mathematical Structures in Computer Science*, 1:215–254, 1992.

[Loa94] R. Loader. The undecidability of λ-definability. In *Proc. Church Festschrift CSLI/University of Chicago Press, Zeleny (ed.)*, 1994.

[LRS94] Y. Lafont, B. Reus, and T. Streicher. Continuation semantics: Abstract machines and control operators. University of Munich, 1994.

[LS86] J. Lambek and P. Scott. *Introduction to higher order categorical logic.* Cambridge University Press, 1986.

[Mar76] G. Markowsky. Chain-complete p.o. sets and directed sets with applications. *Algebra Universalis*, 6:53–68, 1976.

[McC84] D. McCarty. *Realizability and recursive mathematics.* PhD thesis, Oxford University, 1984.

[Mil77] R. Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4:–23, 1977.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mil92] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[Mit88] J. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76:211–249, 1988.

[ML71] S. Mac Lane. *Categories for the working mathematician.* Springer Verlag, New York, 1971.

[ML83] P. Martin-Löf. Lecture notes on the domain interpretation of type theory. Technical report, in Proc. Workshop on Semantics of Programming Languages, Programming Methodology Group, Chalmers University of Technology, Göteborg, 1983.

[ML84] P. Martin-Löf. *Intuitionistic type theory.* Bibliopolis, 1984.

[Mog88] E. Moggi. Partial morphisms in categories of effective objects. *Information and Computation*, 76:250–277, 1988.

[Mog89] E. Moggi. Computational lambda-calculus and monads. *Information and Computation*, 93:55–92, 1989.

[MPW92] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Process, Parts 1-2. *Information and Computation*, 100(1):1–77, 1992.

[MS92] R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proc. ICALP 92, Springer Lect. Notes in Comp. Sci. 623*, 1992.

[Mul81] P. Mulry. Generalized Banach-Mazur functionals in the topos of recursive sets. *J. of Pure and Applied Algebra*, 26:71–83, 1981.

[Mul89] K. Mulmuley. *Full abstraction and semantic equivalence.* MIT Press, 1989.

[Mur91] C. Murthy. An evaluation semantics for classical proofs. In *Proc. LICS*, Amsterdam, 1991.

[Par81] D. Park. Concurrency and automata on infinite sequences. In *Springer Lect. Notes in Comp. Sci. 104*, 1981.

[Pau87] L. Paulson. *Logic and Computation, Interactive proof with Cambridge LCF.* Cambridge University Press, 1987.

[Pho90] W. Phoa. Effective domains and intrinsic structure. In *Proc. LICS*, 1990.

[Pit95] A. Pitts. Relational properties of domains. *Information and Computation*, 1995. To appear.

[PJ87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[Plo75] G. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Plo76] G. Plotkin. A powerdomain construction. *SIAM J. of Computing*, 5:452–487, 1976.

[Plo77] G. Plotkin. LCF as a programming language. *Theoretical Computer Science*, 5:223–257, 1977.

[Plo83] G. Plotkin. Domains. Lecture Notes, Edinburgh University, 1983.

[Plo85] G. Plotkin. Denotational semantics with partial functions. Lecture Notes, CSLI-Stanford, 1985.

[Pra65] D. Prawitz. *Natural deduction.* Almqvist and Wiksell, Stockholm, 1965.

[RdR93] S. Ronchi della Rocca. Fundamentals in lambda-calculus. Summer School in Logic for Computer Science, Chambery, 1993.

[Rep91] J. Reppy. CML: A higher-order concurrent language. In *Proc. ACM-SIGPLAN 91, Conf. on Prog. Lang. Design and Impl.*, 1991.

[Rey70] J. Reynolds. Gedanken: a simple typeless programming language based on the principle of completeness and the reference concept. *Communications ACM*, 5, 1970.

[Rey74] J. Reynolds. Towards a theory of type structures. In *Colloque sur la Programmation*, Springer Lect. Notes in Comp. Sci. 19, 1974.

[Rog67] H. Rogers. *Theory of recursive functions and effective computability.* MacGraw Hill, 1967.

[Ros86] G. Rosolini. *Continuity and effectivity in topoi.* PhD thesis, Oxford University, 1986.

[RR88] E. Robinson and P. Rosolini. Categories of partial maps. *Information and Computation*, 79:95–130, 1988.

[Sal66] A. Salomaa. Two complete systems for the algebra of complete events. *Journal of ACM*, 13-1, 1966.

[San92] D. Sangiorgi. *Expressing mobility in process algebras: first-order and higher order paradigms.* PhD thesis, Edinburgh University, September 1992.

[Sco72] D. Scott. Continuous lattices. In Lawvere, editor, *Proc. Toposes, Algebraic Geometry and Logic*, pages 97–136. Springer Lect. Notes in Mathematics 274, 1972.

[Sco76] D. Scott. Data types as lattices. *SIAM J. of Computing*, 5(522-587), 1976.

[Sco80] D. Scott. Relating theories of the lambda-calculus. *Curry Festschrift, Hindley-Seldin (eds)*, pages 589–606, 1980. Academic Press.

[Sco82] D. Scott. Domains for denotational semantics. In *Proc ICALP*, Springer Lect. Notes in Comp. Sci. 140, 1982.

[Sco93] D. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121:411–440, 1993. Manuscript circulated since 1969.

[See89] R. Seely. Linear logic, $*$-autonomous categories and cofree coalgebras. *Applications of categories in logic and computer science, Contempory Mathematics*, 92, 1989.

[Sie92] K. Sieber. Reasoning about sequential functions via logical relations. In *Proc. Applications of Categories in Computer Science, LMS Lecture Note Series, Cambridge University Press*, 1992.

[Smy78] M. Smyth. Power domains. *Journal of Computer and System Sciences*, 16:23–36, 1978.

[Smy83a] M. Smyth. The largest cartesian closed category of domains. *Theoretical Computer Science*, 27:109–119, 1983.

[Smy83b] M. Smyth. Powerdomains and predicate transformers. Springer Lect. Notes in Comp. Sci. 154, 1983.

[Soa87] R. Soare. *Recursively enumerable sets and degrees.* Springer-Verlag, 1987.

[SP82] M. Smyth and G. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. of Computing*, 11:761–783, 1982.

[Sto94] A. Stoughton. Mechanizing logical relations. In *Proc. MFPS, Springer Lect. Notes in Comp. Sci. 802*, 1994.

[Tay90a] P. Taylor. An algebraic approach to stable domains. *J. of Pure and Applied Algebra*, 64:171–203, 1990.

[Tay90b] P. Taylor. The trace factorisation of stable functors. Manuscript, Imperial College, 1990.

[Tho93] B. Thomsen. Plain Chocs. *Acta Informatica*, 30:1–59, 1993. Also appeared as TR 89/4, Imperial College, London.

[TLP+93] B. Thomsen, L. Leth, S. Prasad, T.M. Kuo, A. Kramer, F. Knabe, and A. Giacalone. Facile Antigua release programming guide. Technical Report ECRC-93-20, ECRC, Munich, December 1993. Available at `ftp.ecrc.de`.

[TvD88] A. Troelstra and D. van Dalen. *Constructivism in mathematics (2 volumes)*. North-Holland, 1988.

[Vic89] S. Vickers. *Topology via logic*. Cambridge University Press, 1989.

[vR96] F. van Raamsdonk. *Confluence and Normalisation for Higher-Order Rewriting Systems*. PhD thesis, Vrije Universitet,Amsterdam, 1996.

[Vui74] J. Vuillemin. *Syntaxe, sémantique et axiomatique d'un langage de programmation simple*. PhD thesis, Université Paris VII, 1974.

[Wad76] C. Wadsworth. The relation between computational and denotational properties for Scott's D-infinity-models of the lambda-calculus. *SIAM J. of Computing*, 5:488–521, 1976.

[Wan79] M. Wand. Fixed-point constructions in order-enriched categories. *Theoretical Computer Science*, 8:13–30, 1979.

[Wel94] J. Wells. Typability and type-checking in the second order λ-calculus are equivalent and undecidable. In *LICS*, Paris, 1994.

[Win80] G. Winskel. *Events in computation*. PhD thesis, Edinburgh University, 1980. PhD thesis.

[Win86] G. Winskel. Event structures. Springer Lect. Notes in Comp. Sci. 255, 1986.

[Win93] G. Winskel. *The formal semantics of programming languages*. MIT-Press, 1993.

[Zha91] G. Zhang. *Logic of domains*. Birkhäuser, 1991.

List of Figures

Index