Logical Semantics of Types for Concurrency

Luís Caires

CITI / Departamento de Informática, Universidade Nova de Lisboa, Portugal

Abstract. We motivate and present a logical semantic approach to types for concurrency and to the soundness of related systems. The approach is illustrated by the development of a generic type system for the π-calculus, which may be instantiated for specific notions of typing by extension with adequate subtyping principles. Soundness of our type system is established using a logical predicate technique, based on a compositional spatial logic interpretation of types.

1 Introduction

The aim of this paper is to present a semantic approach to types for concurrency and soundness of related systems, based on spatial logic interpretations. Types are definitely one of the most successful applications of logical methods in concrete programming languages and tools. A type system for a programming language or programming calculus should really be seen as a specialized logic, usually decidable, and presented by a syntax-directed proof system. A classical example is the familiar type system for assigning simple functional types to the λ-calculus. In this case, the properties of interest are absence of errors due to undefined function applications, and (last but not the least) strong normalization. In this case, termination is obtained as a consequence of the soundness of the simple type system with respect to a logical predicate interpretation [22].

As programming languages and calculi evolved, so to include increasingly sophisticated features such as state, exceptions, polymorphism, and concurrency, it has become clearer that classical semantic approaches to prove soundness of type systems did not scale or generalize very well, due to the independent difficulty of finding suitable semantic domains. Fortunately, if one is essentially interested in properties of programs such as absence of certain types of runtime errors, and not really in higher (logical) complexity properties such as termination, more convenient, purely syntactic, proof techniques may frequently be used. As Curry and Feys have put in [9] if one makes sure that "subject reduction preserves the predicate", and the "predicate" implies absence of immediate errors, then any "subject" program that satisfies the predicate is safe. Motivated by this remark, the (now standard) technique of "subject-reduction" (SR) was first proposed by Felleisen and Wright [23], by letting the "predicate" be identified with formal provability of typing judgments in a system of typing rules.

The SR soundness proof method has certainly been very successful, and revealed to be applicable to various kinds of languages and calculi, in particular, to types for concurrency. In fact, most modern type theoretic analyses of concurrent, distributed, and mobile calculi have been developed in such a framework.

Nevertheless, the purely syntactic SR method is not without its weaknesses, and sometimes appears to have contributed to widespread a too syntactic understanding of types and typing, far from the original semantic view of types as explicit properties or predicates. Usually, SR soundness proofs are quite monolithic, and each intermediate result proceeds by tedious inductions on type derivations. Adding a new construct to the programming language or a new typing rule to the system forces a cross-cutting modification on several auxiliary proofs. This lack of modularity is also caused by the usual absence of any independently defined compositional (algebraic, co-algebraic, or logical) semantics for the type structure. Although one may be careful enough to define such a semantics, unfortunately the SR method does not require such a semantics to be formally defined. Thus, usually we just find some useful but informal intuitions about what the typing rules or the types are intended to mean. It also seems that the SR methods does not by itself improve the degree of reuse foreseen in [23], given the particularities of each operational model.

On the other hand, a semantic proof of soundness builds on an explicitly defined compositional interpretation of types, that potentially provides deeper intuitions, and focuses the proof developments on behavioral aspects of the computational domain, rather than on details of the syntactic presentation of a calculus or of their types as syntactic annotations. In principle, the semantic technique is also more powerful, inducing in general some form of compositionality of typing, and being potentially applicable to properties that are not provable by the SR method (such as termination).

In the original spirit of semantic soundness proofs, we develop in this paper a feasible approach to types for concurrency that combines the advantages of the semantic approach with the technical simplicity of syntactic approaches, such as the SR method. More precisely, we show how the semantics of a general type structure for processes modeled in the π-calculus may be compositionally defined by resorting to a logical interpretation, reminiscent of the logical predicate (or relations) method, and considering as underlying semantic model the standard labeled transition system and associated operational techniques. As in purely semantic approaches, we proceed by defining a compositional semantics of the type language, by induction on the type structure. A formal type system then assigns types to processes by induction on the structure of processes. We illustrate the approach by developing a generic type system $\mathbf{T}$ for the π-calculus, and prove its soundness by showing that typing preserves the validity of typing assertions with respect to an interpretation of types as process predicates.

The generic type system $\mathbf{T}$ may be instantiated to check for various specific properties, just by extending it with appropriate (sound) subtyping principles. In fact, a remarkable advantage of the approach is due to the way the several properties of interest may be factored out. For example, subtyping may be dealt with as a completely orthogonal aspect, so that our soundness proof does not depend on the syntactic presentation of subtyping, but only on its semantic properties. So, we can pick for subtyping any sound axiomatization of semantic entailment in the underlying logic; soundness of each instance of $\mathbf{T}$ is then immediately granted as a consequence of this modular approach.

Typically, most interesting process properties of the kinds considered by type systems (*e.g.*, channel arity mismatch) are not invariant under standard behavioral equivalences of processes, for instance, bisimilarity. Therefore, to characterize such kind of properties, the traditional behavioral logics (cf. Hennessy-Milner logics [11]) are not adequate. It turns out that spatial logics for concurrency offer the appropriate expressiveness, as already argued elsewhere [4, 5, 1, 7].

Spatial logics have been proposed with the aim of specifying distributed behavior and other essential aspects of distributed computing systems. An important feature of spatial logics, shared by some other sub-structural logics such as separation logics [20, 18], is that its operators are able to separate and count resources; this sometimes seems to add an "intensional" character to these logics (although not always [6]). It is precisely such intensional character that seems necessary for the logical characterization of many type-like properties [4, 1]. So, our type language combines behavioral operators, that observe process actions, with spatial logic operators, namely the composition $A \mid B$, and its adjunct $A \triangleright B$. Then, a judgment in the type system $\mathbf{T}$, of the form $P :: A \vdash B$, expresses a rely guarantee property and is interpreted by the $A \triangleright B$ operation.

The structure of the paper is as follows. In Section 2, we present an overview of the syntax and semantics of the fragment of the π-calculus we will base our study on. The main intent of Section 3 is to motivate the semantic approach to typing, by providing an alternative proof of soundness for a standard simple type system for the π-calculus. In Section 4, we develop and present the generic type system $\mathbf{T}$, and prove its soundness with respect to a logical predicate semantics. We will also consider several incremental extensions to $\mathbf{T}$. In Section 5, we will show how $\mathbf{T}$ may be instantiated so to capture some familiar notions of typing, namely the simple types, I/O types, and some kind of session types. We will close the paper with some conclusions and remarks.

2 The Process Model

In this section, we briefly introduce the syntax and semantics of our intended process model, a fragment of the monadic π-calculus.

Definition 2.1 (Processes). *Given infinite sets Λ of* names (m, n, p)*, and χ of* process variables $(\mathcal{X}, \mathcal{Y})$ *the set $\mathcal{P}$ of* processes (P, Q, R) *is given by*

$$P, Q ::= \mathbf{0} \;\mid\; m(n).P \;\mid\; m\langle n\rangle.P \;\mid\; P \mid Q \;\mid\; (\boldsymbol{\nu} n)P \;\mid\; \mathcal{X} \;\mid\; \texttt{rec}\, \mathcal{X}.P$$

In restriction $(\boldsymbol{\nu} n)P$ and input $m(n).P$ the distinguished occurrence of name n is binding, with scope the process P. We denote by $\equiv_\alpha$ the relation of α-equivalence on processes: we will implicitly consider processes up to α-equivalence, with care. For any process P, we assume defined as usual the set $fn(P)$ of *free names* of P. By $\{^m/_n\}$ (resp. $\{^{\mathcal{X}}/_Q\}$) we denote the safe substitution of m by n (resp. of $\mathcal{X}$

by Q), and by $\{m \leftrightarrow n\}$ the safe transposition of m and n. Structural congruence expresses basic identities on the spatial structure of processes:

Definition 2.2 (Structural congruence). *Structural congruence $\equiv$ is the least congruence relation on processes such that*

$$
\begin{array}{ll}
P \mid \mathbf{0} \equiv P & (\textit{Struct Par Void}) \\
P \mid Q \equiv Q \mid P & (\textit{Struct Par Comm}) \\
P \mid (Q \mid R) \equiv (P \mid Q) \mid R & (\textit{Struct Par Assoc}) \\
n \notin \mathit{fn}(P) \Rightarrow P \mid (\boldsymbol{\nu} n)Q \equiv (\boldsymbol{\nu} n)(P \mid Q) & (\textit{Struct Res Par}) \\
(\boldsymbol{\nu} n)\mathbf{0} \equiv \mathbf{0} & (\textit{Struct Res Void}) \\
(\boldsymbol{\nu} n)(\boldsymbol{\nu} m)P \equiv (\boldsymbol{\nu} m)(\boldsymbol{\nu} n)P & (\textit{Struct Res Comm}) \\
\texttt{rec}\ \mathcal{X}.P \equiv P\{^{\mathcal{X}}/_{\texttt{rec}\ \mathcal{X}.P}\} & (\textit{Struct Unfold})
\end{array}
$$

The behavior of processes is defined by a relation of reduction that captures the computations that a process may perform by itself.

Definition 2.3 (Reduction). Reduction *$(P \rightarrow Q)$ is defined as follows:*

$$
\begin{array}{ll}
m\langle n\rangle.Q \mid m(p).P \rightarrow Q \mid P\{^{p}/_{n}\} & (\textit{Red React}) \\
Q \rightarrow Q' \Rightarrow P \mid Q \rightarrow P \mid Q' & (\textit{Red Par}) \\
P \rightarrow Q \Rightarrow (\boldsymbol{\nu} n)P \rightarrow (\boldsymbol{\nu} n)Q & (\textit{Red Res}) \\
P \equiv P',\ P' \rightarrow Q',\ Q' \equiv Q \Rightarrow P \rightarrow Q & (\textit{Red Struct})
\end{array}
$$

We denote by $\Rightarrow$ the reflexive-transitive closure of $\rightarrow$. We say that $P \mapsto Q$ if $P \rightarrow Q$ results from a communication on a restricted channel name of P. By $\Mapsto$ we denote the reflexive-transitive closure of $\mapsto$. To observe the interaction between a process and its environment one introduces a labeled transition semantics, in this case, the standard (late) labeled transition system [21]. For that we introduce

Definition 2.4 (Labels). *Labels $\mathcal{L}$ (α, β) are define by*

$$L ::= (\nu n)\alpha \ \mid\ m\langle n\rangle \ \mid\ m(n) \ \mid\ \tau$$

Name restriction on labels is used to express bound output [21]. We assume defined the standard $\mathit{fn}(\alpha)$ (free names) and $\mathit{bn}(\alpha)$ (bound names) of label α.

Definition 2.5 (Labeled Transition System). *The relation of* labeled transition *$(P \overset{\alpha}{\rightarrow} Q)$ is defined by the rules:*

$$a(n).P \overset{a(n)}{\rightarrow} P\ (\mathit{In}) \qquad m\langle n\rangle.P \overset{m\langle n\rangle}{\rightarrow} P\ (\mathit{Out})$$

$$\frac{P \overset{\alpha}{\rightarrow} Q}{P \mid R \overset{\alpha}{\rightarrow} Q \mid R}\ (\mathit{Par}) \qquad \frac{P \overset{\alpha}{\rightarrow} Q \quad n \notin \mathit{fn}(\alpha)}{(\boldsymbol{\nu} n)P \overset{\alpha}{\rightarrow} (\boldsymbol{\nu} n)Q}\ (\mathit{Res}) \qquad \frac{P\{^{\mathcal{X}}/_{\texttt{rec}\ \mathcal{X}.P}\} \overset{\alpha}{\rightarrow} Q}{\texttt{rec}\ \mathcal{X}.P \overset{\alpha}{\rightarrow} Q}\ (\mathit{Rec})$$

$$\frac{P \overset{(\nu \overline{s})n\langle m\rangle}{\rightarrow} P' \quad Q \overset{n(p)}{\rightarrow} Q'}{P \mid Q \overset{\tau}{\rightarrow} (\boldsymbol{\nu} \overline{s})(P' \mid Q'\{^{p}/_{m}\})}\ (\mathit{Com}) \qquad \frac{P \overset{m\langle n\rangle}{\rightarrow} Q}{(\boldsymbol{\nu} n)P \overset{(\nu n)m\langle n\rangle}{\rightarrow} Q}\ (\mathit{Open})$$

The following provisos apply: rule (Par) subject to $\mathit{fn}(Q)\#\mathit{bn}(\alpha)$, rule (Com) subject to $\overline{s}\#Q$, rule (Open) subject to $p \neq m$.

Reduction $\rightarrow$ coincides with silent transition $\xrightarrow{\tau}$, and does not increase the free names of processes. We write $P \xrightarrow{n} Q$ when $P \xrightarrow{\alpha} Q$ and either $\alpha = (\nu\overline{s})n\langle m\rangle$ or $\alpha = n(m)$, and abbreviate a label $(\nu\overline{s})\alpha$ by α when the identity of $\overline{s}$ is not important. Strong late bisimilarity over the labeled transition system defined above is taken as our reference behavioral semantic equivalence of processes.

Definition 2.6. A strong late bisimulation $\mathcal{R}$ *is a symmetric binary relation over processes such that for all* P, Q

- *If* $P \mathrel{\mathcal{R}} Q$ *and* $P \xrightarrow{\alpha} P'$ *for some* P' *then exists* Q' *st.* $Q \xrightarrow{\alpha} Q'$ *and* $P' \mathrel{\mathcal{R}} Q'$.
- *If* $P \mathrel{\mathcal{R}} Q$ *and* $P \xrightarrow{n(p)} P'$ *for some* P' *then exists* Q' *st.* $Q \xrightarrow{n(p)} Q'$ *and for all* m $P'\{^{p}/_{m}\} \mathrel{\mathcal{R}} Q'\{^{p}/_{m}\}$.

Strong late bisimilarity $\sim$ *is the greatest strong late bisimulation.*

There are well known characterizations of bisimilarities using modal logics. These are mostly variants of Hennessy-Milner logics [11]. For the strong late bisimilarity case, we may consider the logic $\mathcal{LM}$ [16]: essentially Hennessy-Milner logic augmented with a modality $\langle x(y)\rangle^{E}A$, such that

$$P \models \langle x(y)\rangle^{E}A \textit{ iff All } Q.P \xrightarrow{x(y)} Q \textit{ implies All } m.\ Q\{^{y}/_{m}\} \models A\{^{y}/_{m}\}$$

Two processes P and Q are defined to be *logically equivalent* ($P =_{\mathcal{L}} Q$) for a logic $\mathcal{L}$ if they satisfy exactly the same formulas of $\mathcal{L}$. For $\mathcal{LM}$, one have that $P =_{\mathcal{LM}} Q$ if and only if $P \sim Q$ [16]. In general, purely behavioral logics such as $\mathcal{LM}$ do not distinguish between bisimilar processes. As we shall see in the next section, the kind of properties captured by the simplest type systems for concurrency are not invariant under bisimilarity, and therefore cannot be expressed by logics that just rely on observing process actions.

3 Simple Types

The simplest type systems for concurrent systems modeled in the π-calculus, originating in Milner's system of sorts [15] for the polyadic π-calculus, are intended to enforce communication safety. If a process tries to communicate on a shared channel, but the sender issues a tuple of length different from the one expected by the receiver, an error occurs (undefined synchronization). In our simpler monadic setting, we introduce a slightly different, but in some sense equivalent, notion of error. We partition the set of names Λ into disjoint subsets of channel names Λ_c (x, a, b) and basic names Λ_v (v, u). Then, while channel names may be used to send and receive values, a basic name (cf., an integer value) cannot. More precisely, a process is *wrong*, when it attempts writing to or reading from something that does not refer to a communication channel.

$$\text{Wrong}(P) \triangleq (P \equiv (\boldsymbol{\nu}\overline{m})(a(n).Q \mid R) \text{ or } P \equiv (\boldsymbol{\nu}\overline{m})(a\langle n\rangle.Q \mid R)) \text{ and } a \in \Lambda_v$$

N.B. This may be seen as a special case of arity mismatch: names in Λ_v cannot be used at any arity, and names in Λ_c may be used at all arities (since there is a single arity). We say a process is *safe* if not wrong: $\text{Safe}(P) \triangleq \neg\text{Wrong}(P)$.

Notice that being wrong is not a purely behavioral property, because a wrong process may be bisimilar to a safe process. A type system for arity matching is usually based on formal judgments of the form $\Gamma \vdash P$, where P is the process to be typed, and Γ a typing environment, more precisely, an assignment of a type T to *every* free name of P. We consider channel types (T) and a base type `nil`.

Definition 3.1. *The set of $\mathcal{ST}$ of simple types is given by $T, U, V ::= \mathtt{nil} \mid (T)$.*

Given a finite set of names N, a typing environment of domain N is a mapping Γ assigning each name $n \in D$ a type $\Gamma(n) \in T$ such that $n \in \Lambda_v$ implies $T = \mathtt{nil}$. We denote by $\mathcal{C}$ the set of all typing environments. We denote by $\mathfrak{D}(\Gamma)$ the domain N of Γ. As usual, the typing environment Γ of domain $\{n_1, \ldots, n_m\}$ that maps n_i to T_i may be written $n_1 : T_1, n_2 : T_2, \ldots, n_m : T_m$. Usually, types such as the simple types given above are seen as formal annotations, and type safety for the type system proven by resorting to a subject reduction result. In order to motivate our approach, we will instead develop a semantic proof of soundness. For that purpose, we need to define a compositional interpretation of typing environments as properties (sets of) of processes. We say that a mapping $J\llbracket - \rrbracket : \mathcal{C} \to \wp(\mathcal{P})$ is *conjunctive* if $J\llbracket \Gamma, \Delta \rrbracket = J\llbracket \Gamma \rrbracket \cap J\llbracket \Delta \rrbracket$.

Definition 3.2. *A* typing interpretation $J\llbracket - \rrbracket : \mathcal{C} \to \wp(\mathcal{P})$ *is a conjunctive mapping assigning to each typing environment a set of processes such that:*

If $P \in J\llbracket n : T \rrbracket$ *then* $\mathrm{Safe}(P)$
If $P \in J\llbracket n : T \rrbracket$ *and* $P \xrightarrow{\alpha} Q$ *then* $Q \in J\llbracket n : T \rrbracket$
If $P \in J\llbracket n : (U) \rrbracket$ *and* $P \xrightarrow{(\boldsymbol{\nu}\bar{s})n\langle m \rangle} Q$ *then* $Q \in J\llbracket m : U \rrbracket$
If $P \in J\llbracket n : (U) \rrbracket$ *and* $P \xrightarrow{n(m)} Q$ *then* $Q \in J\llbracket m : U \rrbracket$
If $P \in J\llbracket n : \mathtt{nil} \rrbracket$ *and* $P \xrightarrow{n} Q$ *then* $False$

Notice that $J\llbracket \Gamma \rrbracket_{\Gamma \in \mathcal{C}}$ is a (typing environment)-indexed family of sets of processes; inductively defined on types, co-inductively defined on transitions. This definition is parametric on the safety predicate $\mathrm{Safe}(-)$, and on standard behavioral observations on processes, expressed by transitions on a labeled transition system. Indeed, if $\mathrm{Safe}(-)$ were closed under bisimilarity (*e.g.*, if $P \sim Q$ and $\mathrm{Safe}(P)$ implies $\mathrm{Safe}(Q)$), then we might check that the corresponding typing interpretation would also be closed under bisimilarity, in the sense that if $P \in J\llbracket \Gamma \rrbracket$ and $P \sim Q$ then also $Q \in J\llbracket \Gamma \rrbracket$. However, as remarked above, the safety properties of interest captured by type systems are seldom purely behavioral, so that usually any correct (sound) logical interpretation of types is bound to be "intensional" (finer than usual extensional behavioral types). We can check that typing interpretations are closed under arbitrary unions.

Lemma 3.3. *Let $\mathcal{J}$ be a family of typing interpretations. Then $\bigcup_{J \in \mathcal{J}} J$ (defined pointwise as $\Gamma \mapsto \bigcup_{J \in \mathcal{J}} J\llbracket \Gamma \rrbracket$) is also a typing interpretation.*

We may then define our interpretation of typing environments.

Definition 3.4. *We define typing, noted $\mathcal{T}\llbracket - \rrbracket$, by letting, for all $\Gamma \in \mathcal{C}$,*

$$\mathcal{T}\llbracket \Gamma \rrbracket \quad \triangleq \quad \bigcup \{ J\llbracket \Gamma \rrbracket : \ J \textit{ is a typing interpretation} \}$$

By definition, $\mathcal{T}\llbracket - \rrbracket$ is the largest (with relation to the inclusion partial ordering) typing interpretation. It is immediate that if $P \in \mathcal{T}\llbracket \Gamma \rrbracket$ and $P \in \mathcal{T}\llbracket \Delta \rrbracket$ then $P \in \mathcal{T}\llbracket \Gamma, \Delta \rrbracket$ and conversely. We can already verify the key properties of our typing interpretation $\mathcal{T}\llbracket - \rrbracket$: these properties hold whenever the type covers all free names of the process. It is typical of predicates of terms defined via realizability or logical relations techniques to characterize the intended properties just when all free variables/names of the subject are covered. We then define

$$P \models_{\mathsf{s}} \Gamma \quad \triangleq \quad P \in \mathcal{T}\llbracket \Gamma \rrbracket \textit{ and } \mathit{fn}(P) \in \mathfrak{D}(\Gamma)$$

Lemma 3.5. *The following closure properties of $\mathcal{T}\llbracket - \rrbracket$ hold:*

1. $\mathbf{0} \models_s \Gamma$.
2. *If* $P \models_s \Gamma \wedge n : T \wedge m : T$ *then* $P\{^n/_m\} \models_s \Gamma \wedge m : T$.
3. *If* $P \models \Gamma$ *and* $n \notin \mathfrak{D}(\Gamma)$ *then* $P \models_s \Gamma \wedge n : T$.
4. *If* $P \models_s \Gamma$ *and* $Q \models_s \Gamma$ *then* $P \mid Q \models_s \Gamma$.
5. *If* $P \models_s \Gamma \wedge n : T$ *and* $n \notin \mathfrak{D}(\Gamma)$ *then* $(\boldsymbol{\nu} n)P \models_s \Gamma$.
6. *If* $P \models_s \Gamma$ *and* $\Gamma(n) = (U)$ *and* $\Gamma(m) = U$ *then* $n\langle m \rangle.P \models_s \Gamma$.
7. *If* $P \models_s \Gamma \wedge x : U$ *and* $\Gamma(n) = (U)$ *then* $n(x).P \models_s \Gamma$.

Proof. The proof of most cases is by coinduction, given the definition of $\mathcal{T}\llbracket - \rrbracket$. It is instructive to look at a few cases (full proofs of this and other results in [2]).

1. We have $\mathbf{0} \in \mathcal{T}\llbracket n : T \rrbracket$ for any n and T, since $\mathbf{0} \overset{\alpha}{\not\rightarrow}$. Hence $\mathbf{0} \models_{\mathsf{s}} \Gamma$ for any Γ.
2. We show that $S(\Gamma \wedge m : T) \triangleq \{ P\{^n/_m\} \mid P \models_{\mathsf{s}} \Gamma \wedge n : T \wedge m : T \}$ is a typing interpretation. Pick $R \in S(\Gamma \wedge m : T)$. Then $R = P\{^n/_m\}$ where $P \models_{\mathsf{s}} \Gamma \wedge n : T \wedge m : T$. Let $R \overset{\alpha}{\rightarrow} R'$.
   (a) If $\alpha = \tau$ then $R' \models_{\mathsf{s}} \Gamma \wedge n : T \wedge m : T$, and so $R'\{^n/_m\} \in S(\Gamma \wedge m : T)$.
   (b) if $\alpha = a(v)$ then $P \overset{b(v)}{\rightarrow} Q$ where $a = b\{^n/_m\}$. We have $Q \models_{\mathsf{s}} \Gamma \wedge n : T \wedge m : T \wedge v : V$ and $\Gamma_{m,n}(b) = (V)$. If $b \neq n$ then $R' \in S(\Gamma \wedge m : T \wedge v : V)$. If $b = n$ then $\alpha = m(v)$ and $T = (V)$. Then $R' \in S(\Gamma \wedge m : T \wedge v : V)$.
   (c) if $\alpha = (\boldsymbol{\nu}\overline{s})a\langle v \rangle$ then $P \overset{(\boldsymbol{\nu}\overline{s})b\langle q \rangle}{\rightarrow} Q$ where $a = b\{^n/_m\}$ and $v = q\{^n/_m\}$. We have $P \overset{(\boldsymbol{\nu}\overline{s})b\langle q \rangle}{\rightarrow} Q$, where $Q \models_{\mathsf{s}} \Gamma \wedge n : T \wedge m : T \wedge q : V$ and $\Gamma_{m,n}(b) = (V)$. If $b \neq n$ then $R' \in S(\Gamma \wedge m : T \wedge q : V)$. If $b = n$ then $\alpha = (\boldsymbol{\nu}\overline{s})m\langle v \rangle$ and $T = (\overline{V})$. Then $R' \in S(\Gamma \wedge m : T \wedge q : V)$.

   We conclude that S is a typing interpretation. Then $P \models_{\mathsf{s}} \Gamma \wedge n : T \wedge m : T$ implies $P\{^n/_m\} \in S(\Gamma \wedge m : T) \subseteq \mathcal{T}\llbracket \Gamma \wedge m : T \rrbracket$. So $P\{^n/_m\} \models_{\mathsf{s}} \Gamma \wedge m : T$. ■

Given the previous Lemma 3.5, it is immediate that the (standard) proof system ST depicted in Figure 1 is sound for simple types, in the following sense.

Proposition 3.6. *If* $\Gamma \vdash P$ *and* $\mathit{fn}(P) \in \mathfrak{D}(\Gamma)$ *then* $P \models_s \Gamma$.

$$\overset{(\,\textit{ST-Void}\,)}{\Gamma \vdash \mathbf{0}} \qquad \overset{(\,\textit{ST-Par}\,)}{\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \mid Q}} \qquad \overset{(\,\textit{ST-Res}\,)}{\frac{\Gamma \wedge m : U \vdash P}{\Gamma \vdash (\boldsymbol{\nu} n)P}}$$

$$\overset{(\,\textit{ST-Inp}\,)}{\frac{\Gamma \wedge m : U \vdash P \quad \Gamma(n) = (U)}{\Gamma \vdash n(m).P}} \qquad \overset{(\,\textit{ST-Out}\,)}{\frac{\Gamma \vdash P \quad \Gamma(n) = (U) \quad \Gamma(m) = U}{\Gamma \vdash n\langle m\rangle.P}}$$

Fig. 1. Simple type system.

It is interesting to compare the structure of our semantic proof of consistency with a subject reduction style proof. Obviously, both proofs build on the same operational model, and need to go through the verification of essentially the same properties of the processes. The main advantages of the semantic approach result from a fairly different underlying proof structure. The semantic proof is modular (on the structure of the calculus operations), while the subject reduction proof is not. The subject reduction proof proceeds by induction on reduction and typing derivations, while the semantic proof deals with each inference principle in isolation. Thus, to check the soundness of an extended system, one just needs to check the added rules, while a subject reduction proof would have to be mostly redone (or at least, add a new induction case to all existing auxiliary results). Other potential advantages, in particular the smooth incorporation of subtyping principles, were already discussed in the Introduction. Type safety properties are obtained "for free", as an internal consequence of the meaning of each property denoted by a type. For example, we directly conclude the semantic counterparts of the familiar type safety and subject reduction statements:

Proposition 3.7 (Type Safety).

1. *If* $P \models_s \Gamma$ *then* $\mathrm{Safe}(P)$*, and if* $P \models_s \Gamma$ *and* $P \rightarrow Q$ *then* $Q \models_s \Gamma$*.*
2. *If* $\Gamma \vdash P$ *and* $P \Rightarrow Q$ *then* $\mathrm{Safe}(Q)$*.*

Proof. (1) By definition of $\mathcal{T}[\![-]\!]$. (2) By (1) and Proposition 3.6. ∎

In the case of simple types just discussed, a standard SR proof would have been perhaps more concise. However, the advantages of the semantic approach start to become clearer when more complex type systems are considered. In the next section, we develop a general type system **T** for the π-calculus, based on a behavioral-spatial logic, and prove its soundness using the semantic approach illustrated above. As we shall show later in the paper, the logical primitives of this type system provide a suitable "meta-language" in which the (type-like, intensional) properties captured by many different type systems may be expressed as idioms. Although it would be straightforward to extend **T** with new inference rules, we will show that many interesting instances may be obtained by merely adding new subtyping axioms and rules. The soundness of such subtyping principles may be proven modularly and incrementally, resorting to the semantic approach. The soundness of each (conservative) extension of the type system **T** considered will then be obtained in fairly automatic way.

4 The Generic Type System T

In this section, we present a general type system for the π-calculus, motivated by a logical semantic of types as properties (sets of processes), and prove soundness of typing (Theorem 4.5) and subtyping (Theorem 4.6). Our presentation is close to a presentation of a logic. This is not unexpected, any type system should be seen as a compositional, decidable, and (usually incomplete) proof system for a specialized logic. We start by defining the syntax of types.

Definition 4.1 (Types). *Types of* $\boldsymbol{T}$ *are given by the following abstract syntax:*

$$\begin{array}{ll} \alpha & ::= x.!(T)\triangleright \;\mid\; x.?(T)\triangleright \;\mid\; x.!(T)\,; \;\mid\; x.?(T)\,; \\ A,B,C & ::= \emptyset \mid \mathsf{F} \mid A \wedge B \mid\; A|B \;\mid A \triangleright B \mid \mathsf{H}n.A \mid \alpha A \mid [\alpha]A \mid \Box A \\ & \mid\;\; \mathtt{rec}\; X.A \mid X \end{array}$$

N.B.: We write $T(n)$ *(also* $n : T$*) for a type* A *with a single name* n *occurring. Then, we refer by* T *the name-abstracted type* $T(-)$*.*

Name abstracted types, such as T above, appear as arguments to behavioral operators α, to type channels parameters. *E.g.*, if $T(-) = -.!()\,;\; -.?()\,;\emptyset$, then $T(n) = n : T = n.!()\,;n.?()\,;\emptyset$, replacing the hole $-$ with n in T; likewise $n : \emptyset = \emptyset$.

For each type A we define the set $\mathit{fn}(A)$ of free names as usual, considering n bound in $\mathsf{H}n.A$. In any type $A \triangleright B$ we require $\mathit{fn}(A)\#\mathit{fn}(B)$.

At least superficially, the type language of $\mathbf{T}$ is not far from the spatial logics for concurrency, as presented in [4, 5, 1]. In fact, we may construct an embedding of $\mathbf{T}$ in the spatial logic of [4, 5]. However, we are here interested in a more refined and specific semantics, reflecting the intended safety properties of types. The semantics of types as logical predicates on processes is given by the relation of satisfaction $P \models A$ defined between processes P and formulas A.

Spatial composition $A \mid B$ is interpreted in the standard way, while enforcing free name containment. In the hidden name quantifier $\mathsf{H}n.A$ the name n is bound; this construct is a generic mechanism allowing us to define types with bound names, even if most of such names will be elided away by subtyping. Behavioral modalities are classified along two dimensions: input/output, and spatial/sharing (depending on whether the argument is handled via the spatial (|) or sharing ($\wedge$) conjunction). $\Box$ is useful to express invariants. We omit a full treatment of recursive types, interpreted as greatest fixed point, that will not bring unexpected difficulties. We also assume that in $P \Mapsto R$ every reduction step occurs in a restricted link. For technical reasons, we split restricted names in *links* and *plain* names (a reduction step on an internal link corresponds to a form of β-step). We define the abbreviations:

$$\begin{array}{ll} P{\Downarrow_{\mathit{safe}}} & \triangleq \mathit{All}\ R.\ P \Mapsto R\ \mathit{implies}\ \mathrm{Safe}(R) \\ P{\Mapsto_{\mathit{safe}}}Q & \triangleq P{\Downarrow_{\mathit{safe}}}\,\mathit{and}\ P \Mapsto R \end{array}$$

Definition 4.2 (Satisfaction). *Semantics of types is inductively defined as shown in Figure 2. N.B. We define* $P \models_n A$ *as* $P \models A$ *and* $\mathit{fn}(P) \subseteq \mathit{fn}(A)$*.*

$$
\begin{array}{ll}
P \models \mathsf{F} & \textit{iff False} \\
P \models \emptyset & \textit{iff } P{\Downarrow_{safe}} \\
P \models A \mid B & \textit{iff } P \equiv\ R \mid Q \ \textit{and } R \models_n A \ \textit{and } Q \models_n B \\
P \models A \triangleright B & \textit{iff All } R.\ R \models_n A \ \textit{implies } (\boldsymbol{\nu} fn(A))(R \mid P){\Mapsto_{safe}} \models B \\
P \models A \wedge B & \textit{iff } P \models A \ \textit{and } P \models B \\
P \models \mathsf{H}m.A & \textit{iff } P \equiv (\boldsymbol{\nu} n)Q \ \textit{and } Q \models A\{^m/_n\} \ \textit{and } n\#fn(\mathsf{H}m.A) \\
P \models \Box A & \textit{iff } \mathrm{Safe}(P)\, \textit{and All } \alpha.\ \textit{if } P \xrightarrow{\alpha} Q \ \textit{then } Q \models A \\
P \models x.!(T){\triangleright}A & \textit{iff } P{\Downarrow_{safe}}\, \textit{and if } P \Mapsto R \xrightarrow{\alpha} Q \ \textit{then } \alpha = (\nu)x\langle n\rangle,\ Q \models A \mid n : T \textit{and } n\#A \\
P \models x.?(T){\triangleright}A & \textit{iff } P{\Downarrow_{safe}}\, \textit{and if } P \Mapsto R \xrightarrow{\alpha} Q \ \textit{then } \alpha = x(n),\ Q \models n : T \triangleright A \\
P \models x.!(T)\,;A & \textit{iff } P{\Downarrow_{safe}}\, \textit{and if } P \Mapsto R \xrightarrow{\alpha} Q \ \textit{then } \alpha = (\nu)x\langle n\rangle,\ Q \models A \wedge n : T \textit{and } n\#A \\
P \models x.?(T)\,;A & \textit{iff } P{\Downarrow_{safe}}\, \textit{and if } P \Mapsto R \xrightarrow{\alpha} Q \ \textit{then } \alpha = x(n),\ Q \models n : T \wedge A \\
P \models [x.!(T){\triangleright}]A & \textit{iff } P{\Downarrow_{safe}}\, \textit{and All } n.\ \textit{if } P \Mapsto R \xrightarrow{(\nu)x\langle n\rangle} Q \ \textit{then } Q \models A \mid n : T \ \textit{and } n\#A \\
P \models [x.?(T){\triangleright}]A & \textit{iff } P{\Downarrow_{safe}}\, \textit{and All } n.\ \textit{if } P \Mapsto R \xrightarrow{x(n)} Q \ \textit{then } Q \models n : T \triangleright A \\
P \models [x.!(T)\,;]A & \textit{iff } P{\Downarrow_{safe}}\, \textit{and All } n.\ \textit{if } P \Mapsto R \xrightarrow{(\nu)x\langle n\rangle} Q \ \textit{then } Q \models A \wedge n : T \ \textit{and } n\#A \\
P \models [x.?(T)\,;]A & \textit{iff } P{\Downarrow_{safe}}\, \textit{and All } n.\ \textit{if } P \Mapsto R \xrightarrow{x(n)} Q \ \textit{then } Q \models A \wedge n : T
\end{array}
$$

Fig. 2. Logical semantics of types.

We can now state some fundamental properties of the satisfaction relation.

Lemma 4.3. *Properties of satisfaction.*

1. *Let $P \models A$. If $P \equiv Q$ then $Q \models A$.*
2. *Let $P \models A$. If $P \Mapsto Q$, then $Q \models A$.*
3. *Let $P \models A$. Then $P{\Downarrow_{safe}}$.*
4. *Let $P \models A$. Then $P\{m{\leftrightarrow}n\} \models A\{m{\leftrightarrow}n\}$.*
5. *Let $P \models A$. If $n \notin A$, then $(\boldsymbol{\nu} n)P \models A$.*

Proof. Induction on the structure of type A. ∎

4.1 Type System

The typing rules of our generic type system **T** is based on formal judgments of two forms: typing judgments and subtyping judgments.

$A <: B$ (*Subtyping Judgment*) $\qquad\qquad$ $P :: A \vdash B$ (*Typing Judgement*)

Some formation rules apply. Intuitively, a typing judgment expresses a rely guarantee property, interpreted by the composition adjunct operator of the underlying logic. Thus, we require in any such judgment that $fn(A)\#fn(B)$, and all names in A are links. Moreover, we require the antecedent to be separated, in

(*Void*)

$$\mathbf{0} :: \emptyset \rhd \emptyset$$

(*Out-Left*)

$$\frac{(y\ \mathit{nfc.}) \quad P :: A \mid C \mid y : T \vdash B}{x(y).P :: x :!(T) \rhd A \mid C \vdash B}$$

(*In-Right*)

$$\frac{(y\ \mathit{nfc.}) \quad P :: A \mid y : T \vdash B}{x(y).P :: A \vdash x :?(T) \rhd B}$$

(*In-Left*)

$$\frac{P :: A \mid C \vdash B}{x\langle n\rangle.P :: x :?(T) \rhd A \mid n : T \mid C \vdash B}$$

(*Out-Right*)

$$\frac{P :: A \vdash B}{x\langle n\rangle.P :: A \mid n : T \vdash x :!(T) \rhd B}$$

(*Par*)

$$\frac{P :: A \vdash B \quad Q :: C \vdash D}{(P \mid Q) :: A \mid C \vdash B \mid D}$$

(*Rec*)

$$\frac{(P :: A \vdash \alpha) \quad P :: A \vdash B}{P :: A \vdash \mathtt{rec}\ \alpha.B}$$

(*Sub*)

$$\frac{A \mathrel{\mathtt{<:}} A' \quad P :: A' \vdash B' \quad B' \mathrel{\mathtt{<:}} B}{P :: A \vdash B}$$

(*Seq*)

$$\frac{(\mathit{fn}(B)\ \mathit{nfc.}) \quad P :: A \vdash A' \mid B \quad Q :: B \mid B' \vdash C}{(\boldsymbol{\nu} B)(P \mid Q) :: A \mid B' \vdash A' \mid C}$$

(*Res*)

$$\frac{(n\ \mathit{nfc.},\ n\ \mathit{plain}) \quad P :: A \vdash B}{(\boldsymbol{\nu} n)P :: A \vdash \mathsf{H}n.B}$$

Fig. 3. The Generic Type System **T**.

the sense that for all composition types $C \mid D$ occurring in the A, we must have $\mathit{fn}(C)\#\mathit{fn}(D)$. On the other hand, the right-hand side B is not subject to any special proviso. These constraints will be preserved by all inference axioms and rules, via adequate provisos. Judgments express certain assertions about types and processes. The meaning of such assertions is given by the notion of validity.

Definition 4.4 (Validity). *Validity of judgments is defined as follows.*

$$\begin{array}{ll} \mathit{valid}(P :: A \vdash B) & \triangleq P \models_{\boldsymbol{n}} A \rhd B \\ \mathit{valid}(A \mathrel{\mathtt{<:}} B) & \triangleq \mathit{All}\ P.\ \mathit{if}\ P \models_{\boldsymbol{n}} A\ \mathit{then}\ P \models_{\boldsymbol{n}} B \end{array}$$

A proof system for subtyping is sound if whenever it derives $A \mathrel{\mathtt{<:}} B$, then $\mathit{valid}(A \mathrel{\mathtt{<:}} B)$. Likewise, a proof system for typing is sound if whenever it derives $P :: A \rhd B$ then $\mathit{valid}(P :: A \rhd B)$. An immediate consequence of soundness of typability is that if $P :: \emptyset \rhd \emptyset$ is derivable, then, by Lemma 4.3(2,3), we conclude that for all Q such that $P \Rightarrow Q$ we have Safe(Q).

In Figure 3, we present the rules of the generic type system **T**. A proviso of all rules is that only well-formed judgments may be concluded, and $x \in \Lambda_c$. We abbreviate "x not free in the conclusion" by (x *nfc.*). Notice that typing depends on subtyping just in the (*Sub*) rule. As in any type system, the rules are directed by the syntax of processes (even if we may have more than one rule for each construct). A main result of this paper is then:

Theorem 4.5 (Soundness of Type System *T*). *Let* $\mathtt{<:}$ *be any sound subtyping relation. If* $P :: A \rhd B$ *is derivable in T, then* $\mathit{valid}(P :: A \rhd B)$.

Proof. We show that each rule preserves validity. We start by showing the following fact (induction on A): if A is separated, and $R \models A$ then $R \Rightarrow Q$ implies $R \Mapsto Q$. We consider each rule in turn; it is interesting to look at a few cases.

- (Case of (*Void*)) Pick $P \models_{\mathtt{n}} \emptyset$. Then $P \mid \mathbf{0} \equiv P$, by closure of satisfaction under structural congruence, and we conclude $P \mid \mathbf{0} \models \emptyset$. Thus $\mathbf{0} \models_{\mathtt{n}} \emptyset \triangleright \emptyset$.
- (Case of (*Par*)) Pick R such that $R \models_{\mathtt{n}} A \mid C$. Then $R \equiv R_1 \mid R_2$ where $R_1 \models_{\mathtt{n}} A$ and $R_2 \models_{\mathtt{n}} B$. By the premises, $(\boldsymbol{\nu} A)(R_1 \mid P) \Mapsto_{safe} \models_{\mathtt{n}} B$ and $(\boldsymbol{\nu} C)(R_2 \mid Q) \Mapsto_{safe} \models_{\mathtt{n}} D$. We have $B \# C$ and $A \# D$ and $A \# C$. Hence $(\boldsymbol{\nu} AC)(R \mid P \mid Q) \Mapsto_{safe} \models_{\mathtt{n}} B \mid D$, and so $(P \mid Q) \models_{\mathtt{n}} A \mid C \triangleright B \mid D$.
- (Case of (*Seq*)) Pick any R such that $R \models_{\mathtt{n}} A \mid B'$. So $R \equiv R_1 \mid R_2$ where $R_1 \models_{\mathtt{n}} A$ and $R_2 \models_{\mathtt{n}} B'$. We know that $A \# C$ and $A' \# B'$. By left premise, $(\boldsymbol{\nu} A)(R_1 \mid P) \Mapsto_{safe} T \models_{\mathtt{n}} A' \mid B$. Then $T = T_1 \mid T_2$ where $T_1 \models_{\mathtt{n}} A'$ and $T_2 \models_{\mathtt{n}} B$. Thus $T_2 \mid R_2 \models_{\mathtt{n}} B \mid B'$. By the right premise, we obtain that $(\boldsymbol{\nu} BB')(T_2 \mid R_2 \mid Q) \Mapsto_{safe} \models_{\mathtt{n}} C$, and so $T_1 \mid (\boldsymbol{\nu} BB')(T_2 \mid R_2 \mid Q) \Mapsto_{safe} \models_{\mathtt{n}} A' \mid C$. Then

$$\begin{array}{l} (\boldsymbol{\nu} BB')(T_1 \mid T_2 \mid R_2 \mid Q) \Mapsto_{safe} \models_{\mathtt{n}} A' \mid C \\ (\boldsymbol{\nu} BB')(T \mid R_2 \mid Q) \Mapsto_{safe} \models_{\mathtt{n}} A' \mid C \\ (\boldsymbol{\nu} BB')(\boldsymbol{\nu} A)(R \mid P \mid Q) \Mapsto_{safe} \models_{\mathtt{n}} A' \mid C \end{array}$$

  by Lemma 4.3(1). Since R is arbitrary, $(\boldsymbol{\nu} B)(P \mid Q) \models_{\mathtt{n}} A \mid B' \triangleright A' \mid C$.
- (Case of (*In-Right*)) From $P :: A \mid y : T \vdash B$ we get $x(y).P :: A \vdash x.?(T) \triangleright B$. Pick R such that $R \models_{\mathtt{n}} A$. Let $S \triangleq (\boldsymbol{\nu} A)(R \mid x(y).P)$. Since $x \notin A$, we have $x \notin fn(R)$. If $S \Mapsto\xrightarrow{\alpha} S'$ with a visible ($\neq \tau$) action α, then $S' \equiv (\boldsymbol{\nu} A)(R' \mid P)$ where $R \Mapsto_{safe} R'$ and $\alpha = x(y)$ for some $y \# A, R$. By validity of the premise, we have $P \models_{\mathtt{n}} (A \mid y : T) \triangleright B$. By Lemma 4.3(2), $R' \models_{\mathtt{n}} A$. Pick any $Q \models_{\mathtt{n}} y : T$. Then $(\boldsymbol{\nu} y)(Q \mid S') \Mapsto_{safe} \models_{\mathtt{n}} B$. So $S' \models_{\mathtt{n}} y : T \triangleright B$. Hence $(\boldsymbol{\nu} A)(R \mid x(y).P) \models_{\mathtt{n}} x.?(T) \triangleright B$. We conclude $x(y).P \models_{\mathtt{n}} A \triangleright x.?(T) \triangleright B$. ∎

4.2 Subtyping

We have deliberately left open the definition of any concrete subtyping relation, in order to given a general soundness result for the core system $\mathbf{T}$, independently of any such subtyping relation. However, one expects any interesting subtyping relation to contain at least the deductive closure of the proof system in Figure 4. These principles essentially state the commutative monoidal structure of spatial composition $- \mid -$ with unit $\emptyset$, congruence principles, and (logical) scope extrusion rules for the hidden name quantifier (see [4]). We may then show

Theorem 4.6 (Soundness of Subtyping). *Let A <: B be derivable in $\boldsymbol{T}_{<:}$. Then valid(A <: B).*

Proof. We show that each rule preserves validity of <: judgments. The proof is straightforward for most axioms, using Lemma 4.3(2). For (*HidWeak*), we show (induction on B) that $P \models_{\mathtt{n}} B$ and Safe(Q) implies $P \mid Q \models B$. ∎

$A \mathrel{\texttt{<:>}} A \mid \emptyset$ $(\mathit{ParVoid})$	$A \mid B \mathrel{\texttt{<:>}} B \mid A$ (ParCom)
$A \mid (B \mid C) \mathrel{\texttt{<:>}} (A \mid B) \mid C$ $(\mathit{ParAssoc})$	$A \mathrel{\texttt{<:}} B \Rightarrow A \mid C \mathrel{\texttt{<:}} B \mid C$ $(\mathit{ParCong})$
$\mathsf{H}\lvert A\rvert.(A \mid B) \mathrel{\texttt{<:}} B$ $(\mathit{HidWeak})$	$\mathsf{H}\lvert A\rvert.\emptyset \mathrel{\texttt{<:}} \emptyset$ $(\mathit{HidVoid})$
$A \mid \mathsf{H}n.B \mathrel{\texttt{<:}} \mathsf{H}n.(A \mid B)$ (HidExt)	$A \mathrel{\texttt{<:}} B \Rightarrow \mathsf{H}n.A \mathrel{\texttt{<:}} \mathsf{H}n.B$ $(\mathit{HidCong})$
$A \mathrel{\texttt{<:>}} A \wedge A$ $(\mathit{ConjAdd})$	$A \mathrel{\texttt{<:>}} A \wedge \emptyset$ $(\mathit{ConjVoid})$
$A \wedge B \mathrel{\texttt{<:>}} B \wedge A$ $(\mathit{ConjCom})$	$(A \wedge B) \wedge C \mathrel{\texttt{<:>}} A \wedge (B \wedge C)$ $(\mathit{ConjAssoc})$
$A \mathrel{\texttt{<:}} B \Rightarrow A \wedge C \mathrel{\texttt{<:}} B \wedge C$ $(\mathit{ConjCong})$	$A \mathrel{\texttt{<:}} B \Rightarrow \alpha A \mathrel{\texttt{<:}} \alpha B$ $(\mathit{ActCong})$
$\mathsf{F} \mathrel{\texttt{<:}} A$ (Bot)	$A \mathrel{\texttt{<:}} B \Rightarrow [\alpha]A \mathrel{\texttt{<:}} [\alpha]B$ $(\mathit{ActCong})$
$\emptyset \mathrel{\texttt{<:}} \alpha A$ $(\mathit{ActVoid})$	$\emptyset \mathrel{\texttt{<:}} [\alpha]A$ $(\mathit{ActVoid})$

Fig. 4. Basic Subtyping Axioms and Rules $\mathbf{T}_{\texttt{<:}}$.

4.3 Sharing

The typing rules of the core type system $\mathbf{T}$ presented above do not make special use of conjunctive types. In fact, only "linear" usages of channel names seem to be allowed. We will now show how conjunctive types may be used to type general forms of sharing, and express common properties of type systems, as the ones described in Section 3. We start by defining

Definition 4.7. *A family $\mathcal{F}$ of types is* sharing *if its is closed under conjunction, and satisfies the following contraction conditions relative to the spatial and sharing conjunctions, for any types A, $m : T$, and $n : T$ in $\mathcal{F}$:*

1. *$A \mid A \models_{\boldsymbol{n}} A$.*
2. *If $P \models_{\boldsymbol{n}} A \wedge n : T \wedge m : T$ then $P\{{}^{n}/_{m}\} \models_{\boldsymbol{n}} A \wedge m : T$.*

For example, the simple types of Section 3 are sharing, in face of Lemma 3.5(2,4).

Notice that as far as behavioral type constructors are concerned, the αA types express fairly strong safety properties, while $[\alpha]A$ types are close to the Hennessy-Milner logic operators. For that reason, we will call *classical* those types with no occurrences of spatial ($A \mid B$, $A \triangleright B$) or αA operators. We also call *invariant* any type A such that $A \models \Box A$. We then prove the following (perhaps surprising) result, that shows that spatial and shared properties may be composed for the important class of classical (purely behavioral) types.

Lemma 4.8 (Spatial/Sharing Cut). *Let C by a classical invariant and $R \models A \wedge C$ and $P \models A \triangleright B$, with $\mathit{fn}(A)\#\mathit{fn}(C)$. Then $(\boldsymbol{\nu} A)(P \mid R) \models B \wedge C$.*

Proof. Since $R \models A$, we have $(\boldsymbol{\nu} A)(P \mid R) \models B$. We have $R \models C$. By induction on the type C, we show that $(\boldsymbol{\nu} A)(P \mid R) \models C$. ■

Conjunctive typing rules are depicted in Figure 5, all (*In-S-*) and (*Out-S-*) rules are subject to the proviso that the types in the right-hand-side of the premises (e.g, $B \wedge y : T$), belong to an invariant sharing type family. Essentially, we define a left and right rule for input and output processes, and the "sharing cut" (cf. the (*Seq*) typing rule) motivated by Lemma 4.8. As before, we can state:

$$\overset{(\ \mathit{In\text{-}S\text{-}Right}\ )}{\dfrac{P :: A \vdash B \wedge y : T}{x(y).P :: A \vdash x :?(T) \,;B}}\ (y\ \mathit{nfc.}) \qquad \overset{(\ \mathit{In\text{-}S\text{-}Left}\ )}{\dfrac{P :: A \vdash B \wedge y : T}{x(y).P :: x :!(T) \,; A \vdash B}}\ (y\ \mathit{nfc.})$$

$$\overset{(\ \mathit{Out\text{-}S\text{-}Left}\ )}{\dfrac{P :: A \vdash B \wedge n : T \quad (n \# B)}{x\langle n\rangle.P :: x :?(T) \,; A \vdash B \wedge n : T}} \qquad \overset{(\ \mathit{Out\text{-}S\text{-}Right}\ )}{\dfrac{P :: A \vdash B \wedge n : T \quad (n \# B)}{x\langle n\rangle.P :: A \vdash x :!(T) \,; B \wedge n : T}}$$

$$\overset{(\ \mathit{Sharing\text{-}Cut}\ )}{\dfrac{\begin{array}{c} B \# C \quad C\ \mathit{classical\ invariant} \\ P :: A \vdash B \wedge C \quad Q :: B \vdash D \end{array}}{(\boldsymbol{\nu} B)(P \mid Q) :: A \vdash D \wedge C}}$$

Fig. 5. Sharing typing rules **S**.

$$\overset{(\ \mathit{Out\text{-}P\text{-}Right}\ )}{\dfrac{P :: A \vdash B \mid n : T}{(\boldsymbol{\nu} n)x\langle n\rangle.P :: A \vdash x :!(T) \triangleright B}} \qquad \overset{(\ \mathit{Out\text{-}P\text{-}Left}\ )}{\dfrac{P :: A \vdash B \mid n : T}{(\boldsymbol{\nu} n)x\langle n\rangle.P :: x :?(T) \triangleright \emptyset \mid A \vdash B}}$$

Fig. 6. Bound output typing rules **P**.

Proposition 4.9. *The sharing typing rules* **S** *are sound.*

Proof. We show that each rule preserves validity. We show here a few cases.

1. (Case of (*Sharing-Cut*)) By Lemma 4.8(1).
2. (Case of (*In-S-Left*)) We have $x(y).P :: x.!(T)\,; A \vdash B$ derived from $P :: A \vdash y : T \wedge B$, where $x \# A$. Pick R such that $R \models x.!(T)\,; A$. Let $S \triangleq (\boldsymbol{\nu} Ax)(R \mid x(y).P)$. Consider any reduction from S: it has the form $S \Rightarrow S'$ with $S' \equiv (\boldsymbol{\nu} Ax\overline{s})(R' \mid P\{{}^{y}/_{n}\})$ where $R {\mapsto\!\!\!\!\Rightarrow}_{\mathit{safe}} \xrightarrow{\alpha} R'$ and $\alpha = (\boldsymbol{\nu}\overline{s})x\langle n\rangle$ and $R' \models A \wedge n : T$ with $n \# A$. By Lemma 4.8(1), $(\boldsymbol{\nu} A)(R' \mid P) \models n : T \wedge B \wedge y : T$. Since $B \wedge y : T$ is sharing, $(\boldsymbol{\nu} A)(R' \mid P\{{}^{y}/_{n}\}) \models n : T \wedge B$. We conclude $x(y).P :: x.!(T)\,; A \vdash B$.
3. (Case of (*Out-S-Right*)) We have $x\langle n\rangle.P :: A \vdash x.!(T)\,; B \wedge n : T$ concluded from $P :: A \vdash B \wedge n : T$. Pick $R \models A$, and let $S \triangleq (\boldsymbol{\nu} A)(R \mid x\langle n\rangle.P)$.
Since $n, x \# A$, if $S \mapsto\!\!\!\!\Rightarrow \xrightarrow{\alpha} S'$ with a visible action α, then $S' \equiv (\boldsymbol{\nu} A)(R' \mid P)$, $R {\mapsto\!\!\!\!\Rightarrow}_{\mathit{safe}} R'$, and $\alpha = x\langle n\rangle$. Then $R' \models_{\mathtt{n}} A$. By the premise, we have $P \models_{\mathtt{n}} A \triangleright B \wedge n : T$, and thus $S' \models B \wedge n : T$ with $n \# B$. Since R is arbitrary, we have $x\langle n\rangle.P \models A \triangleright x :!(T)\,; B$. Then $x\langle n\rangle.P \models_{\mathtt{n}} A \triangleright x.!(T)\,; B \wedge n : T$. ∎

4.4 Additional Typing Rules

We may still consider additional typing and subtyping rules. For example, reasoning by symmetry, it would seem sensible to consider a variation of the output rules (*Out-Right*) and (*Out-Left*) of **T**, where the source of the output is obtained from the continuation of the output process, rather than from the parallel spatial context. It is interesting to notice that this pattern of resource hand-over seems associated to bound output. We illustrate this development by introducing the post output **P** rules depicted in Figure 6. These rules do not seem derivable from the ones already presented. In any case, we are not concerned here with finding a minimal (in some well defined sense) set of rules, but rather to illustrate the modularity and flexibility of the approach. We have

Proposition 4.10. *The bound output typing rules* **P** *are sound.*

5 Some Instances of Typing and Subtyping

In previous sections, we have motivated and developed the generic type system T, and prove its soundness using a semantic technique based on a logical interpretation of types. In this section, we discuss the expressiveness of our framework, by showing how some type systems of well-known kind, namely, simple types, I/O types, and a form of session types, may be embedded in a fairly direct way in the type system **T**, just by choosing suitable additional subtyping axioms.

5.1 Simple Types

It is instructive to elaborate a representation of the simple type system of Section 3 in the general type system **T**. Essentially, we need to express typing interpretations (Definition 3.2) using our type primitives. We set

$$\begin{array}{ll} \lceil \texttt{nil} \rceil(n) & \triangleq \texttt{rec}\ X.\emptyset \wedge [n.!(\emptyset)\,;]\mathsf{F} \wedge [n.?(\emptyset)\,;]\mathsf{F} \wedge \Box X \\ \lceil (T) \rceil(n) & \triangleq \texttt{rec}\ X.\emptyset \wedge [n.!(\lceil T \rceil)\,;]X \wedge [n.?(\lceil T \rceil)\,;]X \wedge \Box X \\ \lceil\ \rceil & \triangleq \emptyset \\ \lceil n : T, \Gamma \rceil & \triangleq \lceil T \rceil(n) \wedge \lceil \Gamma \rceil \end{array}$$

Notice that the translation of a typing context Γ essentially just spells out, fairly directly, the coinductive definition of $[\![\Gamma]\!]$ of Definition 3.2. It is then not difficult to check that $P \models_{\mathtt{n}} \lceil \Gamma \rceil$ if and only if $P \models_{\mathtt{s}} \Gamma$. We will then deliberately mix (the syntax of) simple types (U, V, T) with general types, with the assumption that the former are seen as the abbreviations defined above. We may then show:

Proposition 5.1. *The subtyping judgments* $\mathbf{ST}_{\texttt{<:}}$ *, listed below, are valid:*

$$\begin{array}{ll} \emptyset \texttt{ <: } \Gamma & (\mathit{Weaken}) \\ n.?(T)\,;(\Gamma \wedge n : (T)) \texttt{ <: } \Gamma \wedge n : (T) & (\mathit{ContrInp}) \\ n.!(T)\,;(\Gamma \wedge n : (T)) \texttt{ <: } \Gamma \wedge n : (T) & (\mathit{ContrOut}) \\ \Gamma \mid \Gamma \texttt{ <: } \Gamma & (\mathit{ContrPar}) \\ \mathsf{H}n.(\Gamma \wedge n : T) \texttt{ <: } \Gamma & (\mathit{ContrRes}) \end{array}$$

Proof. Verification is direct for (*Weaken*). We show (*ContrOut*) this in detail, for (*ContrInp*) is similar. Let $P \models_{\mathtt{n}} n.!(T)\,;(\Gamma \wedge n : (T))$, and $P \xrightarrow{\alpha} Q$. By the assumption, either $\alpha = \tau$ and $Q \models n.!(T)\,;(\Gamma \wedge n : (T))$, or $\alpha = (\boldsymbol{\nu}\overline{s})n\langle m \rangle$ and $R \models \Gamma \wedge n : (T) \wedge m : T$ with $m \# \Gamma, n$. By coinduction, we conclude $P \models \Gamma \wedge n : (T)$. (*ContrPar*) and (*ContrRes*) are valid by Lemma 3.5(4,5). ■

The laws presented in Proposition 5.1 express weakening and contraction principles that confirm the “exponential” (sharing) character of simple types.

Notice that these principles are justified by semantics entailments, independently of any proof theoretic considerations. We can also verify that simple types are classical, and sharing (Definition 4.7), as a consequence of Lemma 3.5(4,5). If one considers these contraction principles in (the subtyping relation of) the general type system $\mathbf{T}$, obtaining the system $\mathbf{T} + \mathbf{ST}_{\texttt{<:}}$, then we may show that each rule of the Simple type system (in Figure 1) becomes admissible.

Proposition 5.2. *The Simple type system is admissible in* $\mathbf{T} + \mathbf{ST}_{\texttt{<:}}$ *.*

Proof. Each judgment $\Gamma \vdash P$ is represented by $P :: \emptyset \vdash \lceil \Gamma \rceil$ in $\mathbf{T}$. Then (*ST-Void*) is admissible by (*Void*) and subtyping by (*Weakening*). (*ST-Par*) is admissible by (*Par*) and subtyping by (*ContrPar*). (*ST-Res*) is admissible by (*Res*) and subtyping by (*ContrRes*). (*ST-Out*) is represented as follows:

$$\begin{array}{lll}
1. & \Gamma, n : U, m : (U) \vdash P & P :: \emptyset \vdash \lceil \Gamma \rceil \wedge \lceil n : U \rceil \wedge \lceil m : (U) \rceil \\
2. & & P :: \emptyset \vdash \lceil \Gamma \rceil \wedge \lceil U \rceil(n) \wedge \lceil m : (U) \rceil \\
3. & & m\langle n \rangle.P :: \emptyset \vdash m.!(\lceil U \rceil)\,;(\lceil \Gamma \rceil \wedge \lceil m : (U) \rceil) \wedge \lceil U \rceil(n) \\
4. & \Gamma, m : (U) \vdash m\langle n \rangle.P & m\langle n \rangle.P :: \emptyset \vdash \lceil \Gamma \rceil \wedge \lceil m : (U) \rceil \wedge \lceil n : U \rceil
\end{array}$$

So, (2,3) by (*Out-S-Right*) and (3,4) (*Sub*) (by (*ContrOut*)). ■

5.2 I/O Types

We show how I/O types, along the lines of [19], may be represented in the type system $\mathbf{T}$. I/O types are similar to simple types, but now channel types are tagged with a mode $\mu \in \{+, -, \pm\}$. Standard channel types (U), now written $(U)^{\pm}$, are refined into input only $(U)^{-}$ and output only $(U)^{+}$ channel types. A logical semantics of I/O types may be given as follows.

$$\begin{array}{ll}
\lceil (T)^{\pm} \rceil(n) & \triangleq \texttt{rec}\ X.\emptyset \wedge [n.!(\lceil T \rceil^{\circ})\,;]X \wedge [n.?(\lceil T \rceil)\,;]X \wedge \Box X \\
\lceil (T)^{+} \rceil(n) & \triangleq \texttt{rec}\ X.\emptyset \wedge [n.!(\lceil T \rceil^{\circ})\,;]X \wedge [n.?(\emptyset)\,;]\mathsf{F} \wedge \Box X \\
\lceil (T)^{-} \rceil(n) & \triangleq \texttt{rec}\ X.\emptyset \wedge [n.!(\emptyset)\,;]\mathsf{F} \wedge [n, ?(\lceil T \rceil)\,;]X \wedge \Box X \\
\lceil (T)^{\pm} \rceil^{\circ}(n) & = \texttt{rec}\ X.\emptyset \wedge [n.!(\lceil T \rceil^{\circ})\,;]X \wedge [n.?(\lceil T \rceil^{\circ})\,;]X \wedge \Box X \\
\lceil (T)^{+} \rceil^{\circ}(n) & = \lceil (T)^{-} \rceil^{\circ}(n) \triangleq \lceil (T)^{\pm} \rceil(n)
\end{array}$$

Again, we notice that the translation above offers a fairly direct specification of the semantics of I/O types, and that these are introduced as an orthogonal (conservative) extension of simple types. Indeed, we can check that if T is a type containing just the $(-)^{\pm}$ type constructor, and U is the simple type "erasure" of T then $[\![T]\!] = [\![U]\!]$. We may also verify that all the subtyping principles stated in Proposition 5.1 remain valid for I/O types. Moreover, we have

Proposition 5.3. *The subtyping rules* $\mathbf{IO}_{\texttt{<:}}$ *are valid for any I/O types* U, T*:*

$$\begin{array}{lll}
n : (T)^{+} \texttt{ <: } n : (T)^{\pm} & & (\mathit{InpIO}) \\
n : (T)^{-} \texttt{ <: } n : (T)^{\pm} & & (\mathit{OutIO}) \\
U \texttt{ <: } T \Rightarrow n.?(U)\,;(\Gamma \wedge n : (T)^{\mu}) \texttt{ <: } \Gamma \wedge n : (T)^{\mu} & (- \in \mu) & (\mathit{ContrIOInp}) \\
T \texttt{ <: } U \Rightarrow n.!(U)\,;(\Gamma \wedge n : (T)^{\mu}) \texttt{ <: } \Gamma \wedge n : (T)^{\mu} & (+ \in \mu) & (\mathit{ContrIOOut})
\end{array}$$

$$\overset{(\ \mathit{SubInp}\ )}{\frac{n : T \texttt{ <: } n : U}{n : (T)^{-} \texttt{ <: } n : (U)^{-}}} \quad \overset{(\ \mathit{SubOut}\ )}{\frac{n : U \texttt{ <: } n : T}{n : (T)^{+} \texttt{ <: } n : (U)^{+}}} \quad \overset{(\ \mathit{SubIO}\ )}{\frac{n : U \texttt{ <:> } n : T}{n : (T)^{\pm} \texttt{ <: } n : (U)^{\pm}}}$$

$$
\begin{array}{l}
P :: \emptyset \vdash a : ((T)^-)^\pm \wedge b : ((T)^+)^\pm \wedge p : (T)^\pm \wedge c : (T)^- \\
a(c).P :: \emptyset \vdash a.?((T)^-)\,;(a : ((T)^-)^\pm \wedge b : ((T)^+)^\pm \wedge p : (T)^\pm) \\
P_s :: \emptyset \vdash a : ((T)^-)^\pm \wedge b : ((T)^+)^\pm \wedge p : (T)^\pm \quad \text{by } (\mathit{ContrIOInp}) \\[1ex]
C_i :: \emptyset \vdash a : ((T)^-)^\pm \wedge b : ((T)^+)^\pm \wedge p : (T)^+ \wedge c : (T)^+ \\
b(c).C_i :: \emptyset \vdash b.?((T)^+)\,;(a : ((T)^-)^\pm \wedge b : ((T)^+)^\pm \wedge p : (T)^\pm) \\
S_i :: \emptyset \vdash a : ((T)^-)^\pm \wedge b : ((T)^+)^\pm \wedge p : (T)^\pm \quad \text{by } (\mathit{ContrIOInp}) \\[1ex]
\mathbf{0} :: \emptyset \vdash a : ((T)^-)^\pm \wedge b : ((T)^+)^\pm \wedge p : (T)^\pm \\
b\langle p\rangle :: \emptyset \vdash b.!((T)^\pm)\,;(a : ((T)^-)^\pm \wedge b : ((T)^+)^\pm \wedge p : (T)^\pm) \\
b\langle p\rangle :: \emptyset \vdash a : ((T)^-)^\pm \wedge b : ((T)^+)^\pm \wedge p : (T)^\pm \quad \mathit{by}\ (\mathit{ContrIOOut}) \\
b\langle p\rangle.b\langle p\rangle :: \emptyset \vdash a : ((T)^-)^\pm \wedge b : ((T)^+)^\pm \wedge p : (T)^\pm \quad \mathit{Identical} \\
a\langle p\rangle.b\langle p\rangle.b\langle p\rangle :: \emptyset \vdash a.!((T)^\pm)\,;(a : ((T)^-)^\pm \wedge b : ((T)^+)^\pm \wedge p : (T)^\pm) \\
a\langle p\rangle.b\langle p\rangle.b\langle p\rangle :: \emptyset \vdash a : ((T)^-)^\pm \wedge b : ((T)^+)^\pm \wedge p : (T)^\pm \quad \mathit{by}\ (\mathit{ContrIOOut}) \\
(P_s \mid S_1 \mid S_2 \mid I) :: \emptyset \vdash a : ((T)^-)^\pm \wedge b : ((T)^+)^\pm \wedge p : (T)^\pm \\
\mathit{Sys} :: \emptyset \vdash a : ((T)^-)^\pm \wedge b : ((T)^+)^\pm
\end{array}
$$

Fig. 7. Sample derivation of I/O types.

Proof. Immediate for (*InpIO*), (*OutIO*) and (*ContrIOInp*), and (*SubInp*). For the remaining ones we first show that (a) $\lceil T\rceil \models \lceil T\rceil^\circ$, and (b) $U \mathrel{\mathtt{<:}} V$ implies $\lceil V\rceil^\circ \models \lceil U\rceil^\circ$. (*ContrIOOut*) follows from (a) and (b), and (*SubOut*) from (b). ∎

Interestingly, the subtyping relation induced by the logical semantics satisfy the syntactically defined relation $\leq$ in [19] (reading $\geq$ as $\mathtt{<:}$, and apart from recursive types, for which one should add a coinduction rule). All of its typing rules may be shown admissible in the extension of $\mathbf{T}+\mathbf{IO}_{\mathtt{<:}}$: we just need to verify that I/O types are classical and invariant (by inspection), sharing, and therefore that all the sharing typing rules **S** rules are applicable to them.

For an illustration, we borrow an example from [19]. A system composed by a printer P and two clients C_1 and C_2 is set up so that the printer is only allowed to read from the clients, while clients are only allowed to write to the printer. For readability, we tag bound names with their intended types.

$$
\begin{array}{lll}
\mathit{Sys} \triangleq (\boldsymbol{\nu} p : (T)^\pm)(P_s \mid S_1 \mid S_2 \mid I) & \qquad & I \triangleq a\langle p\rangle.b\langle p\rangle.b\langle p\rangle \\
P_s \triangleq a(c : (T)^-).P & & S_i \triangleq b(c : (T)^+).C_i
\end{array}
$$

We can then derive $\mathit{Sys} :: \emptyset \vdash a : ((T)^-)^\pm \wedge b : ((T)^+)^\pm$, as presented in the Figure 7. Interpreting the types as the intended logical predicates, by soundness, we conclude, for instance, that $P \models \mathtt{rec}\ X.[n.!(\emptyset)\,;]\mathsf{F} \wedge \Box X$. This means that the printer will never attempt to write on channel c.

5.3 Behavioral and Session Types

Various behavioral type disciplines for π-calculi have been proposed (*e.g.*, [13, 14, 10]), the intention being to discipline the sequence of interactions between

$$\mathit{Session}(x) \triangleq x.!(\mathit{Op}) \triangleright x.!(\mathit{Int}) \triangleright x.!(\mathit{Int}) \triangleright x.?(\mathit{Int}) \triangleright \emptyset$$

$$\begin{array}{l}
\mathbf{0} :: \emptyset \vdash \emptyset \\
x(u).\mathbf{0} :: \emptyset \vdash x.!(\mathit{Int}) \triangleright \emptyset \\
x\langle 1\rangle.x\langle 2\rangle.x(u).\mathbf{0} :: \emptyset \vdash x.!(\mathit{Int}) \triangleright x.!(\mathit{Int}) \triangleright x.?(\mathit{Int}) \triangleright \emptyset \\
\mathit{ClientBody}(x) :: \emptyset \vdash \mathit{Session}(x) \\
(\boldsymbol{\nu} x)(a\langle x\rangle.\mathit{ClientBody}(x)) :: a.?(\mathit{Session}) \triangleright \emptyset \vdash \emptyset \quad \mathit{by}\ (\mathit{Out\text{-}P\text{-}Left})
\end{array}$$

$$\begin{array}{l}
\mathbf{0} :: \emptyset \vdash \emptyset \\
y\langle v_1 + v_2\rangle.\mathbf{0} :: \mathit{op} : \mathit{Op} \mid v_1 : \mathit{Int} \mid v_2 : \mathit{Int} \mid y.?(\mathit{Int}) \triangleright \emptyset \vdash \emptyset \\
y(v_1).y(v_2).y\langle v_1 + v_2\rangle.\mathbf{0} :: \mathit{op} : \mathit{Op} \mid y.!(\mathit{Int}) \triangleright y.!(\mathit{Int}) \triangleright y.?(\mathit{Int}) \triangleright \emptyset \vdash \emptyset \\
\mathit{ServerBody}(y) :: \mathit{Session}(y) \vdash \emptyset \\
a(y).\mathit{ServerBody}(y) :: \emptyset \vdash a :?(\mathit{Session}) \triangleright \emptyset \quad \mathit{by}\ (\mathit{In\text{-}Right})
\end{array}$$

$$\mathit{Sys} :: \emptyset \vdash \emptyset \quad \mathit{by}\ (\mathit{Seq})$$

Fig. 8. Sample derivation of session types.

processes, so that certain liveness and safety properties may be obtained. Particularly interesting are session types [12], that may be used to discipline dialogue-like interactions between exactly two parties. At least certain forms of session types may be embedded in the generic type system $\mathbf{T}$ in a rather straightforward way, by combining behavioral types with simple types. The basic idea is to use judgments of the form $P :: S_i \vdash S_o \wedge \Gamma$ where S_i represents the (session) types of incoming (from the process environment) sessions, S_o the (session) types of outgoing (to the process environment) sessions, and Γ, a sharing type, declares the types of shared channels. Usually, one would expect Γ to be a conjunction of sharing types, for instance, simple types, or I/O types. On the other hand, the types S_i and S_o may be quite arbitrary, as far as one ensures $\mathit{fn}(\Gamma)\#\mathit{fn}(S_o)$ (need to combine processes using (*Shared-Cut*)). We illustrate with a simple example [10]: a server that offers a integer addition service, and its client.

$$\begin{array}{ll}
\mathit{Sys} & \triangleq (\boldsymbol{\nu} a)(\mathit{Client} \mid \mathit{Server}) \\
\mathit{ClientBody}(x) & \triangleq x\langle \mathit{plus}\rangle.x\langle 1\rangle.x\langle 2\rangle.x(u).\mathbf{0} \\
\mathit{ServerBody}(x) & \triangleq x(\mathit{op}).x(v_1).x(v_2).x\langle v_1 + v_2\rangle.\mathbf{0} \\
\mathit{Client} & \triangleq (\boldsymbol{\nu} x)(a\langle x\rangle.\mathit{ClientBody}(x)) \\
\mathit{Server} & \triangleq a(y).\mathit{ServerBody}(y)
\end{array}$$

A possible typing for the system *Sys* in $\mathbf{T} + \mathbf{ST}_{<:}$ is shown in Figure 8, where we assume the extension of the system with some pure value types (*Int*, *Op*), along predictable lines (cf. `nil`). Notice that no sharing types have been used, and channel a is used just once. However, channel a may be shared, even if it moves around "session" partners as resources, using the spatial modalities $a.?(\mathit{Session})\triangleright$ and $a.!(\mathit{Session})\triangleright$. However, we may set $a : (\mathit{Session})\triangleright$, where

$$\lceil (T)\triangleright \rceil(n) \triangleq \mathtt{rec}\ X.\emptyset \wedge [n.!(\lceil T\rceil)\triangleright]X \wedge [n.?(\lceil T\rceil)\triangleright]X \wedge \Box X$$

The intention is to let $(T)\triangleright$ be a "ownership-transfer" version of the simple type (T). We may check that the types $(T)\triangleright$ are classical, invariant, and sharing. We denote by **OT** the expected subtyping axioms for "ownership-transfer" simple types. Using just the axioms and rules of $\mathbf{T}+\mathbf{ST}_{<:}+\mathbf{OT}_{<:}$ we may then show an alternative typing for the system *Client* | *Server*, where the name a is free.

$$\mathit{Client} \mid \mathit{Server} :: \emptyset \vdash a : (\mathit{Session})\triangleright$$

Again, soundness of the obtained type system is obtained for "free", after one proves certain abstract properties (*e.g.*, sharing) of new type constructions.

6 Concluding Remarks

The original understanding of types as predicates has not always been a guiding principle in the design of types for process calculi, where a syntactical view seems to be dominant (an exception is [8], where a notion of semantic subtyping for names was developed). In this paper, we have developed a formal semantic approach to types in concurrency, based on an interpretation of types as spatial logic definable properties. The feasibility of the approach was demonstrated by the proposal of a generic type system, where many interesting notions of typing for mobile processes may be embedded just by introducing suitable subtyping relations, while modularly preserving soundness (Theorems 4.5 and 4.6). Thus, our approach seems to generalize other existing proposals to generic typing [13], that rely on more standard (syntactical) techniques. Some of the logical characterizations we have introduced allowed us to understand notions such as sharing and linearity [14] in types for concurrency in a rather abstract setting; it would be interesting to compare ours with other interpretations of sharing [17].

The framework proposed here may be generalized along several directions. Our development is not dependent on the structure of the underlying basic safety predicate, it would then be interesting to consider different basic properties (*e.g.*, security). Different notions of sharing might also be accommodated, if replication replaces recursion in the process calculus.

We believe that spatial logics provide a suitable metalanguage in which many type-like properties of interest may be formally expressed at an adequate level of abstraction, and that soundness proofs developed along the lines we have shown here are more modular and more intuitive than purely syntactic subject reduction style proofs. The representation of a type is essentially a process predicate that explicitly affirms of the subject the safety properties of interest. Our results suggest that these techniques may be used with some advantage over purely syntactic approaches to the semantics of typing, at least in some situations, in particular when traditional subject reduction techniques do not scale so to comfortably handle an increased complexity in global proof invariants, for example, due to the introduction of rich subtyping relations [3].

References

1. L. Caires. Behavioral and Spatial Properties in a Logic for the Pi-Calculus. In I. Walukiewicz, editor, *Foundations of Software Science and Computation Structures*, number 2987 in Lecture Notes in Computer Science. Springer Verlag, 2004.

2. L. Caires. Logical Semantics of Types for Concurrency. Technical Report 2/07, Departamento de Informatica FCT/UNL, 2007.
3. L. Caires. Spatial-Behavioral Types, Distributed Services, and Resources. In U. Montanari and D. Sanella, editors, *TGC 2006 2dn Intl. Symp. on Trustworthy Global Computing*, Lecture Notes in Computer Science. Springer-Verlag, 2007.
4. L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part I). *Information and Computation*, 186(2):194–235, 2003.
5. L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part II). *Theoretical Computer Science*, 3(322):517–565, 2004.
6. L. Caires and H. Vieira. Extensionality of Spatial Observations in Distributed Systems. *Electronic Notes in Theoretical Computer Science*, 2007.
7. L. Cardelli and A. D. Gordon. Anytime, Anywhere. Modal Logics for Mobile Ambients. In *27th ACM Symp. on Principles of Programming Languages*, pages 365–377. ACM, 2000.
8. G. Castagna, R. De Nicola, and D. Varacca. Semantic Subtyping for the π-Calculus. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 92–101. IEEE Computer Society, 2005.
9. H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.
10. S. J. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2-3):191–225, 2005.
11. M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *JACM*, 32(1):137–161, 1985.
12. K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In C. Hankin, editor, *7th European Symposium on Programming*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer-Verlag, 1998.
13. A. Igarashi and N. Kobayashi. A Generic Type System for the Pi-Calculus. In *POPL 2001: 28th ACM Symp. on Principles of Programming Languages*, 2001.
14. N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the Pi-Calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999.
15. R. Milner. The Polyadic π-Calculus: A Tutorial. Technical Report 180, University of Edinburgh LFCS, 1991.
16. R. Milner, J. Parrow, and D. Walker. Modal Logics for Mobile Processes. *Theoretical Computer Science*, 114:149–171, 1993.
17. P. O'Hearn and D. Pym. The Logic of Bunched Implications. *The Bulletin of Symbolic Logic*, 5(2):215–243, 1999.
18. P. W. O'Hearn. Resources, Concurrency and Local Reasoning. In P. Gardner and N. Yoshida, editors, *Concur 2004 15th Intl. Conf. on Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 49–67. Springer-Verlag, 2004.
19. B. C. Pierce and D. Sangiorgi. Typing and Subtyping for Mobile Processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996.
20. J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Third Annual Symposium on Logic in Computer Science*, Copenhagen, Denmark, 2002. IEEE Computer Society.
21. D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
22. W. Tait. Intensional Interpretations of Functionals of Finite Type. *J. Symbolic Logic*, 32(2):198–212, 1967.
23. A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Inf. Comput.*, 115(1):38–94, 1994.