Typing and Subtyping for Mobile Processes

Benjamin Pierce* Davide Sangiorgi†

May 10, 1994

Abstract

The π-calculus is a process algebra that supports process mobility by focusing on the communication of channels. Milner's presentation of the π-calculus includes a type system assigning arities to channels and enforcing a corresponding discipline in their use. We extend Milner's language of types by distinguishing between the ability to read from a channel, the ability to write to a channel, and the ability both to read and to write. This refinement gives rise to a natural subtype relation similar to those studied in typed λ-calculi.

The greater precision of our type discipline yields stronger versions of standard theorems about the π-calculus. These can be used, for example, to obtain the validity of β-reduction for the more efficient of Milner's encodings of the call-by-value λ-calculus, which fails in the ordinary π-calculus.

We define the syntax, typing, subtyping, and operational semantics of our calculus, prove that the typing rules are sound, apply the system to Milner's λ-calculus encodings, and sketch extensions to higher-order process calculi and polymorphic typing.

1 Introduction

Milner, Parrow, and Walker's π-calculus [MPW92] achieved a remarkable simplification and generalization of its predecessors (including CCS [Mil89] and ECCS [EN86]) by focusing on *naming* and allowing the data values communicated along channels to be channels themselves. The calculus can be presented with a handful of rules, but is sufficiently expressive to describe concurrent systems in which the topology of communication may evolve dynamically. Moreover, the existence of natural embeddings of both lazy and call-by-value λ-calculi into the π-calculus [Mil90] suggests that it may form an appropriate foundation for the design of new programming languages.

Milner extended the original π-calculus to a *polyadic π-calculus*, in which finite tuples of names, instead of single names, are the atomic unit of communication [Mil91]. The basic theory of the polyadic π-calculus straightforwardly generalizes that of the monadic fragment; furthermore, the fact that a tuple of names is exchanged at each communication step suggests a natural discipline of *channel sorts*.[1] For example, the expression

$$P_t = b(t, f) \,.\, \bar{t}\langle\rangle \,.\, P_t$$

denotes a process that accepts a pair of names, t and f, from a channel b, responds by sending a null tuple along t, and returns to its original state. Similarly,

$$P_f = b(t, f) \,.\, \bar{f}\langle\rangle \,.\, P_f$$

*Department of Computer Science, University of Edinburgh, The King's Buildings, Edinburgh, EH9 3JZ, U.K. Electronic mail: bcp@dcs.ed.ac.uk.

†Same postal address; electronic mail: sad@dcs.ed.ac.uk

[1] In the λ-calculus literature, the word "type" is standard; for process calculi, "sort" is more common. We use the two terms interchangeably.

accepts t and f and sends a null tuple along f. These processes can be used as encodings of the boolean values *true* and *false*: to test whether a process waiting for input on b is P_t or P_f, it suffices to put it in parallel with the process

$$test = (\boldsymbol{\nu}\, x, y)\ \ (\overline{b}\langle x, y\rangle \,.\, \mathbf{0} \mid x() \,.\, Q_t \mid y() \,.\, Q_f)$$

which creates a pair of channels with the fresh names x and y, sends them along b, and waits for a response along x or y before continuing with Q_t or Q_f. (Here $\mathbf{0}$ is the inactive process and $\boldsymbol{\nu}$ the restriction operator.) When *test* is run in parallel with P_t, the branch Q_t is activated; when in parallel with P_f, on the other hand, Q_f is activated.

Milner observed that this trio of processes obey a strict discipline in their use of names for communication: t, f, x, and y are all used to communicate tuples of arity zero, whereas b is always used to communicate pairs of names that are themselves used to communicate empty tuples. This situation can be described with the following *sorting*:

$$\begin{array}{lcl} t, f, x, y & : & S_e \\ b & : & S_p \\ S_e & \mapsto & () \\ S_p & \mapsto & (S_e, S_e) \end{array}$$

A key idea here is that sort information is assigned only to channels; processes are either well-sorted under a particular set of assumptions for their bound and free names, or they are not. This stands in contrast to recent proposals that attempt to assign more informative types to processes, describing the sets of channels on which processes may communicate, their interaction protocols, freedom from deadlock, etc. Also, this sort information is completely static: it does not describe the *sequencing* of values communicated along a channel. (For example, it cannot describe a channel that is used to carry an alternating sequence of two- and three-element tuples.) Although it may be possible to develop sorting disciplines capturing more dynamic constraints on the behavior of processes, such constraints seem difficult to express while maintaining the essential characteristics of traditional type systems — the methodological perspective that types are best used to describe structural properties of the data manipulated by programs, as well as the more pragmatic requirement that well-typedness must be automatically and efficiently verifiable.

Milner's sort discipline plays an essential role in later papers on properties of the π-calculus [Wal92, San92]; it has been further studied by Turner [Tur94], Gay [Gay93], and Vasconcelos and Honda [VH93], who consider the problem of inferring most general sortings.

Our typing discipline retains the basic character of Milner's, while extending it in two dimensions. First, we replace by-name matching of sorts with the more straightforward notion of structural matching, a technical modification that allows a substantially more elegant presentation of our system of sorts.[2] Second, and more interestingly, we refine the language of sorts to include a notion of *subsort* allowing the use of a channel to be restricted to input-only or output-only in a given context.

In Milner's π-calculus, two sorts with different names are considered to be different, so that the process

$$\overline{a}\langle x\rangle \,.\, \mathbf{0}$$

is ill formed under the sorting

$$\begin{array}{lcl} a & : & S \\ x & : & S_2 \\ S & \mapsto & (S_1) \\ S_1 & \mapsto & () \\ S_2 & \mapsto & () \end{array}$$

even though it is well formed under the isomorphic sorting

$$\begin{array}{lcl} a & : & S \\ x & : & S_1 \\ S & \mapsto & (S_1) \\ S_1 & \mapsto & () \\ S_2 & \mapsto & (). \end{array}$$

[2] A by-structure presentation of Milner's π-calculus sorting has been studied independently by Turner [Tur94]. His system is essentially identical to the fragment of ours in which subtyping is omitted.

By contrast, in our system only the recursive structure of sorts is significant.

Although the change from by-name to by-structure matching entails some loss of precision in the sortings (we give up the ability to prevent mixing of "accidentally isomorphic" data structures) it substantially simplifies the technical aspects of the system by allowing sorts to be treated as regular trees [Cou83].

Our subsort relation can be motivated by the common situation in which two processes must cooperate in the use of a shared resource such as a printer. The printer provides a request channel p carrying values of some sort T, which represent data sent for printing by the client processes. If one client process has the form

$$C_1 = \overline{p}\langle j_1\rangle \,.\, \overline{p}\langle j_2\rangle \,.\, \ldots,$$

then we expect that executing the program

$$(\boldsymbol{\nu}\, p{:}(T)) \;\; (P \mid C_1 \mid C_2)$$

should result in the print jobs represented by j_1 and j_2 eventually being received and processed, in that order, by the printer process P. But this is not necessarily the case: a misbehaving implementation of C_2 can disrupt the protocol expected by P and C_1 just by reading print requests from p and, for example, throwing them away:

$$C_2 = p(j{:}T) \,.\, C_2.$$

We can prevent such bad behavior by distinguishing three kinds of access to a channel — the ability to send values, the ability to read values, and the ability to do both — and extending channel sorts with a tag declaring which operations are allowed: r for input, w for output, or b for both. Here, for example, the client processes should only be allowed to write to p; from their point of view, p's sort is $(T)^{\mathsf{w}}$. The printer, on the other hand, should only read from p; from its point of view, p has sort $(T)^{\mathsf{r}}$. Such an approach is reminiscent of the restriction in the visibility of local state achieved in sequential languages using notions of *data encapsulation*: the appropriate behavior of each component of the system can be ensured statically by typechecking it in an environment where the sort of p allows only the appropriate form of access. More generally, we can define a relation $\leq$ on sorts and stipulate that an output operation $\overline{x}\langle y\rangle$ is well sorted only if $S_y \leq S_x$, where $y{:}S_y$ and $x{:}(S_x)^I$, with I either b or w. Similarly, an input operation $x(y{:}T)$ is well sorted if $S_x \leq T$, where $x{:}(S_x)^I$ and I is either b or r.

The subsort relation is formalized in Section 2 using a notion of *simulation* of regular trees and treated using techniques similar to those developed by Amadio and Cardelli for λ-calculi with subtyping and recursive types [AC91]. Section 3 proves the soundness of the resulting type system. The greater precision of our type discipline yields stronger versions of some standard theorems about the π-calculus. For example, in Sections 4 and 5 we obtain the validity of β-reduction for the more efficient of Milner's encodings of the call-by-value λ-calculus (which does not hold for the ordinary π-calculus). Section 6 sketches some possible extensions.

2 Basics

We now define the syntax, operational semantics, typing rules, and subsort relation of our calculus and develop some of its fundamental properties.

2.1 Notational Preliminaries

Since our purpose here is to study basic theoretical properties of our type system rather than to propose a pragmatic notation for programming, we adopt an explicitly typed presentation in which every bound name is annotated with a sorting. The algorithmic problem of inferring these annotations is deferred to future investigation (c.f. [Tur94, Gay93, VH93]). Our basic syntactic categories are defined by the grammar in Figure 1.

We use the metavariables S, T, and U for sorts; P, Q, and R for process expressions; Γ and Δ for sorting assumptions (or *sortings*); and a, b, c, etc. for channels (or *names*). For most of the paper, the only type of

			channel sorts				*processes*
S	$::=$	$(S_1, \ldots, S_n)^I$	tagged tuple	P	$::=$	$\mathbf{0}$	nil process
	$\mid$	$\mu A.\ S$	recursive sort		$\mid$	$P \mid P$	parallel
	$\mid$	A	sort variable		$\mid$	$(\boldsymbol{\nu}\, a{:}S)\, P$	restriction
					$\mid$	$a(a_1{:}S_1, \ldots, a_n{:}S_n)\, .\, P$	input
			I/O tags		$\mid$	$\overline{a}\langle a_1, \ldots, a_n\rangle\, .\, P$	output
I	$::=$	r	input only		$\mid$	$!\, P$	replication
	$\mid$	w	output only		$\mid$	*wrong*	error
	$\mid$	b	either				
							sortings
			process types	Γ	$::=$	$\emptyset$	empty
X	$::=$	$\circ$	ok process		$\mid$	$\Gamma,\, a{:}S$	sorted name

Figure 1: Syntax

well-formed processes is $\circ$; more interesting process types are considered in Section 6.2.[3]

$\tilde{S}$ stands for a sequence of sorts $S_1, \ldots, S_n$; similarly, $\tilde{a}$ stands for a sequence of names. We call a sort S *guarded* if it has the form $S = (\tilde{S})^I$.

We often write the process expression $\alpha.\ \mathbf{0}$, where α is an input or output prefix, as just α. Also, following Milner [Mil91], we introduce an explicit replication operator ! instead of allowing systems of recursive definitions of process expressions; $!\, P$ stands for the parallel composition of an infinite (countable) number of copies of P. To simplify the presentation, we omit the notational devices of *abstraction* and *concretion* [Mil91] and the basic operations of *summation* of processes and *matching* of names. We believe that our results can be extended straightforwardly to a calculus including summation, abstraction, and concretion, while the case of matching is more delicate; Section 6.1 discusses these extensions.

The constant *wrong* stands for a process in which a run-time type error has occured — i.e., one in which an attempted communication has involved an arity mismatch or, in the tagged reduction semantics of Section 3.2, a violation of an I/O restriction. The principal goal of the soundness theorem in Section 3 is to guarantee that a well-typed process expression cannot reduce to a process expression containing *wrong*.

We restrict our attention to *closed* judgements $\Gamma \vdash P : \circ$, i.e., those in which the free names of P are all bound in Γ, and we formally identify judgements up to renaming both of bound variables in P and of variables free in P and bound by Γ. This is equivalent to regarding alphabetic variable names — channel names and sort variables — as informal abbreviations for an underlying representation based on de Bruijn indices [dB72], and implies the usual conventions about name capture during substitution, alpha-conversion, side-conditions concerning freshness of names, etc. It also follows from this point of view that the names bound by a context Γ are always taken to be pairwise distinct; this justifies an abuse of notation whereby Γ is regarded as a finite function from names to sorts: $\Gamma(a)$ is the sort assigned to a by Γ. The order of bindings in Γ is ignored.

Since we want to view sort expressions as abbreviations for regular trees, we require that the body of a recursive channel sort $\mu A.\ S$ be *contractive* in the recursion variable A: either A does not appear at all in S, or else it appears inside at least one set of brackets. $T\{^S/_A\}$ denotes the capture-avoiding substitution of S for A in T. Similarly, $P\{^a/_b\}$ denotes the capture-avoiding substitution of a for b in P.

If $S = (\tilde{T})^I$, we call $\tilde{T}$ the *object part* of S, written $obj(S)$. Thus:

$$obj((\tilde{S})^I) = \tilde{S}$$

$$obj(\mu A.\ S) = obj(S\{^{\mu A.\ S}/_A\}).$$

We write $obj_i(S)$ for the ith component of $obj(S)$.

[3] We use nesting of brackets to describe, recursively, the sorts of the channels carried by other channels. Other papers on the π-calculus or extensions of it (e.g. [Mil91, San92]) use a similar bracket nesting notation for higher-order sorts, i.e. sorts of process abstractions. Our brief discussion of higher-order sorts in 6.2 uses arrow-sorts for this purpose.

Similarly, the function $\mathcal{I}$ extracts the top-level I/O annotation of a sort after unrolling as many outermost recursions as necessary to reach a guarded sort:

$$\mathcal{I}((S_1,\ldots,S_n)^I) = I$$

$$\mathcal{I}(\mu A.\ S) = \mathcal{I}(S\{{}^{\mu A.\ S}/A\}).$$

$\mathcal{I}_i$ extracts the I/O annotation of the ith component of a sort:

$$\mathcal{I}_i((S_1,\ldots,S_n)^I) = \mathcal{I}(S_i)$$

$$\mathcal{I}_i(\mu A.\ S) = \mathcal{I}_i(S\{{}^{\mu A.\ S}/A\}).$$

2.2 Reduction Semantics

Following Milner [Mil91], we present the operational semantics of the π-calculus using two relations: a *structural equivalence* on process terms that permits the rearrangement of parallel compositions, replications, and restrictions so that the participants in a potential communication can be brought into immediate proximity; and a *reduction relation* that describes the act of communication itself. Our semantics differs from the standard presentation only in the rules involving *wrong*.

2.2.1. Definition: The process-equivalence relation $P \equiv Q$ is the least congruence closed under the following rules:

$$P \,|\, Q \;\equiv\; Q \,|\, P \qquad \text{(E-Commut)}$$

$$(P \,|\, Q) \,|\, R \;\equiv\; P \,|\, (Q \,|\, R) \qquad \text{(E-Assoc)}$$

$$P \,|\, \mathbf{0} \;\equiv\; P \qquad \text{(E-Zero)}$$

$$!\,P \;\equiv\; P \,|\, !\,P \qquad \text{(E-Repl)}$$

$$((\boldsymbol{\nu}\, a{:}S)\, P) \mid Q \;\equiv\; (\boldsymbol{\nu}\, a{:}S)\, (P \,|\, Q) \qquad \text{when } a \text{ not free in } Q \qquad \text{(E-Extrusion)}$$

$$(\boldsymbol{\nu}\, a{:}S)\, (\boldsymbol{\nu}\, b{:}T)\, P \;\equiv\; (\boldsymbol{\nu}\, b{:}T)\, (\boldsymbol{\nu}\, a{:}S)\, P \qquad \text{(E-Switch)}$$

$$(\boldsymbol{\nu}\, a{:}S)\, \mathbf{0} \;\equiv\; \mathbf{0} \qquad \text{(E-Drop)}$$

$$P \mid \mathit{wrong} \;\equiv\; \mathit{wrong} \qquad \text{(E-Par-Wrong)}$$

$$!\,\mathit{wrong} \;\equiv\; \mathit{wrong} \qquad \text{(E-Repl-Wrong)}$$

$$(\boldsymbol{\nu}\, a{:}S)\, \mathit{wrong} \;\equiv\; \mathit{wrong} \qquad \text{(E-Restr-Wrong)}$$

(Note that the side condition on rule E-Extrusion can be viewed as a consequence of our convention of regarding names as deBruijn indices.) By "least congruence" we mean that a process P is related to P' "in a single step" if P' can be obtained from P by replacing a subphrase of P by another subphrase that is equivalent according to one of these rules.

2.2.2. Definition: The one-step reduction $P \longrightarrow Q$ is the least relation closed under the following rules:

$$\frac{m = n}{a(b_1{:}S_1,\ldots,b_m{:}S_m)\,.\,P \mid \overline{a}\langle c_1,\ldots,c_n\rangle\,.\,Q \;\longrightarrow\; P\{{}^{c_1,\ldots,c_n}/{}_{b_1,\ldots,b_m}\} \mid Q} \qquad \text{(R-Comm)}$$

$$\frac{m \neq n}{a(b_1{:}S_1,\ldots,b_m{:}S_m)\,.\,P \mid \overline{a}\langle c_1,\ldots,c_n\rangle\,.\,Q \;\longrightarrow\; \mathit{wrong}} \qquad \text{(R-Comm-Wrong)}$$

$$\frac{P \;\longrightarrow\; P'}{P \,|\, Q \;\longrightarrow\; P' \,|\, Q} \qquad \text{(R-Par)}$$

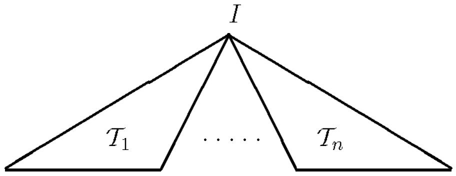
$$\frac{P \;\longrightarrow\; P'}{(\boldsymbol{\nu}\, a{:}S)\, P \;\longrightarrow\; (\boldsymbol{\nu}\, a{:}S)\, P'} \qquad \text{(R-Restr)}$$

$$\frac{P \;\equiv\; P' \;\longrightarrow\; Q' \;\equiv\; Q}{P \;\longrightarrow\; Q} \qquad \text{(R-Eqv)}$$

2.3 Subsorting

2.3.1. Definition: The subsort relation is generated by a basic *sub-tag* relation $I \leq J$, the least preorder containing $\mathsf{b} \leq \mathsf{r}$ and $\mathsf{b} \leq \mathsf{w}$.

2.3.2. Definition: An *I/O-tree* is a finitely branching tree whose nodes are labelled with symbols from $\{\mathsf{r}, \mathsf{w}, \mathsf{b}\}$. The metavariables $\mathcal{S}$, $\mathcal{T}$ and $\mathcal{U}$ range over trees; Tr denotes the class of all trees. The tree



whose root is labelled I and whose subtrees are $\mathcal{T}_1, \ldots, \mathcal{T}_n$ is written $[\mathcal{T}_1, \ldots, \mathcal{T}_n]^I$.

2.3.3. Definition: To each closed sort S we associate an I/O-tree called $\mathit{Tree}(S)$; it is the unique tree satisfying the following equations:

1. if $S = (S_1, \ldots, S_n)^I$ then $\mathit{Tree}(S) = [\mathit{Tree}(S_1), \ldots, \mathit{Tree}(S_n)]^I$;
2. if $S = \mu A.S'$ then $\mathit{Tree}(S) = \mathit{Tree}(S'\{^{\mu A.S'}/A\})$.

2.3.4. Definition: A relation $\Re \subseteq Tr \times Tr$ is a *tree simulation* if $(\mathcal{S}, \mathcal{T}) \in \Re$ implies:

1. if $\mathcal{T} = [\mathcal{T}_1, \ldots, \mathcal{T}_n]^{\mathsf{b}}$ then $\mathcal{S} = [\mathcal{S}_1, \ldots, \mathcal{S}_n]^{\mathsf{b}}$ and, for all $1 \leq i \leq n$, both $(\mathcal{S}_i, \mathcal{T}_i) \in \Re$ and $(\mathcal{T}_i, \mathcal{S}_i) \in \Re$;
2. if $\mathcal{T} = [\mathcal{T}_1, \ldots, \mathcal{T}_n]^{\mathsf{r}}$ then $\mathcal{S} = [\mathcal{S}_1, \ldots, \mathcal{S}_n]^I$ with $I \leq \mathsf{r}$ and, for all $1 \leq i \leq n$, $(\mathcal{S}_i, \mathcal{T}_i) \in \Re$;
3. if $\mathcal{T} = [\mathcal{T}_1, \ldots, \mathcal{T}_n]^{\mathsf{w}}$ then $\mathcal{S} = [\mathcal{S}_1, \ldots, \mathcal{S}_n]^I$ with $I \leq \mathsf{w}$ and, for all $1 \leq i \leq n$, $(\mathcal{T}_i, \mathcal{S}_i) \in \Re$.

2.3.5. Definition: The *tree subsort relation* $\mathcal{S} \leq_{tr} \mathcal{T}$ is

$$\mathcal{S} \leq_{tr} \mathcal{T} \quad \text{iff} \quad (\mathcal{S}, \mathcal{T}) \in \Re \text{ for some tree simulation } \Re.$$

2.3.6. Definition: The *subsort relation* $\vdash S \leq T$ is

$$\vdash S \leq T \quad \text{iff} \quad \mathit{Tree}(S) \leq_{tr} \mathit{Tree}(T).$$

We write $\vdash S \lessgtr T$ when $\vdash S \leq T$ and $\vdash T \leq S$. Note that the subsort relation is invariant under folding and unfolding of recursively defined sorts, since $\mathit{Tree}(\mu A.\ S) = \mathit{Tree}(S\{^{(\mu A.\ S)}/A\})$.

There is a close analogy between channel sorts and the *reference types* found in some programming languages. In Reynolds's language Forsythe [Rey88], for example, the type of a mutable storage cell holding a value of type T can be written $\mathit{ref}(T)$, an abbreviation for $\mathit{sink}(T) \wedge \mathit{source}(T)$. (Reynolds's actual notation is slightly different.) That is, a mutable cell containing an element of T is viewed as a connected pair of locations (or one location with two different types): a "source" for elements of T and a "sink" into which elements of T can be placed. The *source* constructor behaves covariantly in the subtype relation, since a source for elements of T can be used as a source for elements of U if $T \leq U$; whereas the *sink* constructor is contravariant: a sink for elements of U can be used in a context that expects a sink for elements of T, but not the other way around. The *ref* constructor, like our b tag, is constrained by both requirements and behaves non-variantly in the subtype relation.

A different analogy relates our definition of subsorting and the subtype relations found in some typed λ-calculi [Car86, CW85, Rey85]. A function f with type $S \rightarrow T$ can be thought of as a process that reads a value of type S from one location and writes a result of type T to another; to start f, we send it a pair of channels along a request channel a_f, telling it where to find its argument and where to put its result. The sort of a_f itself, from the point of view of a process invoking f, is $((S)^{\mathsf{r}}, (T)^{\mathsf{w}})^{\mathsf{w}}$. Note that S occurs in a contravariant position (under both an r tag and a w tag) and T occurs in a covariant position (under two w tags). Thus, the intuitive "compilation" $(S \rightarrow T) \mapsto ((S)^{\mathsf{r}}, (T)^{\mathsf{w}})^{\mathsf{w}}$ validates the λ-calculus subtyping rule for arrow types: $S_1 \rightarrow T_1 \leq S_2 \rightarrow T_2$ if $S_2 \leq S_1$ and $T_1 \leq T_2$.

Defining the subsort relation on two sort expressions via a relation on their regular tree expansions has the advantages of being semantically natural and technically well suited to our needs in the proof of soundness in Section 3. Furthermore, it supports straightforward verification of some important basic properties, for example, transitivity.

2.3.7. Proposition: The relation $\leq_{tr}$ is transitive.

Proof: Suppose $\mathcal{S}_1 \leq_{tr} \mathcal{S}_2 \leq_{tr} \mathcal{S}_3$. To prove that $\mathcal{S}_1 \leq_{tr} \mathcal{S}_3$ we have to find a tree simulation $\Re$ containing the pair $(\mathcal{S}_1, \mathcal{S}_3)$. Let $\Re_1$ and $\Re_2$ be tree simulations containing the pairs $(\mathcal{S}_1, \mathcal{S}_2)$ and $(\mathcal{S}_2, \mathcal{S}_3)$, respectively, and define

$$\begin{array}{rcl} \Re & = & \Re_1 \cdot \Re_2 \;\cup\; \Re_2 \cdot \Re_1 \\ & = & \{(\mathcal{S},\mathcal{U}) \mid \text{for some } \mathcal{T},\ (\mathcal{S},\mathcal{T}) \in \Re_1, (\mathcal{T},\mathcal{U}) \in \Re_2\} \\ & & \cup\, \{(\mathcal{U},\mathcal{S}) \mid \text{for some } \mathcal{T},\ (\mathcal{U},\mathcal{T}) \in \Re_2, (\mathcal{T},\mathcal{S}) \in \Re_1\}. \end{array}$$

The relation $\Re$ clearly includes the pair $(\mathcal{S}_1, \mathcal{S}_3)$, so it remains only to check that $\Re$ is a tree simulation. For each pair of trees in $\Re$, there are six cases to consider, since $\Re$ is the union of two relations and the definition of tree simulation has three clauses:

- Suppose $(\mathcal{S},\mathcal{U}) \in \Re_1 \cdot \Re_2$, and $\mathcal{U} = [\mathcal{U}_1, \ldots, \mathcal{U}_n]^{\mathsf{b}}$. Then there is some $\mathcal{T}$ such that $(\mathcal{S},\mathcal{T}) \in \Re_1$ and $(\mathcal{T},\mathcal{U}) \in \Re_2$. By the definition of tree simulation, $(\mathcal{T},\mathcal{U}) \in \Re_2$ implies

  $$\mathcal{T} = [\mathcal{T}_1, \ldots, \mathcal{T}_n]^{\mathsf{b}} \text{ with } (\mathcal{T}_i,\mathcal{U}_i), (\mathcal{U}_i,\mathcal{T}_i) \in \Re_2 .$$

  Similarly, $(\mathcal{S},\mathcal{T}) \in \Re_1$ implies

  $$\mathcal{S} = [\mathcal{S}_1, \ldots, \mathcal{S}_n]^{\mathsf{b}} \text{ with } (\mathcal{S}_i,\mathcal{T}_i), (\mathcal{T}_i,\mathcal{S}_i) \in \Re_1 .$$

  Thus

  $$\mathcal{S} = [\mathcal{S}_1, \ldots, \mathcal{S}_n]^{\mathsf{b}} \text{ with } (\mathcal{S}_i,\mathcal{U}_i), (\mathcal{U}_i,\mathcal{S}_i) \in \Re ,$$

  as required.

- Suppose $(\mathcal{S},\mathcal{U}) \in \Re_1 \cdot \Re_2$, and $\mathcal{U} = [\mathcal{U}_1, \ldots, \mathcal{U}_n]^{\mathsf{r}}$. Then there is some $\mathcal{T}$ such that $(\mathcal{S},\mathcal{T}) \in \Re_1$ and $(\mathcal{T},\mathcal{U}) \in \Re_2$. The latter implies $\mathcal{T} = [\mathcal{T}_1, \ldots, \mathcal{T}_n]^{I}$ with $I \in \{\mathsf{b}, \mathsf{r}\}$ and, for all $1 \leq i \leq n$, $(\mathcal{T}_i,\mathcal{U}_i) \in \Re_2$. If $I = \mathsf{r}$, then, since $\Re_1$ is a tree simulation, we have $\mathcal{S} = [\mathcal{S}_1, \ldots, \mathcal{S}_n]^{J}$ with $J \in \{\mathsf{b}, \mathsf{r}\}$ and, for all $1 \leq i \leq n$, $(\mathcal{S}_i,\mathcal{T}_i) \in \Re_2$. Thus, $(\mathcal{S}_i,\mathcal{U}_i) \in \Re_1 \cdot \Re_2$ for all $1 \leq i \leq n$, as required. The case where $I = \mathsf{b}$ proceeds similarly.

- The remaining four cases are similar. □

In practice, it is also useful to have a finitary characterization of subsorting in terms of a deterministic algorithm. Let Σ range over *subsorting contexts*:

$$\begin{array}{rcl} \Sigma & ::= & \emptyset \\ & \mid & \Sigma, S{\leq}T \end{array}$$

2.3.8. Definition: The *algorithmic subsort relation* $\Sigma \vdash S \leq T$ is the least relation closed under the following rules:

$$\frac{\text{for each } i,\ \Sigma \vdash S_i \gtrless T_i}{\Sigma \vdash (S_1, \ldots, S_n)^{\mathsf{b}} \leq (T_1, \ldots, T_n)^{\mathsf{b}}} \quad \text{(A-BB)}$$

$$\frac{I \leq \mathsf{r} \qquad \text{for each } i,\ \Sigma \vdash S_i \leq T_i}{\Sigma \vdash (S_1, \ldots, S_n)^{I} \leq (T_1, \ldots, T_n)^{\mathsf{r}}} \quad \text{(A-XI)}$$

$$\frac{I \leq \mathsf{w} \qquad \text{for each } i,\ \Sigma \vdash T_i \leq S_i}{\Sigma \vdash (S_1, \ldots, S_n)^{I} \leq (T_1, \ldots, T_n)^{\mathsf{w}}} \quad \text{(A-XO)}$$

$$\Sigma, S{\leq}T \vdash S \leq T \quad \text{(A-Ass)}$$

$$\frac{\Sigma, \mu A.\, S \leq T \vdash S\{(\mu A.\, S)/A\} \leq T}{\Sigma \vdash \mu A.\, S \leq T} \quad \text{(A-Rec-L)}$$

$$\frac{\Sigma, S \leq \mu B.\, T \vdash S \leq T\{(\mu B.\, T)/B\}}{\Sigma \vdash S \leq \mu B.\, T} \quad \text{(A-Rec-R)}$$

These rules can be read as an algorithm by imposing an ordering on them. We attempt to construct a derivation of a subsorting judgement $\Sigma \vdash S \leq T$ by applying the rules backwards in a goal-directed fashion, using A-REC-L and A-REC-R in either order when both apply, but preferring A-ASS over A-REC-L and A-REC-R whenever both apply. Derivations that obey these constraints are called *algorithmic derivations*. When Σ is empty we often omit it.

Note that our notation for the subsort relation defined by tree simulation ($S \leq T$) and the notation for the relation defined by this algorithm ($\Sigma \vdash S \leq T$) are nearly identical. This notational ambiguity will be justified by the fact that the two relations coincide when Σ is empty (Corollary 2.4.9).

Note that when the subsort algorithm is started on a closed judgement, all the subgoals generated during its execution are also closed judgements. Also, note that the nondeterminism of the algorithm is inessential, in the sense that the rules can be ordered so that no backtracking is needed.

2.4 Properties of Subsorting

We now establish a number of properties of the subsorting algorithm, including proofs of its termination and its soundness and completeness for the definition of subsorting as a tree simulation. For the moment, we use the notation $\Sigma \vdash S \leq T$ exclusively for the algorithmic subsort relation. The soundness and completeness of the algorithm will later justify our using the same notation to refer ambiguously to either definition of subsorting.

2.4.1. Lemma: The subsorting algorithm is terminating

Proof: Given a sort U, let $Sub[U]$ be the set obtained from the set of all the subterms of U by replacing each variable appearing free in a subterm with its definition; formally, $Sub[U]$ is defined inductively as follows:

$$\begin{array}{rcl} Sub[(U_1,\ldots,U_n)^I] & = & \{(U_1,\ldots,U_n)^I\} \cup Sub[U_1] \cup \ldots \cup Sub[U_n] \\ Sub[\mu A.U] & = & \{\mu A.U\} \cup \{S\{^{\mu A.U}/A\} \mid S \in Sub[U]\} \\ Sub[A] & = & \{A\}. \end{array}$$

It is easy to see that $Sub[U]$ is finite for any U; indeed, $Sub[U]$ can have no more elements than the number of distinct subterms of U. Now, let S and T be two sorts to which the algorithm is applied, and let $Sub[S,T] = Sub[S] \cup Sub[T]$.

We begin by showing that each node in a (not necessarily finite) execution trace $\mathcal{E}$ of the algorithm starting from the input $\vdash S \leq T$ satisfies three properties; the termination of the algorithm will then follow from these properties. Let $\Sigma \vdash S' \leq T'$ be the goal at a node N in $\mathcal{E}$. Then:

1. $S', T' \in Sub[S,T]$;
2. for every assumption $U_1 \leq U_2 \in \Sigma$, both $U_1, U_2 \in Sub[S,T]$;
3. no assumption occurs in Σ more than once: if both $U_1 \leq U_2 \in \Sigma$ and $U_1' \leq U_2' \in \Sigma$, then either $U_1 \neq U_1'$ or $U_2 \neq U_2'$.

These three properties clearly hold for the root $\emptyset \vdash S \leq T$ of the execution trace. Moreover, at each step of the algorithm, if the current goal node satisfies the three properties, then each immediate subgoal generated by the algorithm does too. For instance, take the rule A-REC-L:

$$\frac{\Sigma, \mu A.\ S \leq T \vdash S\{^{(\mu A.\ S)}/A\} \leq T}{\Sigma \vdash \mu A.\ S \leq T}$$

By the definition of the Sub function, if $\mu A.\ S' \in Sub[S]$, then $S'\{^{(\mu A.\ S')}/A\} \in Sub[S]$; thus property (1) holds. As for property (2), the pairs in Σ are in $Sub[S,T] \times Sub[S,T]$ by assumption, whereas the pair $\mu A.\ S \leq T$ is in $Sub[S,T] \times Sub[S,T]$ by property (1) of the main goal. Property (3) holds because the assumption $\mu A.\ S \leq T$ cannot already be in Σ. (If it were, the algorithm would have applied rule A-ASS rather than A-REC-L.)

Thus, each node of the execution trace $\mathcal{E}$ satisfies (1)–(3). We now associate a measure to each node, and, using properties (1)–(3), give a well-founded ordering such that each step of the algorithm decreases the measure of any new goals with respect to this ordering. It follows that the algorithm terminates.

The measure is

$$\mathcal{M}[\,\Sigma \vdash S \le T\,] = (n, m),$$

where n is the number of pairs in Σ and m is the maximum nesting of brackets in either S or T. Measures are ordered as follows:

$$(n, m) > (n', m') \text{ if } n < n' \text{ or } (n = n' \text{ and } m > m').$$

That is, $\mathcal{M}[\,\Sigma_1 \vdash S_1 \le T_1\,] > \mathcal{M}[\,\Sigma_2 \vdash S_2 \le T_2\,]$ if Σ_1 is shorter than Σ_2 or the Σ's have the same size and the maximum nesting of brackets is deeper in S_1 and T_1 than in S_2 and T_2.

It is easy to check that the application of a rule by the algorithm generates new nodes with a smaller measure than the main goal. The rule A-ASS does not generate any subgoals. The rules A-REC-L and A-REC-R strictly increase the length of the assumptions. The rules A-BB, A-XI, and A-XO remove an outer layer of brackets on both sides and thus decrease the maximum bracket nesting, while leaving the set of assumptions unchanged. Finally, the measure cannot decrease forever, since the first component can never become larger than the cardinality of $Sub[S,T] \times Sub[S,T]$, which we have previously seen is finite, and the second component cannot become smaller than 1. □

2.4.2. Lemma: [Semi-completeness of the algorithm] If $Tree(S) \le_{tr} Tree(T)$, then the algorithm does not return false when applied to $\vdash S \le T$.

Proof: Call a goal $\Sigma \vdash S \le T$ *sound* if

1. $Tree(S) \le_{tr} Tree(T)$;
2. $Tree(U_1) \le_{tr} Tree(U_2)$ for each $U_1 \le U_2 \in \Sigma$.

Fact: If a goal is sound, then the conclusion of one of the rules in Definition 2.3.8 matches the goal and the new subgoals corresponding to the premises of this rule are all sound goals.

Proof of the fact. Let $\Sigma \vdash S \le T$ be a sound goal. If $S \le T \in \Sigma$, then the rule A-ASS can be applied, generating no subgoals. On the other hand, suppose $S \le T \notin \Sigma$. If S and T are both guarded, then one of rules A-BB, A-XI or A-XO must be applicable, for otherwise the hypothesis $Tree(S) \le_{tr} Tree(T)$ would collapse; this follows straightforwardly from the fact that $Tree(S) \le_{tr} Tree(T)$ implies the existence of a simulation containing the pair $(Tree(S), Tree(T))$. To see that the new goals produced are sound, take, for example, the case of rule A-XI. Each of its premises has the form $\Sigma \vdash S_i \le T_i$. Since the set of assumptions is not modified, we only have to check that $Tree(S_i) \le_{tr} Tree(T_i)$; this follows from $Tree(S) \le_{tr} Tree(T)$ and the definition of tree simulation. Finally, if S or T is not guarded, then either rule A-REC-L or A-REC-R can be applied; in each case, the soundness of the new goal follows from the soundness the old goal and the fact that unfolding a term does not change its tree.

Proof of the lemma. Now consider the derivation produced by the algorithm on input $\vdash S \le T$ with $Tree(S) \le Tree(T)$. The algorithm starts with the sound goal $\vdash S \le T$. By the above fact, all the goals it generates are sound, and on a sound goal the algorithm can always either proceed or return success. Therefore the algorithm cannot fail when started with a sound input. □

2.4.3. Corollary: [Completeness of the algorithm] If $Tree(S) \le_{tr} Tree(T)$, then $\vdash S \le T$

Proof: If $Tree(S) \le_{tr} Tree(T)$, then by Lemma 2.4.2 the algorithm does not return false; but by Lemma 2.4.1 the algorithm is terminating; hence it must return true. □

2.4.4. Lemma: $\Sigma \vdash S \le T$ implies $\Sigma, \Sigma' \vdash S \le T$ for any Σ'.

Proof: By inspection of the rules. □

2.4.5. Lemma:

1. Suppose $\vdash \mu A.\ S \le T$ and $(\mu A.\ S \le T) \notin \Sigma$. Then

$$\Sigma, \mu A.\ S \le T \vdash U_1 \le U_2 \text{ implies } \Sigma \vdash U_1 \le U_2.$$

2. Suppose $\vdash S \le \mu A.\ T$ and $(S \le \mu A.\ T) \notin \Sigma$. Then

$$\Sigma, S \le \mu A.\ T \vdash U_1 \le U_2 \text{ implies } \Sigma \vdash U_1 \le U_2.$$

Proof: We show just part (1), since part (2) is analogous. The proof proceeds by induction on the length of a derivation of $\Sigma, \mu A.\ S \leq T \vdash U_1 \leq U_2$.

When the last rule used in the derivation is A-Ass, we must distinguish two subcases:

1. $U_1 \leq U_2 \in \Sigma$: In this case, the rule A-Ass can also be applied with the smaller premise $\Sigma \vdash U_1 \leq U_2$;
2. $U_1 = \mu A.\ S$ and $U_2 = T$: By hypothesis, $\vdash \mu A.\ S \leq T$ holds; hence by Lemma 2.4.4, $\Sigma \vdash \mu A.\ S \leq T$.

When the last rule of the derivation is not A-Ass, it must be the case that $(U_1 \leq U_2) \notin \Sigma$; moreover, the applicability of the rule does not depend on the contents of Σ. The conclusion follows by induction. □

2.4.6. Corollary:

1. If $\vdash \mu A.S \leq T$, then $\vdash S\{^{\mu A.S}/A\} \leq T$.
2. If $\vdash S \leq \mu A.T$, then $\vdash S \leq T\{^{\mu A.T}/A\}$.

Proof: Since we stipulated at the outset that the bodies of recursive sorts must be contractive in the recursion variable, each of S and T can be unrolled a finite number of times to obtain guarded sorts S' and T'. Proceed by induction on the total number of unrollings needed to reach this state.

In the induction step, when either S or T has the form of a recursion, the last rule applied in a derivation of $\vdash S \leq T$ must be either A-Rec-L or A-Rec-R; consider A-Rec-L (the other case is symmetric). Then $S = \mu A.S_1$ and $\mu A.S_1 \leq T \vdash S\{^{\mu A.S_1}/A\} \leq T$. Use Lemma 2.4.5 to obtain $\vdash S\{^{\mu A.S_1}/A\} \leq T$, as required for part (1). Part (2) then follows by induction. □

2.4.7. Corollary: Suppose $\vdash S \leq T$. Then for the guarded S' obtained by unfolding all the outermost recursions of S and the guarded T' formed by unfolding the outermost recursions of T, it holds that $\vdash S' \leq T'$

2.4.8. Theorem: [Soundness of the algorithm] If $\vdash S \leq T$ then $\mathit{Tree}(S) \leq_{tr} \mathit{Tree}(T)$.

Proof: By Corollary 2.4.7 and the fact that the unfolding of a term does not change its tree, it suffices to prove the theorem for the case where S and T are guarded. We do this by showing that

$$\Re = \{(\mathit{Tree}(S), \mathit{Tree}(T)) \mid \vdash S \leq T \text{ and } S \text{ and } T \text{ guarded}\}$$

is a tree simulation. We only consider clause (3) of definition 2.3.4; clauses (1) and (2) are similar.

Assume that $(\mathit{Tree}(S), \mathit{Tree}(T)) \in \Re$ and

$$\mathit{Tree}(T) = [\mathcal{T}_1, \ldots, \mathcal{T}_n]^{\mathsf{w}}.$$

Since T is guarded, we have

$$T = (T_1, \ldots, T_n)^{\mathsf{w}} \text{ with } \mathcal{T}_i = \mathit{Tree}(T_i), \text{ for each } i.$$

Since S is also guarded, the final rule used in the derivation of $\vdash S \leq T$ must be A-XO. Therefore, $S = (S_1, \ldots, S_n)^I$ with $I \leq \mathsf{w}$,

$$\vdash T_i \leq S_i, \text{for each } i, \tag{1}$$

and $\mathit{Tree}(S) = [\mathit{Tree}(S_1), \ldots, \mathit{Tree}(S_n)]^I$. To satisfy the definition of tree simulation, it remains to show that

$$(\mathit{Tree}(T_i), \mathit{Tree}(S_i)) \in \Re \tag{2}$$

for each i. From (1) and Corollary 2.4.7 there are guarded terms T_i', S_i' that are unfoldings of T_i, S_i, respectively and such that $\vdash T_i' \leq S_i'$. By the definition of $\Re$, $(\mathit{Tree}(T_i'), \mathit{Tree}(S_i')) \in \Re$; but the unfolding of a term does not change its tree, so $\mathit{Tree}(T_i') = \mathit{Tree}(T_i)$ and $\mathit{Tree}(S_i') = \mathit{Tree}(S_i)$. This proves (2). □

2.4.9. Corollary: [Equivalence of the definitions of subtyping] $\vdash S \leq T$ iff $S \leq T$ (i.e. $\mathit{Tree}(S) \leq_{tr} \mathit{Tree}(T)$), for any S and T.

Proof: By Corollaries 2.4.3 and 2.4.8. □

This equivalence justifies our use of the symbol $\leq$ for both the tree-simulation and the algorithmic subsort relations. Indeed, for the remainder of the paper we shall ignore the distinction between the two systems.

We close the section on subsorting with some technical lemmas that we shall need later.

2.4.10. Lemma: If $S \leq (U_1, \ldots, U_m)^{\mathsf{r}}$ and $S \leq (T_1, \ldots, T_n)^{\mathsf{w}}$, then $m = n$ and, for each i, $T_i \leq U_i$.

Proof: By Corollary 2.4.7, we may assume without loss of generality that S is guarded, i.e. $S = (S_1, \ldots, S_p)^I$. The final rule in an (algorithmic) derivation of $\vdash S \leq (U_1, \ldots, U_m)^{\mathsf{r}}$ can only be A-XI; this implies that $p = m$ and, for each i, that $\vdash S_i \leq U_i$. Similarly, the final rule of a derivation of $\vdash S \leq (T_1, \ldots, T_n)^{\mathsf{w}}$ must be A-XO, which yields $p = n$ and $\vdash T_i \leq S_i$. for each i. So $m = n$ and, by the transitivity of subtyping, $T_i \leq U_i$ for each i. □

2.4.11. Lemma:

1. Suppose $U \leq (\tilde{T})^{\mathsf{r}}$ and $U \leq (\tilde{S})^{\mathsf{w}}$. Then $\mathcal{I}(U) = \mathsf{b}$ with $\tilde{S} \leq obj(U) \leq \tilde{T}$.
2. Suppose $(\tilde{T})^{\mathsf{b}} \leq U \leq (\tilde{S})^{\mathsf{w}}$; then $\mathcal{I}(U) \leq \mathsf{w}$ with $\tilde{S} \leq obj(U) \leq \tilde{T}$.

Proof: Similar to the proof of Lemma 2.4.10. □

2.4.12. Lemma: If $S \leq T$, then $\mathcal{I}(S) \leq \mathcal{I}(T)$.

Proof: By Corollary 2.4.7, it suffices to consider the case where S and T are guarded. The last rule in a derivation of $\vdash S \leq T$ is therefore one of A-BB, A-XI, and A-XO. The result is immediate in each case. □

2.4.13. Lemma: If $\mathcal{I}(T) \leq \mathsf{w}$ and $S \leq T$, then $\mathcal{I}_i(T) \leq \mathcal{I}_i(S)$ for each i.

Proof: Again, it suffices to consider the case where S and T are guarded. Since $\mathcal{I}(T) \leq \mathsf{w}$, then the final rule in a derivation of $\vdash S \leq T$ is either A-BB or A-XO. In either case, the right-hand premise yields $\vdash T_i \leq S_i$, from which the result follows by Lemma 2.4.12. □

2.5 Typing

The typing judgement $\Gamma \vdash P : \circ$ asserts that "P is a well-behaved process under the assumptions Γ," i.e. if P is placed in an execution context whose free names support the protocol described by Γ, then the execution of P can never encounter an internal error in communication and P's use of these names also obeys the protocol. There is one typing rule for each syntactic form except *wrong*, which does not behave correctly under any assumptions.

The inert process $\mathbf{0}$ is well behaved in any context.

$$\Gamma \vdash \mathbf{0} : \circ \qquad \text{(T-NIL)}$$

The parallel composition of two processes is well behaved if each is well behaved in isolation. (The possibility of a bad communication between P and Q is detected as a failure by one or both to satisfy the requirements imposed by Γ.) Similarly, a replication of P is well behaved if a single copy is.

$$\frac{\Gamma \vdash P : \circ \qquad \Gamma \vdash Q : \circ}{\Gamma \vdash P \,|\, Q : \circ} \qquad \text{(T-PAR)}$$

$$\frac{\Gamma \vdash P : \circ}{\Gamma \vdash !\,P : \circ} \qquad \text{(T-REPL)}$$

A process whose outermost constructor is a restriction is well behaved if its body observes the constraints imposed both by Γ and by the declared sorting of the new local channel.

$$\frac{\Gamma, a{:}S \vdash P : \circ}{\Gamma \vdash (\boldsymbol{\nu}\, a{:}S)\, P : \circ} \qquad \text{(T-RESTR)}$$

The interesting cases are the rules for input and output. In order to be sure that the input expression $a(b_1{:}S_1, \ldots, b_n{:}S_n)\,.\,P$ is well behaved, we must first show that the sorting of a in the current context has arity n and guarantees that a tuple of values read from a will have sorts smaller than $S_1, \ldots, S_n$. These two conditions can be combined by requiring that $\Gamma(a) \leq (S_1, \ldots, S_n)^{\mathsf{r}}$. (The use of the subsort relation validates the cases where $\Gamma(a) = (S_1, \ldots, S_n)^{\mathsf{b}}$ or when $\Gamma(a) = (S'_1, \ldots, S'_n)^{\mathsf{r}}$ for some $S'_1 \leq S_1, \ldots, S'_n \leq S_n$.)

Furthermore, we must check that the body P is well behaved assuming that the channels $b_1, \ldots, b_n$ behave consistently with the sorts $S_1, \ldots, S_n$.

$$\frac{\vdash \Gamma(a) \leq (S_1, \ldots, S_n)^{\mathsf{r}} \qquad \Gamma, b_1{:}S_1, \ldots, b_n{:}S_n \vdash P : \circ}{\Gamma \vdash a(b_1{:}S_1, \ldots, b_n{:}S_n) \,.\, P : \circ} \tag{T-In}$$

The case for output is analogous. To verify that the output expression $\overline{a}\langle b_1, \ldots, b_n \rangle \,.\, P$ is well behaved, we must check that the sort of a in Γ permits a to be used for outputting the n-tuples of values $b_1, \ldots, b_n$. Using the subsort relation, we can express this constraint elegantly by stipulating that the sort of a should be a subtype of the tuple of the sorts of the b_i's. Finally, P itself must be well-behaved.

$$\frac{\vdash \Gamma(a) \leq (\Gamma(b_1), \ldots, \Gamma(b_n))^{\mathsf{w}} \qquad \Gamma \vdash P : \circ}{\Gamma \vdash \overline{a}\langle b_1, \ldots, b_n \rangle \,.\, P : \circ} \tag{T-Out}$$

2.5.1. Lemma: [Syntax-directedness of typing] For each process constructor C, there is at most one typing rule R such that every valid derivation of $\Gamma \vdash P : \circ$, where P's outer constructor is C, ends with an instance of R. Moreover, each typing rule has the property that all metavariables appearing in its premises also appear in its conclusion. Thus, given a derivation with conclusion $\Gamma \vdash P : \circ$, we can uniquely determine the conclusions of its immediate subderivations by examining the form of P.

2.6 Properties of Typing

2.6.1. Lemma: [Weakening] If $\Gamma \vdash P : \circ$ then $\Gamma, a{:}S \vdash P : \circ$ for any S.

Proof: Since P is well-formed in a context where a is not bound, a cannot occur free in P; so the sort of a plays no role in the typing of P. □

2.6.2. Lemma: [Narrowing] If $\Gamma, a{:}S \vdash P : \circ$ and $T \leq S$ then $\Gamma, a : T \vdash P : \circ$.

Proof: By induction on the structure of P.

1. $P = \mathbf{0}$: Trivial.
2. $P = P_1 | P_2$: Use induction on the subcomponents P_1 and P_2.
3. $P = (\boldsymbol{\nu}\, b : U)\, Q$: We know that $\Gamma, a : S \vdash P : \circ$, which, by the syntax-directedness of typing, means that
$$\Gamma, a : S, b : U \vdash Q : \circ. \tag{3}$$
Now, $\Gamma, a : T \vdash Q : \circ$ holds if
$$\Gamma, b : U, a : T \vdash Q : \circ,$$
which can be inferred from (3) and the induction hypothesis.
4. $P = b(\tilde{x} : \tilde{S}) \,.\, Q$: Let $\Gamma' = \Gamma, a : S$. From the hypothesis $\Gamma' \vdash P : \circ$ we get
$$\Gamma', \tilde{x} : \tilde{S} \vdash Q : \circ \tag{4}$$
$$\Gamma'(b) < (\tilde{S})^{\mathsf{r}}. \tag{5}$$
Let $\Gamma'' = \Gamma, a : T$. To infer $\Gamma'' \vdash P : \circ$ we have to show that
$$\Gamma'', \tilde{x} : \tilde{S} \vdash Q : \circ$$
$$\Gamma''(b) < (\tilde{S})^{\mathsf{r}}.$$
Now, the former follows from (4) and the induction assumption. The latter follows from (5); in the case $b = a$ we need the hypothesis $\Gamma''(a) = T < S = \Gamma'(a)$.

5. $P = \overline{b}\langle\tilde{y}\rangle \,.\, Q$: We reason similarly as in the case for input. Thus, let $\Gamma' = \Gamma, a : S$ and $\Gamma'' = \Gamma, a : T$. From $\Gamma' \vdash P : \circ$ we have, for some $\tilde{U}$

$$\Gamma' \vdash Q : \circ \tag{6}$$
$$\Gamma'(b) < (\tilde{U})^{\mathsf{w}} \tag{7}$$
$$\Gamma'(\tilde{y}) < \tilde{U}. \tag{8}$$

On the other hand, $\Gamma'' \vdash P : \circ$ holds if we can show

$$\Gamma'' \vdash Q : \circ$$
$$\Gamma''(b) < (\tilde{U})^{\mathsf{w}}$$
$$\Gamma''(\tilde{y}) < \tilde{U}.$$

These assertions can be inferred from (6-8) using the induction assumption and the hypothesis $\Gamma'' < \Gamma'$.

6. $P = !\, P_1$: By induction. □

2.6.3. Lemma: [Substitution] If $\Gamma,\ b_1{:}S_1, \ldots, b_n{:}S_n \ \vdash\ P \ :\ \circ$ and, for each i, $\vdash \Gamma(c_i) \leq S_i$, then $\Gamma \vdash P\{c_1, \ldots, c_n / b_1, \ldots, b_n\} : \circ$.

Proof: By a straightforward induction, using narrowing in the T-In and T-Out cases. □

2.6.4. Lemma: If $\Gamma \vdash a(b_1 : S_1, \ldots, b_n : S_n) \,.\, P_1 \,|\, \overline{a}\langle c_1, \ldots, c_n\rangle \,.\, P_2 : \circ$, then $\Gamma(c_i) \leq S_i$, for all i.

Proof: By the definition of the typing relation we have:

$$\Gamma(a) \leq (\Gamma(c_1), \ldots, \Gamma(c_i), \ldots, \Gamma(c_n))^{\mathsf{w}} \quad \text{and} \quad \Gamma(a) \leq (S_1, \ldots, S_n)^{\mathsf{r}}.$$

By Lemma 2.4.10, $\Gamma(c_i) \leq S_i$. □

2.6.5. Definition: We say that Γ is an *extended subcontext* of Δ, written $\Gamma <_{\mathrm{E}} \Delta$, if the restriction of Γ to the names defined in Δ is a subtype of Δ; that is, for each $a \in dom(\Gamma)$, either $\Delta(a)$ is not defined or $\Gamma(a) \leq \Delta(a)$.

2.6.6. Corollary: If $\Gamma <_{\mathrm{E}} \Delta$ and $\Delta \vdash P : \circ$ then $\Gamma \vdash P : \circ$.

Proof: By weakening and narrowing. □

3 Soundness

In this section, we show that the I/O annotations on channel sorts can be trusted to restrict improper use of channels. This is accomplished by defining a refined reduction semantics in which each occurrence of a name is *tagged* to indicate whether that occurrence may validly be used for input and/or output. Attempts to misuse names during communication are caught by checking the consistency of the tags.

On well typed processes, this sematics is isomorphic to the given above, in the sense that every process development in the tagged semantics can be mirrored in the original semantics, and vice versa. Thus, the soundness theorem for the tagged semantics yields the soundness of the original semantics as an easy corollary.

3.1 Tagged Processes

3.1.1. Definition: The syntax of processes is refined as follows:

			tagged process expressions
E	$::=$	$\mathbf{0}$	nil process
	$\mid$	$E \,\vert\, F$	parallel composition
	$\mid$	$(\boldsymbol{\nu}\, a{:}S)\, E$	restriction
	$\mid$	$a^I(a_1{:}S_1, \ldots, a_n{:}S_n) \,.\, E$	input
	$\mid$	$\overline{a^I}\langle a_1^{I_1}, \ldots, a_n^{I_n}\rangle \,.\, E$	output
	$\mid$	$!\, E$	replication
	$\mid$	*wrong*	error

We use E, F, and G to range over process expressions with tagged names.

3.1.2. Definition: *Erase* is the function mapping tagged process expressions to ordinary process expressions by erasing tags.

3.1.3. Definition: The relation $ok_\Gamma(E)$, pronounced "the tags of E are statically consistent with Γ," is the least relation such that:

$$
\begin{array}{ll}
ok_\Gamma(\mathbf{0}) & \\
ok_\Gamma(\mathit{wrong}) & \\
ok_\Gamma(E \mid F) & \text{if } ok_\Gamma(E) \text{ and } ok_\Gamma(F) \\
ok_\Gamma((\boldsymbol{\nu}\, a{:}S)\, E) & \text{if } ok_{\Gamma,\ a:S}(E) \\
ok_\Gamma(a^I(b_1{:}S_1,\ldots,b_n{:}S_n)\,.\,E) & \text{if } \mathcal{I}(\Gamma(a)) \leq I \leq \mathsf{r} \\
& \text{and } ok_{\Gamma,\ b_1:S_1,\ldots,b_n:S_n}(E) \\
ok_\Gamma(\overline{a^I}\langle b_1^{I_1},\ldots,b_n^{I_n}\rangle\,.\,E) & \text{if } \mathcal{I}(\Gamma(a)) \leq I \leq \mathsf{w} \\
& \text{and } \mathcal{I}(\Gamma(b_i)) \leq I_i \leq \mathcal{I}_i(\Gamma(a)) \text{ for each } i \\
& \text{and } ok_\Gamma(E) \\
ok_\Gamma(!\,E) & \text{if } ok_\Gamma(E)
\end{array}
$$

3.1.4. Definition: The compilation function *Tag* is defined as follows:

$$
\begin{array}{lcl}
Tag_\Gamma(\mathbf{0}) & = & \mathbf{0} \\
Tag_\Gamma(\mathit{wrong}) & = & \mathit{wrong} \\
Tag_\Gamma(P \mid F) & = & Tag_\Gamma(P) \mid Tag_\Gamma(Q) \\
Tag_\Gamma((\boldsymbol{\nu}\, a{:}S)\, P) & = & (\boldsymbol{\nu}\, a{:}S)\ Tag_{\Gamma,\ a:S}(P) \\
Tag_\Gamma(a(b_1{:}S_1,\ldots,b_n{:}S_n)\,.\,P) & = & a^{\mathcal{I}(\Gamma(a))}(b_1{:}S_1,\ldots,b_n{:}S_n)\,.\ Tag_{\Gamma,\ b_1:S_1,\ldots,b_n:S_n}(P) \\
Tag_\Gamma(\overline{a}\langle b_1,\ldots,b_n\rangle\,.\,P) & = & \overline{a^{\mathcal{I}(\Gamma(a))}}\langle b_1^{\mathcal{I}(\Gamma(b_1))},\ldots,b_n^{\mathcal{I}(\Gamma(b_n))}\rangle\,.\ Tag_\Gamma(P) \\
Tag_\Gamma(!\,P) & = & !\,(Tag_\Gamma(P))
\end{array}
$$

3.1.5. Fact: If $\Gamma \vdash P : \circ$, then $ok_\Gamma(Tag_\Gamma(P))$.

Proof: Straightforward induction. □

Furthermore, it is easy to check that static consistency is invariant under tag-respecting substitutions:

3.1.6. Lemma: If $ok_{\Gamma,\ b_1:S_1,\ldots,b_n:S_n}(E)$ and, for each i, $\Gamma(c_i) \leq S_i$, then $ok_\Gamma(E\{c_1,\ldots,c_n/b_1,\ldots,b_m\})$.

Proof: By induction on the structure of E. The most interesting case is when $E = \overline{a}\langle a_1,\ldots,a_m\rangle\,.\,F$. Let $a' = a\{c_1,\ldots,c_n/b_1,\ldots,b_m\}$. If $a \notin \{b_1,\ldots,b_n\}$, then $a = a'$ and the first condition, $\mathcal{I}(\Gamma(a')) \leq I$, is satisfied by hypothesis; otherwise $a = c_i$ with $\Gamma(c_i) \leq \Gamma(b_i)$, and $\mathcal{I}(\Gamma(a')) \leq I$ holds by transitivity. In either case, $I \leq \mathsf{w}$ by hypothesis. Moreover, by Lemma 2.4.13, $\mathcal{I}_i(\Gamma(a)) \leq \mathcal{I}_i(\Gamma(a'))$ for each i, so $I_i \leq \mathcal{I}_i(\Gamma(a'))$ for each i. Thus, if any $a_i \in \{b_1,\ldots,b_n\}$, the assumption $\Gamma(c_i) \leq S_i$ plus transitivity gives $\mathcal{I}(\Gamma(c_i)) \leq I_i \leq \mathcal{I}_i(\Gamma(a'))$, as required. □

3.2 Tagged Reduction

3.2.1. Definition: The tagged equivalence $E \equiv E'$ is identical to the original equivalence $P \equiv P'$ of Definition 2.2.1, with tagged channels replacing untagged ones.

3.2.2. Definition: The one-step tagged reduction relation $E \longrightarrow F$ is the least relation closed under the following rules:

$$
\frac{m = n \qquad I \leq \mathsf{r} \qquad J \leq \mathsf{w} \qquad \text{for each } i,\ J_i \leq \mathcal{I}(S_i)}{a^I(b_1{:}S_1,\ldots,b_m{:}S_m)\,.\,E \mid \overline{a^J}\langle c_1^{J_1},\ldots,c_n^{J_n}\rangle\,.\,F \ \longrightarrow\ E\{c_1,\ldots,c_n/b_1,\ldots,b_m\} \mid F} \quad (\text{TR-Comm})
$$

$$\frac{m \neq n \quad \text{or} \quad I \not\leq \mathsf{r} \quad \text{or} \quad J \not\leq \mathsf{w} \quad \text{or, for some } i, \quad J_i \not\leq \mathcal{I}(S_i)}{a^I(b_1{:}S_1, \ldots, b_m{:}S_m) \,.\, E \mid \overline{a^J}\langle c_1^{J_1}, \ldots, c_n^{J_n}\rangle \,.\, F \;\longrightarrow\; \mathit{wrong}} \quad \text{(TR-Comm-Wrong)}$$

$$\frac{E \;\longrightarrow\; E'}{E \mid F \;\longrightarrow\; E' \mid F} \quad \text{(TR-Par)}$$

$$\frac{E \;\longrightarrow\; E'}{(\boldsymbol{\nu}\, a{:}S)\, E \;\longrightarrow\; (\boldsymbol{\nu}\, a{:}S)\, E'} \quad \text{(TR-Restr)}$$

$$\frac{E \;\equiv\; E' \;\longrightarrow\; F' \;\equiv\; F}{E \;\longrightarrow\; F} \quad \text{(TR-Eqv)}$$

3.3 Soundness of Typing

3.3.1. Lemma: [Correctness of *Erase* w.r.t. $\equiv$]

1. If $E \equiv E'$ then $\mathit{Erase}(E) \equiv \mathit{Erase}(E')$.
2. If $\mathit{Erase}(E) \equiv P$, then there is some $E' \equiv E$ such that $\mathit{Erase}(E') = P$.

Proof: Straightforward, since none of the tagged structural congruence rules either use or change tags. □

3.3.2. Lemma: [Preservation of types and tags under $\equiv$]

1. If $\mathit{ok}_\Gamma(E)$ and $E \equiv E'$, then $\mathit{ok}_\Gamma(E')$.
2. If $\Gamma \vdash E : \circ$ and $E \equiv E'$, then $\Gamma \vdash E' : \circ$.

Proof: Straightforward induction on the length of a derivation of $E \equiv E'$, with an inner induction at each step on the structure of E. □

3.3.3. Lemma: Let $E = a^I(b_1{:}S_1, \ldots, b_m{:}S_m) \,.\, F \mid \overline{a^J}\langle c_1^{J_1}, \ldots, c_n^{J_n}\rangle \,.\, G$. If $\Gamma \vdash \mathit{Erase}(E) : \circ$ and $\mathit{ok}_\Gamma(E)$, then

1. $I \leq \mathsf{r}$, $J \leq \mathsf{w}$, $m = n$, and $J_i \leq \mathcal{I}(S_i)$ for each i;
2. $\Gamma(c_i) \leq S_i$ for each i.

Proof: By the syntax-directedness of typing, we have

$$\begin{array}{l} \Gamma \vdash \mathit{Erase}(a^I(b_1{:}S_1, \ldots, b_m{:}S_m) \,.\, F) : \circ \\ \vdash \Gamma(a) \leq (S_1, \ldots, S_m)^{\mathsf{r}} \\ \Gamma, b_i{:}S_i \vdash \mathit{Erase}(F) : \circ \end{array}$$

and

$$\begin{array}{l} \Gamma \vdash \mathit{Erase}(\overline{a^J}\langle c_1^{J_1}, \ldots, c_n^{J_n}\rangle \,.\, G) : \circ \\ \vdash \Gamma(a) \leq (T_1, \ldots, T_n)^{\mathsf{w}} \\ \text{for each } i, \quad \vdash \Gamma(c_i) \leq T_i \\ \Gamma \vdash \mathit{Erase}(G) : \circ. \end{array}$$

By Lemma 2.4.10, $m = n$ and $\vdash T_i \leq S_i$ for each i. By the definition of ok_Γ, we have $I \leq \mathsf{r}$ and $J \leq \mathsf{w}$, and $J_i \leq \mathcal{I}_i(\Gamma(a))$ for each i. By the definition of $\mathcal{I}$— and transitivity, $J_i \leq \mathcal{I}(S_i)$. Finally, by the transitivity of subtyping (2.3.7), $\vdash \Gamma(c_i) \leq S_i$ for each i. □

Using the properties of typing and subtyping established in this section and the previous one, we obtain a straightforward proof of the soundness of the original typing rules with respect to the tagged semantics.

3.3.4. Theorem: If $\Gamma \vdash \mathit{Erase}(E) : \circ$ and $\mathit{ok}_\Gamma(E)$ and $E \longrightarrow E'$, then $\Gamma \vdash \mathit{Erase}(E') : \circ$ and $\mathit{ok}_\Gamma(E')$.

Proof: By induction on the derivation of $E \longrightarrow E'$.

Case TR-Comm: $E = F \mid G$
$F = a^I(b_1{:}S_1, \ldots, b_n{:}S_n)\,.\,F_1$
$G = \overline{a^J}\langle c_1^{J_1}, \ldots, c_n^{J_n}\rangle\,.\,G_1$
$E' = F_1\{{}^{c_1,\ldots,c_n}/_{b_1,\ldots,b_m}\} \mid G_1$
$m = n, I \leq \mathsf{r}, J \leq \mathsf{w}$, and for each i, $J_i \leq I_i$

By Lemma 3.3.3(2), $\vdash \Gamma(c_i) \leq S_i$ for each i. By the syntax-directedness of typing and the substitution lemma (2.6.3), $\Gamma \vdash \mathit{Erase}(F_1)\{{}^{c_1,\ldots,c_n}/_{b_1,\ldots,b_m}\} : \circ$, i.e., $\Gamma \vdash \mathit{Erase}(F_1\{{}^{c_1,\ldots,c_n}/_{b_1,\ldots,b_m}\}) : \circ$. By T-Par, $\Gamma \vdash \mathit{Erase}(F_1\{{}^{c_1,\ldots,c_n}/_{b_1,\ldots,b_n}\}) \mid \mathit{Erase}(G_1) : \circ$, i.e., $\Gamma \vdash \mathit{Erase}(E') : \circ$. Furthermore Lemma 3.1.6 yields $\mathit{ok}_\Gamma(F_1\{{}^{c_1,\ldots,c_n}/_{b_1,\ldots,b_m}\})$, from which $\mathit{ok}_\Gamma(E')$ follows by definition.

Case TR-Comm-Wrong: $E = F \mid G$
$F = a^I(b_1{:}S_1, \ldots, b_n{:}S_n)\,.\,F_1$
$G = \overline{a^I}\langle c_1^{J_1}, \ldots, c_n^{J_n}\rangle\,.\,G_1$
$E' = \mathit{wrong}$
$m \neq n$ or $I \not\leq \mathsf{r}$ or $J \not\leq \mathsf{w}$ or, for some i, $J_i \not\leq I_i$

By Lemma 3.3.3(1), this case cannot occur.

Case TR-Eqv: $E \equiv F$
$F \longrightarrow F'$
$F' \equiv E'$

By Lemmas 3.3.1 and 3.3.2, $\Gamma \vdash \mathit{Erase}(F) : \circ$ and $\mathit{ok}_\Gamma(F)$. By induction hypothesis, $\Gamma \vdash \mathit{Erase}(F') : \circ$ and $\mathit{ok}_\Gamma(F')$. By 3.3.1 and 3.3.2 again, $\Gamma \vdash \mathit{Erase}(E') : \circ$ and $\mathit{ok}_\Gamma(E')$.

Other cases:
By straightforward use of the induction hypothesis. □

3.3.5. Corollary: If $\Gamma \vdash \mathit{Erase}(E) : \circ$ and $\mathit{ok}_\Gamma(E)$, then $E \not\longrightarrow \mathit{wrong}$ and $\mathit{Erase}(E) \not\longrightarrow \mathit{wrong}$.

Moreover, on well-typed processes there is an exact correspondence between the tagged and the ordinary semantics.

3.3.6. Theorem: Suppose that $\Gamma \vdash \mathit{Erase}(E) : \circ$ and $\mathit{ok}_\Gamma(E)$. Then:

1. if $E \longrightarrow E'$, then $\mathit{Erase}(E) \longrightarrow \mathit{Erase}(E')$;
2. if $\mathit{Erase}(E) \longrightarrow P$, then there is some E' such that $E \longrightarrow E'$ and $\mathit{Erase}(E') = P$.

Proof: We consider only (2), since (1) is simpler and can be handled in a similar way. We proceed by induction on the size of the derivation of $\mathit{Erase}(E) \longrightarrow P$. Corollary 3.3.5 excludes that the last rule used in the derivation be R-Comm-Wrong. Suppose that the last rule was R-Comm. We then have

$$\mathit{Erase}(E) = a(b_1{:}S_1, \ldots, b_n{:}S_n)\,.\,P_1 \mid \overline{a}\langle c_1, \ldots, c_n\rangle\,.\,P_2 \;\longrightarrow\; P_1\{{}^{c_1,\ldots,c_n}/_{b_1,\ldots,b_m}\} \mid P_2.$$

By the definition of Erase, E must be of the form

$$a^I(b_1{:}S_1, \ldots, b_n{:}S_m)\,.\,E_1 \mid \overline{a^J}\langle c_1^{J_1}, \ldots, c_n^{J_n}\rangle\,.\,E_2$$

with $\mathit{Erase}(E_1) = P_1$ and $\mathit{Erase}(E_2) = P_2$. Since $\Gamma \vdash \mathit{Erase}(E) : \circ$ and $\mathit{ok}_\Gamma(E)$, by Lemma 3.3.3 it follows that $I \leq \mathsf{r}$ and $J \leq \mathsf{w}$, and that $J_i \leq \mathcal{I}(S_i)$ for each i. Therefore TR-Comm can be applied to E yielding

$$a^I(b_1{:}S_1, \ldots, b_n{:}S_m)\,.\,E_1 \mid \overline{a^J}\langle c_1^{J_1}, \ldots, c_n^{J_n}\rangle\,.\,E_2 \;\longrightarrow\; E_1\{{}^{c_1,\ldots,c_n}/_{b_1,\ldots,b_m}\} \mid E_2$$

with $\mathit{Erase}(E_1\{{}^{c_1,\ldots,c_n}/_{b_1,\ldots,b_m}\} \mid E_2) = P_1\{{}^{c_1,\ldots,c_n}/_{b_1,\ldots,b_m}\} \mid P_2$.

For the case when the last rule used was R-Par, use the induction hypothesis.

If the last rule used was R-Res, then $E = (\boldsymbol{\nu}\, a{:}S)\, E'$ and $\mathit{Erase}(E) = (\boldsymbol{\nu}\, a{:}S)\, \mathit{Erase}(E')$. By definition of ok_Γ, we have $\mathit{ok}_{\Gamma,a:S}(E')$. The result follows by induction.

Finally, for the case of the rule R-Eqv, use Lemmas 3.3.1 and 3.3.2 and the induction assumption. □

4 Lambda calculus encodings

The use of I/O tags arises naturally in many applications. As an example, we consider Milner's encodings [Mil90] of lazy λ-calculus and call-by-value λ-calculus [Abr89, Plo75]. For each encoding, we first present Milner's original version and then assign sorts to the channels in the translation.

We abbreviate the process expression $(\boldsymbol{\nu}\, m)\,(P \mid !m(\tilde{x})\,.\,R)$ as $P\{m(\tilde{x}) := R\}$. Intuitively, the process R represents a "local environment" for P and m is a "pointer" that allows P to access this local environment; alternatively, we can think of R as a resource with owner P and m as a trigger with which a copy of the resource may be activated. We give $\{m(\tilde{x}) := R\}$ tighter precedence than $\mid$; thus $Q \mid P\{m(\tilde{x}) := R\}$ stands for $Q \mid (P\{m(\tilde{x}) := R\})$.

4.1 The lazy λ-calculus

In Abramsky's lazy lambda calculus [Abr89], only leftmost redexes can be reduced. There are two reduction rules:

$$(\beta)\quad (\lambda x.M)N \Longrightarrow M\{N/x\} \qquad\qquad (App)\ \frac{M \Longrightarrow M'}{MN \Longrightarrow M'N}$$

The core of the encoding of lazy λ-terms into π-calculus is the translation of function application. As usual when translating λ-calculus into a process calculus, application becomes a particular form of parallel composition and β-reduction is modeled by interaction. Since the syntax of the first-order π-calculus only allows for the transmission of names along channels, the communication of a term is simulated by the communication of a *trigger* for it.

In the pure λ-calculus, every term denotes a function. When supplied with an argument, it yields another function (which in turn is waiting for an argument). Analogously, the translation of a λ-term M is a process with an *argument port* p. It rests dormant until it receives along p a trigger x for its argument and a new port q; given these, it evolves to a new process with argument port q. We may also think of p as the "location" of M, since p is the unique port along which M interacts with its environment.

$$\begin{array}{rcll}
\mathcal{L}[\![\lambda x.M]\!]\langle p\rangle & = & p(x,q)\,.\,\mathcal{L}[\![M]\!]\langle q\rangle & \\
\mathcal{L}[\![x]\!]\langle p\rangle & = & \overline{x}\langle p\rangle & \\
\mathcal{L}[\![MN]\!]\langle p\rangle & = & (\boldsymbol{\nu}\, q)\,(\mathcal{L}[\![M]\!]\langle q\rangle \mid (\overline{q}\langle y,p\rangle\{y(r) := \mathcal{L}[\![N]\!]\langle r\rangle\})) & \qquad (y \text{ fresh})
\end{array}$$

Using the sorts

$$\begin{array}{rcl}
S_a & = & (S_t, S_a)^{\mathsf{r}} \\
S_t & = & (S_a)^{\mathsf{w}}
\end{array}$$

we can annotate the translation so that the intended use of argument and trigger ports is enforced statically. Whenever a trigger is communicated along a channel, the receiver may only use it for output; similarly, when an argument port is communicated, the receiver may use it only for input. Using the sorted abbreviation $P\{m(x{:}S_t) := R\} = (\boldsymbol{\nu}\, m{:}(S_t)^{\mathsf{b}})\,(P \mid !(m(x{:}S_t)\,.\,R))$, the translation becomes

$$\begin{array}{rcl}
\mathcal{L}[\![\lambda x.M]\!]\langle p\rangle & = & p(x{:}S_t,\, q{:}S_a)\,.\,\mathcal{L}[\![M]\!]\langle q\rangle \\
\mathcal{L}[\![x]\!]\langle p\rangle & = & \overline{x}\langle p\rangle \\
\mathcal{L}[\![MN]\!]\langle p\rangle & = & (\boldsymbol{\nu}\, q{:}(S_t,\, S_a)^{\mathsf{b}})\,(\mathcal{L}[\![M]\!]\langle q\rangle \mid (\overline{q}\langle y,p\rangle\{y(r{:}S_a) := \mathcal{L}[\![N]\!]\langle r\rangle\}))
\end{array}$$

where the free names p and x have sorts S_a and S_t, respectively.

4.2 The call-by-value λ-calculus

In call-by-value λ-calculus, on the other hand, reductions may only occur when the argument is a value, i.e. an abstraction. The reduction relation used by Milner in [Mil90] is described by the rules β_v, App_L, App_R:

$$(\beta_v)\quad (\lambda x.M)(\lambda y.N) \Longrightarrow M\{\lambda y.N/x\} \qquad (App_L)\ \frac{M \Longrightarrow M'}{MN \Longrightarrow M'N} \qquad (App_R)\ \frac{N \Longrightarrow N'}{MN \Longrightarrow MN'}$$

By contrast with the lazy λ-calculus, in the application MN, also N may evolve, until both M and N have become λ-abstractions. But at this point M's behavior deviates from N's: the former *receives* a value, whereas the latter is *passed as* a value.

Let us see how all this is reflected in the π-calculus encoding. The two arguments $\mathcal{V}[\![M]\!]\langle q\rangle$ and $\mathcal{V}[\![N]\!]\langle r\rangle$ of an application are run in parallel. They signal to the "arbiter process" App when they have reduced to an abstraction by transmitting to App, along the ports q and r, a *trigger* for their *value-body*. Then, the dichotomy in the behavior of the two λ-abstractions is solved by the arbiter, which forces the correct interaction between the two value-bodies. The core of the protocol executed by App is the action on an internal channel v, by which the abstraction M comes to know its arguments. We call v a *pivot*. (In the lazy λ-calculus encoding, the role of the pivot names was played by the argument port names.) In the original version of his paper on Functions as Processes [Mil90], Milner presented two candidates for the encoding of a version of call-by-value λ-calculus [Plo75]. They follow the same idea of translation, but with a technical difference in the rule for variables.

<u>First candidate ($\mathcal{V}$):</u>

$$
\begin{aligned}
\mathcal{V}[\![\lambda x.M]\!]\langle p\rangle &= \overline{p}\langle y\rangle\{y(w) := w(x,q)\,.\,\mathcal{V}[\![M]\!]\langle q\rangle\}\\
\mathcal{V}[\![x]\!]\langle p\rangle &= \overline{p}\langle y\rangle\{y(w) := \overline{x}\langle w\rangle\}\\
\mathcal{V}[\![MN]\!]\langle p\rangle &= (\boldsymbol{\nu}\, q,r)\left(\mathcal{V}[\![M]\!]\langle q\rangle \mid \mathcal{V}[\![N]\!]\langle r\rangle \mid App\langle p,q,r\rangle\right)\\
&\ \text{where } App(p,q,r) = q(x)\,.\,(\boldsymbol{\nu}\, v)\,\overline{x}\langle v\rangle\,.\,r(y)\,.\,\overline{v}\langle y,p\rangle
\end{aligned}
$$

<u>Second candidate ($\mathcal{V}'$):</u>

The rules for application and λ-abstraction are the same as for $\mathcal{V}$; the rule for variables is:

$$\mathcal{V}'[\![x]\!]\langle p\rangle = \overline{p}\langle x\rangle$$

The encoding $\mathcal{V}'$ is more efficient than $\mathcal{V}$. In $\mathcal{V}$, a λ-calculus variable gives rise to a one-place buffer (the local environment $\{y(w) := \overline{x}\langle w\rangle\}$). As the computation proceeds, these buffers are chained together, gradually increasing the number of steps necessary to simulate a β-reduction. This phenomenon does not occur in $\mathcal{V}'$, where a variable disappears after it is used. But the proofs in [Mil90] were conducted with $\mathcal{V}$; the analysis of $\mathcal{V}'$ seemed more difficult and was left open. Indeed, Sangiorgi showed [San92] that, in the standard π-calculus, β-reduction is *not* valid for $\mathcal{V}'$, i.e. $\mathcal{V}'[\![(\lambda x.M)\lambda y.N]\!]\langle p\rangle$ and $\mathcal{V}'[\![M\{^{\lambda y.N}/x\}]\!]\langle p\rangle$ may not be behaviorally equivalent[1] (the same counterexample used in [San92] applies to the calculus used in this paper, which is a subcalculus of the one in [San92]). Nevertheless $\mathcal{V}'$ yields a precise operational correspondence between λ-terms and their process encodings, and intuitively one expects $\mathcal{V}'$ to be correct.

The problems with $\mathcal{V}'$ are caused by the communication of triggers introduced in the translation of variables: the external environment might obtain a trigger and then use it improperly, i.e. in input position. By adding I/O tags, we avoid these misbehavours and $\mathcal{V}'$ becomes correct. We shall prove this fact in Section 5.4, after we have developed the necessary technical machinery. For the moment, let us look at the sorts of the names appearing in $\mathcal{V}$ and $\mathcal{V}'$ (the sorts are the same in both encodings). There are three kinds of names: besides argument ports like p, q, and r and triggers like x and y, both of which were already present in the lazy λ-encoding, we also have pivots like v and w. An argument port is used to send a trigger; a trigger is used to activate a value-body with its appropriate pivot; the pivot of an abstraction $\lambda x.M$ is used to receive a trigger for the λ-input together with the argument port of M. Thus the sorts S_a, S_t, and S_p of argument ports, triggers, and pivots are:

$$
\begin{aligned}
S_a &= (S_t)^{\mathsf{w}}\\
S_t &= (S_p)^{\mathsf{w}}\\
S_p &= (S_t, S_a)^{\mathsf{r}}
\end{aligned}
$$

As for the lazy λ-calculus, the outermost tag of the sort of the names bound by a restriction is b; therefore, such argument ports, triggers, and pivots have sorts $(S_t)^{\mathsf{b}}$, $(S_p)^{\mathsf{b}}$ and $(S_t, S_a)^{\mathsf{b}}$, respectively.

[1] The final version of [Mil90], which appeared in the *Joural of Mathematical Structures*, was written after the results in [San92] were known and presents only the encoding $\mathcal{V}$.

5 Behavioral equivalence

In this section we examine the effect of our sorting discipline on the semantics of processes.

5.1 Contexts

A *process context* (or just *context*) C is a process expression with a hole in it. We write $C[P]$ for the expression obtained by filling the hole with P. The hole itself is written $[\cdot]$. For the typing of contexts, we add the following rule to those in Section 2.5.

$$\Gamma \vdash [\cdot] \;:\; \circ \qquad \text{(HOLE)}$$

We shall employ contexts when considering the behavioral equivalence of processes. For this, we first have to understand when placing a process into a context is sensible. We start by looking at the combination of a context $C[\cdot]$ of sorting Γ and a process P of sorting Δ from a *syntactic* point of view. We want a condition on Δ that guarantees $\Gamma \vdash C[P] \;:\; \circ$. We do not exactly want the bindings in Γ to be subtypes of those in Δ, i.e. $\Gamma <_{\mathrm{E}} \Delta$, since there might be binders (restriction, input prefixes) in $C[\cdot]$ whose scope includes the hole. For instance, if $C[\cdot] = a(c : S_c) \,.\, (\boldsymbol{\nu}\, b : S_b)\, [\cdot]$ and $\Gamma = \{a : S_a\}$, then P must respect the sorts S_b and S_c for the names b and c in addition to S_a for a. This suggests that the *external* sorting Γ of $C[\cdot]$ should be distinguished from an *internal* one Γ_C, which describes the sorts of all the names that are "in scope" at the point where C's hole occurs. The sorting Γ_C, called the *extension of* Γ *by* C, is obtained from Γ by scanning the context from the outside towards the hole and keeping track of the binders that are met along the way. So in the previous example, we have $\Gamma_C = \{a : S_a, b : S_b, c : S_c\}$.

5.1.1. Definition: C is said to be a (Γ,Δ)-context if $\Gamma \vdash C \;:\; \circ$ and $\Gamma_C <_{\mathrm{E}} \Delta$.

5.1.2. Proposition: If C is a (Γ,Δ)-context and $\Delta \vdash P \;:\; \circ$, then $\Gamma \vdash C[P] \;:\; \circ$.

Proof: We know that $\Delta \vdash P \;:\; \circ$ and, since C is a (Γ,Δ)-context, $\Gamma_C <_{\mathrm{E}} \Delta$. This, by Corollary 2.6.6, gives $\Gamma_C \vdash P \;:\; \circ$. Now a derivation for $\Gamma \vdash C[P] \;:\; \circ$ can be constructed from the derivations of $\Gamma_C \vdash P \;:\; \circ$ and $\Gamma \vdash C[\cdot] \;:\; \circ$ by replacing the sole instance of the rule HOLE by the derivation of $\Gamma_C \vdash P \;:\; \circ$. □

We conjecture that $\Gamma_C <_{\mathrm{E}} \Delta$ is also a *necessary* condition for having $\Gamma \vdash C[P] : \circ$ whenever $\Delta \vdash P \;:\; \circ$.

Note that a name could be declared, for instance, as input-only in Δ and yet appear in $C[\cdot]$ in an output position since, in Definition 5.1.1, the internal sorting Γ_C is an (extended) *sub*-sorting of Δ. This situation occurs in the printer example from the introduction: the printer uses the name p only for input, but the surrounding context may perform outputs on p. Indeed, the weaker requirement on the context is clearly essential, for otherwise no communications along p would ever take place.

The composition of a context of sorting Γ and a process of sorting Δ is syntactically meaningful when C is a (Γ,Δ)-context; now we want to check that this composition is also *semantically* meaningful. We know that P behaves correctly with respect to $C[\cdot]$ and Γ, because $\Gamma \vdash C[P] : \circ$ and we have seen that our typing system is sound. More delicate is the opposite implication, namely that the expected use of names emitted by P, as specified in Δ, is consistent with their actual use in C, which is constrained by Γ. (We suppose that the names emitted by P are free; the case in which they are bound is similar.) Formulating this claim in general terms seems difficult, but it can be substantiated in some important special cases.

For example, suppose $P = \overline{a}\langle b\rangle \,.\, P'$ and $\mathcal{I}_1(\Delta(a)) = \mathsf{w}$. Then, according to Δ the name b should be used by its recipient for output only. Suppose that $a(x{:}T) \,.\, Q$ is the subterm of $C[\cdot]$ with which P communicates. To respect P's expectations, the name x should be used by Q for output only, i.e. it should be the case that $\mathcal{I}(T) = \mathsf{w}$. More generally, we would like to guarantee that if $\mathcal{I}_1(\Delta(a)) = J$, then $J \leq \mathcal{I}(T)$.

5.1.3. Proposition: Suppose $\Delta \vdash \overline{a}\langle y\rangle \,.\, P \;:\; \circ$ and $\Gamma \vdash a(x{:}T) \,.\, Q \;:\; \circ$, with $\Gamma <_{\mathrm{E}} \Delta$. Then

$$\mathcal{I}(obj_1(\Delta(a))) \leq \mathcal{I}(T).$$

Proof: We show $obj_1(\Delta(a)) \leq T$, from which the result follows by Lemma 2.4.12. First, from the typing rules, $\Gamma \vdash a(x{:}T) \,.\, Q \;:\; \circ$ gives

$$\Gamma(a) \leq (T)^{\mathsf{r}}. \qquad (9)$$

Similarly, from $\Delta \vdash \overline{a}\langle y\rangle \,.\, P \;:\; \circ$,

$$\Delta(a) \leq (S)^{\mathsf{w}} \qquad (10)$$

for some S. Finally, from $\Gamma <_{\mathrm{E}} \Delta$, we have

$$\Gamma(a) \leq \Delta(a). \tag{11}$$

Now, using (10) and (11) and the transitivity of $\leq$, we obtain

$$\Gamma(a) \leq (S)^{\mathsf{w}}. \tag{12}$$

By Lemma 2.4.11(1), (9) and (12) imply that $\Gamma(a) \lesssim (obj(\Gamma(a)))^{\mathsf{b}}$ and

$$S \leq obj_1(\Gamma(a)) \leq T. \tag{13}$$

Replacing $\Gamma(a)$ with $(obj_1(\Gamma(a)))^{\mathsf{b}}$ in (11), and using (10), we get $(obj_1(\Gamma(a)))^{\mathsf{b}} \leq \Delta(a) \leq (S)^{\mathsf{w}}$, which by Lemma 2.4.11(2) gives

$$S \leq obj_1(\Delta(a)) \leq obj_1(\Gamma(a)). \tag{14}$$

Finally, by (14) and (13) (ignoring S) and transitivity, $obj_1(\Delta(a)) \leq T$, which concludes the proof. $\square$

5.2 Barbed bisimulation and congruence

The most popular way of defining behavioral equivalence on processes is via the notion of bisimulation. In [MS92, San92], Milner and Sangiorgi propose *barbed bisimulation* as a tool for uniformly defining bisimulation-based equivalences in different calculi. The portability of the definition is useful when studying a new calculus or a refinment of an existing one, as we are doing here: barbed bisimulation immediately suggests an interesting congruence for the newborn system.

The definition of barbed bisimulation uses the reduction relation of the calculus along with an observation predicate $\downarrow_a$ for each port a, which detects the possibility of performing a communication with the external environment along a. Thus, in the π-calculus, $P \downarrow_a$ holds if P has a prefix $a(x_1, \ldots, x_n)$ or $\overline{a}\langle x_1, \ldots, x_n\rangle$ which is not underneath another prefix and not in the scope of a restriction on a. For example, if $P = (\boldsymbol{\nu}\, c)\,(\overline{c}\langle b\rangle \mid a(x)\,.\,d(y))$, then $P \downarrow_a$, but not $P \downarrow_c$, $P \downarrow_b$, or $P \downarrow_d$.

By itself, barbed bisimulation is a rather coarse relation. Better discriminating power is achieved by considering the induced congruence, called *barbed congruence*. It can be shown [San92] that barbed congruence coincides in both CCS and π-calculus with the standard bisimilarity congruences, at least if the grammar of these languages is sufficiently rich (for instance, in [San92] infinite recursive definitions of agents and agents with an infinite number of free names are allowed).

Of course, in a sorted calculus, the processes being compared must obey the same sorting and the contexts employed must be compatible with this sorting.

5.2.1. Definition: A relation $\Re \subseteq Pr_\Delta \times Pr_\Delta$ is a *barbed Δ-bisimulation* if $(P,Q) \in \Re$ implies:

1. if $P \longrightarrow P'$ then $Q \longrightarrow Q'$ and $(P',Q') \in \Re$;
2. if $Q \longrightarrow Q'$ then $P \longrightarrow P'$ and $(P',Q') \in \Re$;
3. for each channel a, $P \downarrow_a$ iff $Q \downarrow_a$.

Two processes P and Q are *barbed Δ-bisimilar*, written $P \stackrel{\bullet}{\sim}_\Delta Q$, if $(P,Q) \in \Re$, for some barbed Δ-bisimulation $\Re$. P and Q are *barbed Δ-congruent*, written $P \sim_\Delta Q$, if, for each sorting Γ and (Γ,Δ)-context C, we have $C[P] \stackrel{\bullet}{\sim}_\Gamma C[Q]$.

Barbed bisimulation requires a quantification over all contexts. The following theorem shows a way to lighten this requirement: it is enough to test processes using parallel composition and substitution.

5.2.2. Lemma: Let P_1 and P_2 be two processes obeying the sorting Δ. Suppose that for every sorting Γ, substitution σ, and process Q such that

1. $\Gamma <_{\mathrm{E}} \Delta$,
2. if $\sigma = \{\tilde{c}/\tilde{b}\}$, then $\Gamma(\tilde{c}) \leq \Gamma(\tilde{b})$, and
3. Q obeys Γ,

it holds that

$$Q \mid P_1\sigma \;\dot{\sim}_\Gamma\; Q \mid P_2\sigma.$$

Then $P_1 \sim_\Delta P_2$.

Proof: See Appendix A. □

The weak version of the equivalences, where one abstracts away from the length of the reductions in two matching actions, is obtained in the standard way. Let $\Longrightarrow$ be the reflexive and transitive closure of $\longrightarrow$, and let $\Downarrow_a$ be $\Longrightarrow \downarrow_a$, the composition of the two relations. Then *weak barbed Δ-bisimulation*, written $\dot{\approx}_\Delta$, is defined by replacing in Definition 5.2.1 the transition $Q \longrightarrow Q'$ with $Q \Longrightarrow Q'$ and the predicate $Q \downarrow_a$ with $Q \Downarrow_a$. *Weak barbed Δ-congruence*, written $\approx_\Delta$, is defined by replacing $\dot{\sim}_\Gamma$ with $\dot{\approx}_\Gamma$. We conjecture that the counterpart of Lemma 5.2.2 for the weak case is also true.

In the basic π-calculus (i.e., without I/O tags), the only restriction on the choice of Γ and C in the definition of barbed congruence is that arities of channels must be respected; in our calculus, I/O information is also taken into account. This places a tighter constraint on the set of contexts that are regarded as legal observers. In consequence, the equivalence itself becomes coarser.

Processes of the basic π-calculus correspond, in our calculus, to processes whose I/O tags are all b. Below, we write $\Gamma \vdash P, Q : \circ$ if both $\Gamma \vdash P : \circ$ and $\Gamma \vdash Q : \circ$.

5.2.3. Lemma: Suppose that the sorting Γ and the processes P and Q differ from the sorting Γ' and the processes P' and Q', respectively, on the I/O tags only, and that both $\Gamma \vdash P, Q : \circ$ and $\Gamma' \vdash P', Q' : \circ$. Then we have $P \;\dot{\sim}_\Gamma\; Q$ iff $P' \;\dot{\sim}_{\Gamma'}\; Q'$ and $P \;\dot{\approx}_\Gamma\; Q$ iff $P' \;\dot{\approx}_{\Gamma'}\; Q'$. □

Proof: The I/O tags do not play a part in reductions of well-typed processes; hence, bisimularity (as opposed to congruence, which we consider next) is not affected by changing tags. □

5.2.4. Theorem: Suppose that $\Delta \vdash P, Q : \circ$ and that the sorting Δ_{b} and the processes P_{b} and Q_{b} are obtained from the sorting Δ and the processes P and Q by replacing all I/O tags in Δ, P, and Q with the tag b. Then

1. $\Delta_{\mathsf{b}} \vdash P_{\mathsf{b}}, Q_{\mathsf{b}} : \circ$,
2. $P_{\mathsf{b}} \sim_{\Delta_{\mathsf{b}}} Q_{\mathsf{b}}$ implies $P \sim_\Delta Q$, and $P_{\mathsf{b}} \approx_{\Delta_{\mathsf{b}}} Q_{\mathsf{b}}$ implies $P \approx_\Delta Q$.

Proof: Assertion (1) is straightforward. For (2), we only consider $\sim_{\Delta_{\mathsf{b}}}$; the case of $\approx_{\Delta_{\mathsf{b}}}$ is analogous. By definition, $P \sim_\Delta Q$ holds if, for all (Γ,Δ)-contexts C, we have $C[P] \;\dot{\sim}_\Gamma\; C[Q]$. Let C_{b} and Γ_{b} be the context and sorting obtained from C and Γ by replacing all I/O tags with b. By assertion (1), $\Gamma_{\mathsf{b}} \vdash C_{\mathsf{b}}[P_{\mathsf{b}}], C_{\mathsf{b}}[Q_{\mathsf{b}}] : \circ$. Since $P_{\mathsf{b}} \sim_{\Delta_{\mathsf{b}}} Q_{\mathsf{b}}$, and C_{b} is a $(\Gamma_{\mathsf{b}},\Delta_{\mathsf{b}})$-context, we have $C_{\mathsf{b}}[P_{\mathsf{b}}] \;\dot{\sim}_{\Gamma_{\mathsf{b}}}\; C_{\mathsf{b}}[Q_{\mathsf{b}}]$. Hence $C[P] \;\dot{\sim}_\Gamma\; C[Q]$ follows from Lemma 5.2.3. □

Thus, any equivalence that can be proved in the basic π-calculus, e.g., the equivalence between $!P$ and $!P \mid !P$, can be lifted up to our calculus. We shall see in the next section that the converse does not hold: inequivalent processes in the basic π-calculus may be made equivalent by replacing b tags with r or w.

5.3 The replicator theorems

In Section 4, we introduced the notation $P\{m(\tilde{x}) := R\}$ for "local environments." Perhaps the most useful theorems about local environments are the ones showing that they distribute over parallel composition and replication. For instance, these theorems play a pivotal role in the proof of validity of β-reduction for Milner's encodings of λ-calculus [Mil90], as well as in the proof of representability of higher-order π-calculus in first-order π-calculus [San92]. However, in the basic π-calculus the theorems must be accompanied by a fairly heavy side condition on the use of names in P, Q, and R, namely that m may occur free only as the channel along which an output occurs. To see why, take:

$$P_1 = (\overline{b}\langle m\rangle | Q)\{m := R\} \qquad P_2 = \overline{b}\langle m\rangle\{m := R\} | Q\{m := R\}$$

These processes are not equivalent in the basic π-calculus. Intuitively, the environment external to P_1 can receive m along b and then use it in input position to interfere with an attempt by Q to trigger R. This is not possible in P_2, where Q has its own private access to R.

Of course, the difference between P_1 and P_2 can only be observed using a context that ignores the intended role of m as a trigger; to validate the desired equivalence, such a context should be considered inadmissible. In our sorted calculus, this can be achieved by giving m a sort such that it can only be used for output, as we did in our encodings of λ-calculus in Section 4.

5.3.1. Definition: Let P be a subexpression of a process expression $C[P]$, where $\Gamma \vdash C[P] : \circ$. We say that *$m$ is used (only) as a trigger in P* if $\Delta \vdash P : \circ$, where Δ is obtained from Γ_C, the internal sorting of P's position in C, by replacing the outermost tag of m's sort with w.

In the following, when using the above definition we sometimes omit the judgement $\Gamma \vdash C[P] : \circ$; the context C in question will always be clear. Moreover, we maintain the notation $P\{m(\tilde{x}) := R\}$ in our sorted calculus, eliding the sorts of m and $\tilde{x}$.

5.3.2. Theorem: [First replicator theorem] If $\Gamma \vdash (P \,|\, Q)\{m(\tilde{x}) := R\} : \circ$ and m is used as a trigger in P, Q, and R, then

$$(P \,|\, Q)\{m(\tilde{x}) := R\} \sim_\Gamma P\{m(\tilde{x}) := R\} \mid Q\{m(\tilde{x}) := R\}.$$

5.3.3. Theorem: [Second replicator theorem] If $\Gamma \vdash (!P)\{m(\tilde{x}) := R\} : \circ$ and m is used as a trigger in P and R, then

$$(!P)\{m(\tilde{x}) := R\} \sim_\Gamma !(P\{m(\tilde{x}) := R\}).$$

These versions of the replicator theorems are substantially stronger than the original ones, since they allow the trigger m to be transmitted as value, as long as each such transmission places an appropriate constraint on the possible use of m by the receiver. The proofs of both theorems can be found in Appendix B.

5.4 Correctness of the encoding $\mathcal{V}'$

Using our improved replicator theorems, we can show that in our calculus β-reduction is valid for Milner's encoding $\mathcal{V}'$ of the call-by-value λ-calculus, presented in Section 4.2. With the introduction of I/O tags, the output of a trigger in the translation of λ-variables is rendered safe, since the trigger is tagged w and so cannot be used by the recipient for input. Recall that S_a and S_t are the sorts for argument ports and triggers of the encoding $\mathcal{V}'$, as defined in Section 4.2.

5.4.1. Theorem: [Validity of β-reduction in $\mathcal{V}'$] Suppose $x_1, \ldots, x_n$ is the set of free variables in $(\lambda x.M)\lambda y.N$, and let Δ be the sorting $x_1 : S_t, \ldots, x_n : S_t, p : S_a$. Then

$$\mathcal{V}'[\![(\lambda x M)\lambda y.N]\!]\langle p\rangle \approx_\Delta \mathcal{V}'[\![M\{{}^{\lambda y.N}/x\}]\!]\langle p\rangle.$$

Proof: See Appendix C. □

6 Extensions

6.1 Matching and Summation

Many presentations of the π-calculus include a *summation* operator $P + Q$, which can be used to select one element from a set of pending communications and abort the others, and a *matching* operator $[a = c]P$, which allows P to proceed only if a and c are the same channel. (It seems that both operators may often be avoided in practice, although they play an important theoretical role, for instance, in axiomatisations of behavioral equivalences [San93, PS93].) Our basic results should extend straightforwardly to summation, whose behavior under typing is exactly like that of parallel composition.

Matching, however, is more problematic, since adding it in unrestricted form would destroy some of the basic results presented in Section 5. An appropriate constraint can be obtained from the typing of the other constructs we have considered: A term $a(\tilde{x}) \,.\, P \,|\, \overline{c}\langle\tilde{y}\rangle \,.\, Q$ may evolve to $P \,|\, \{\tilde{y}/\tilde{x}\}Q$ only if a and c are the same name, which thus appears in the term both in input and in output position. This analogy suggests that we should allow a match expression $[a = c]P$ only if both a and c have b as I/O tag, i.e. formally to impose the rule:

$$\frac{\mathcal{I}(\Gamma(a)) = \mathcal{I}(\Gamma(c)) = \mathsf{b} \qquad \Gamma \vdash P : \circ}{\Gamma \vdash [a = c]P : \circ} \qquad \text{(T-MATCH)}$$

Another safe solution would be to extend our set of basic sort annotations so as to distinguish channels that may be compared for identity from those that may not.

6.2 Higher-Order π-calculus

A more significant extension involves adding higher-order communication — communication of processes and abstraction of processes on both channels and processes. Following Sangiorgi [San92], we can enrich our language of sorts to include descriptions of channels carrying processes — e.g. $(\circ)^{\mathsf{b}}$ — channels carrying processes abstracted on processes — e.g. $(\circ{\rightarrow}\circ)^{\mathsf{b}}$ — and so on. The subsort relation now generates a natural subtype relation using the standard subtyping rule for function types. This extension essentially amounts to adjoining to our system a standard functional type system along the lines of Cardelli's simply typed λ-calculi with subtyping [Car86].

6.3 Polymorphism

A general theme in this paper has been the observation that a type system for a process calculus can be presented using concepts familiar from the literature on typed λ-calculi. Carrying this program a step further leads us to wonder what role the *polymorphism* found in ML [Mil78] or the polymorphic λ-calculus [Gir72, Rey74] might play in typing for processes. One possibility is to extend channel sorts so that each channel is thought of as carrying a tuple of both types and values:

			channel sorts
S	$::=$	$(A_1, \ldots, A_m; S_1, \ldots, S_n)^I$	tagged tuple of type variables and sorts
	$\mid$	A	sort variable
	$\mid$	$\mu A.\ S$	recursive sort

In a channel sort of the form $(A_1, \ldots, A_m; S_1, \ldots, S_n)^I$, the type parameters $A_1, \ldots, A_n$ may appear in the sorts $S_1, \ldots, S_n$ — in other words, the tuples passed along the channel may be thought of as elements of a generalized existential type. Thus, for example, the sorting $a : (A;\, (A)^{\mathsf{w}},\, A)^{\mathsf{r}}$ precisely describes the constraints respected by the process $a(A;\, x{:}(A)^{\mathsf{w}},\, y{:}A)\,.\,\overline{x}\langle y\rangle\,.\,\mathbf{0}$, which reads a pair of channels from a and outputs the second along the first.

A similar polymorphic extension of π-calculus sorting has been proposed independently by Turner; it will be described in his forthcoming Ph.D. thesis [Tur94].

Acknowledgements

This research was begun while both authors were visitors at INRIA-Roquencourt (in projects Formel and Para, respectively), and was partially supported by ESPRIT Basic Research Action TYPES and CONFER and by the British Science and Engineering Research Council.

Our ideas were influenced by many conversations with Robin Milner and David N. Turner.

A Appendix: Proof of Lemma 5.2.2

In this appendix and the ones that follow, we omit sorting annotations when they are irrelevant or can easily be recovered from the context.

The derivation proof of an interaction $P \longrightarrow Q$ uniquely determines the two subterms $a(\tilde{b}) . P_1$ and $\overline{a}\langle\tilde{c}\rangle . P_2$ of P whose outermost prefixes are those consumed in the interaction; we call these the *interacting subterms* of the derivation.

A useful fact for reasoning with reduction semantics is that two interactions whose derivations use the same interacting subterms yield processes identical modulo structural congruence. This is the content of Lemma A.5. The results preceding Lemma A.5 are needed for its proof. We start by noticing that the derivation of an interaction can be put into a standard form in which the rule R-EQV is used at most once, at the end (this lemma also appears in [GR93]).

A.1. Lemma: Suppose $Q \longrightarrow Q'$. Then there is a derivation of $Q \longrightarrow Q'$ in which R-EQV appears only as the outermost step, or not at all.

In other words, a derivation $Q \longrightarrow Q'$ can be written as

$$Q \equiv (\boldsymbol{\nu}\,\tilde{b})\left(a(\tilde{x}) . P_1 \,|\, \overline{a}\langle\tilde{y}\rangle . P_2 \,|\, R\right) \longrightarrow (\boldsymbol{\nu}\,\tilde{b})\left(P_1\{\tilde{y}/\tilde{x}\} \,|\, P_2 \,|\, R\right) \equiv Q'$$

Proof: By induction on the original derivation of $Q \longrightarrow Q'$, using the facts that the rule R-EQV commutes with R-PAR and R-RESTR and that successive occurrences of R-EQV can be merged into one. □

We can strengthen this result by requiring that $a(\tilde{x}) . P_1$ and $\overline{a}\langle\tilde{y}\rangle . P_2$ be actual subterms of Q (i.e., they have not been modified with $\equiv$) and that all names in $\tilde{b}$ actually occur free in these subterms.

A.2. Lemma: Suppose that a derivation of $Q \longrightarrow Q'$ has $a(\tilde{x}) . P_1$ and $\overline{a}\langle\tilde{y}\rangle . P_2$ as interacting subterms. Then there is a derivation of $Q \longrightarrow Q'$ in which the last step uses the rule R-EQV with premises of the form

$$Q \equiv (\boldsymbol{\nu}\,\tilde{b})\left(a(\tilde{x}) . P_1 \,|\, \overline{a}\langle\tilde{y}\rangle . P_2 \,|\, R\right) \longrightarrow (\boldsymbol{\nu}\,\tilde{b})\left(P_1 \,|\, P_2\{\tilde{y}/\tilde{x}\} \,|\, R\right) \equiv Q'$$

and $\tilde{b} \subseteq \mathrm{fn}(a(\tilde{x}) . P_1) \cup \mathrm{fn}(\overline{a}\langle\tilde{y}\rangle . P_2)$.

Proof: By A.1 and the structural equivalence rules. □

A.3. Lemma: If $(\boldsymbol{\nu}\,b)\,P \equiv (\boldsymbol{\nu}\,b)\,Q$ (note that our conventions about names also imply that this proof makes no use of alpha-conversion on b), then also $P \equiv Q$.

Proof: A derivation of $P \equiv Q$ can be constructed from the derivation of $(\boldsymbol{\nu}\,b)\,P \equiv (\boldsymbol{\nu}\,b)\,Q$ by simply stripping off those steps which manipulate the restriction on b. □

A.4. Lemma: Suppose $\alpha . P \,|\, Q \equiv \alpha . P \,|\, R$. Then $Q \equiv R$.

Proof: Since none of the structural equivalence rules mention prefixes, it is easy to transform a derivation of $\alpha . P \,|\, Q \equiv \alpha . Q \,|\, R$ into a derivation of $\mathbf{0} \,|\, Q \equiv \mathbf{0} \,|\, R$. Hence $Q \equiv \mathbf{0} \,|\, Q \equiv \mathbf{0} \,|\, R \equiv R$. □

A.5. Lemma: Suppose that the derivations of the interactions $Q \longrightarrow Q_1$ and $Q \longrightarrow Q_2$ use the same interacting subterms. Then $Q_1 \equiv Q_2$.

Proof: Let $a(\tilde{x}) . P_1$ and $\overline{a}\langle\tilde{y}\rangle . P_2$ be the interacting subterms used in the two reductions. By Lemma A.2 the derivation of $Q \longrightarrow Q_i$, $i = 1, 2$, can be written as

$$Q \equiv (\boldsymbol{\nu}\,\tilde{b})\,(a(\tilde{x}) . P_1 \,|\, \overline{a}\langle\tilde{y}\rangle . P_2 \,|\, R_i) \longrightarrow (\boldsymbol{\nu}\,\tilde{b})\,(P_1\{\tilde{y}/\tilde{x}\} \,|\, P_2 \,|\, R_i) \equiv Q_i$$

with $\tilde{b} \subseteq \mathrm{fn}(a(\tilde{x}) . P_1, \overline{a}\langle\tilde{y}\rangle . P_2)$. From $(\boldsymbol{\nu}\,\tilde{b})\,(a(\tilde{x}) . P_1 \,|\, \overline{a}\langle\tilde{y}\rangle . P_2 \,|\, R_1) \equiv (\boldsymbol{\nu}\,\tilde{b})\,(a(\tilde{x}) . P_1 \,|\, \overline{a}\langle\tilde{y}\rangle . P_2 \,|\, R_2)$, Lemma A.3, and Lemma A.4, we get $R_1 \equiv R_2$, hence

$$Q_1 \equiv (\boldsymbol{\nu}\,\tilde{b})\,(P_1\{\tilde{y}/\tilde{x}\} \,|\, P_2 \,|\, R_1) \equiv (\boldsymbol{\nu}\,\tilde{b})\,(P_1\{\tilde{y}/\tilde{x}\} \,|\, P_2 \,|\, R_2) \equiv Q_2.$$

□

A.6. Lemma: Suppose that $\alpha. P$ is a subterm of Q that is not underneath a prefix. Then there are $\tilde{b}$ and R such that $Q \equiv (\boldsymbol{\nu}\, \tilde{b})\,(\alpha. P \mid R)$.

Proof: We prove the lemma by induction on the structure of Q. In the base case, $Q = \alpha. P$ and the result is trivial, for $R = \mathbf{0}$ and $\tilde{b} = \{\}$. Now the induction cases:

Case $Q \equiv Q_1 \mid Q_2$.

Suppose that $\alpha. P$ is in Q_1; the case in which $\alpha. P$ is in Q_2 is analogous. By induction, $Q_1 \equiv (\boldsymbol{\nu}\, \tilde{b})\,(R' \mid \alpha. P)$. Therefore we have

$$Q = Q_1 \mid Q_2 \equiv (\boldsymbol{\nu}\, \tilde{b})\left(R' \mid \alpha. P\right) \mid Q_2 \equiv (\boldsymbol{\nu}\, \tilde{b})\left((R' \mid Q_2) \mid \alpha. P\right).$$

Case $Q = !\, Q'$.

$Q \equiv Q' \mid !\, Q'$ and by induction $Q' \equiv (\boldsymbol{\nu}\, \tilde{b})\,(R' \mid \alpha. P)$. Proceed as in the previous case.

Case $Q = (\boldsymbol{\nu}\, c)\, Q'$.

Easy. □

We now need two simple lemmas about barbed bisimulation, before tackling the proof of Lemma 5.2.2. Barbed bisimulation is not preserved by parallel composition. For instance, $a.b \;\dot{\sim}_\Gamma\; a.\mathbf{0}$, while $a.b \mid \overline{a} \;\dot{\not\sim}_\Gamma\; a.\mathbf{0} \mid \overline{a}$. However, $\dot{\sim}_\Gamma$ is preserved by restriction.

A.7. Lemma: $P \;\dot{\sim}_{\Gamma, a:S}\; Q$ implies $(\boldsymbol{\nu}\, a : S)\, P \;\dot{\sim}_\Gamma\; (\boldsymbol{\nu}\, a : S)\, Q$.

Proof: Straightforward. □

A very useful technique for reducing the amount of work needed for proving a bisimilarity is to consider the closure of the bisimulation by some auxiliary relation; typically the auxiliary relation is taken to be the bisimilarity itself. We write $P \,\Re\, Q$ to mean that $(P, Q) \in \Re$, and $P \;\dot{\sim}_\Gamma\; \Re \;\dot{\sim}_\Gamma\; Q$ to mean that there are R_1 and R_2 such that $P \;\dot{\sim}_\Gamma\; R_1 \,\Re\, R_2 \;\dot{\sim}_\Gamma\; Q$.

A.8. Definition: [Barbed Γ-bisimulation up to $\dot{\sim}$] A relation $\Re$ is a barbed Γ-bisimulation up to $\dot{\sim}$ if $(P, Q) \in \Re$ implies

1. if $P \longrightarrow P'$, then there exists Q' such that $Q \longrightarrow Q'$ and $P \;\dot{\sim}_\Gamma\; \Re \;\dot{\sim}_\Gamma\; Q'$;
2. if $Q \longrightarrow Q'$, then there exists P' such that $P \longrightarrow P'$ and $P' \;\dot{\sim}_\Gamma\; \Re \;\dot{\sim}_\Gamma\; Q'$;
3. for each name a, $P \downarrow_a$ iff $Q \downarrow_a$.

A.9. Lemma: Suppose $\Re$ is a barbed Γ-bisimulation up to $\dot{\sim}$; then $\Re \subseteq \;\dot{\sim}_\Gamma$.

Proof: Standard technique for bisimulation up to a strong bisimulation [Mil89]. □

We are now ready to give the proof of Lemma 5.2.2. We first recall its assertion:

Lemma 5.2.2: *Let P_1 and P_2 be two processes obeying the sorting Δ. Suppose that for every sorting Γ, substitution σ, and process Q such that*

1. $\Gamma <_{\mathrm{E}} \Delta$,
2. *if $\sigma = \{\tilde{c}/\tilde{b}\}$, then $\Gamma(\tilde{c}) \leq \Gamma(\tilde{b})$, and*
3. *Q obeys Γ,*

it holds that

$$Q \mid P_1\sigma \;\dot{\sim}_\Gamma\; Q \mid P_2\sigma.$$

Then $P_1 \sim_\Delta P_2$.

Proof: By Definition 5.2.1, $P_1 \sim_\Delta P_2$ holds if

$$C[P_1] \;\dot{\sim}_\Gamma\; C[P_2] \tag{15}$$

for each sorting Γ and (Γ,Δ)-context $C[\cdot]$.

To obtain a sufficiently strong induction hypothesis, we prove something slightly different, adding a substitution σ and a parallel composition with a process Q. That is, we prove that for every sorting Γ, every (Γ,Δ)-context $Q \mid C[\cdot]$, and every substitution $\sigma = \{\tilde{c}/\tilde{b}\}$ with $\Gamma(\tilde{c}) \leq \Gamma(\tilde{b})$, it holds that

$$Q \mid C[P_1]\sigma \;\dot{\sim}_\Gamma\; Q \mid C[P_2]\sigma \tag{16}$$

Taking $Q = \mathbf{0}$, (15) is an instance of (16).

The *depth* of a context is the number of operators encountered to reach the hole; thus the depth of $[\cdot]$ is one and the depths of $P \mid C[\cdot]$, $\alpha . C[\cdot]$ and $!C[\cdot]$ equal the depth of $C[\cdot]$ plus one. We prove (16) by induction on the depth of the context $C[\cdot]$. In the proof, we use the process Q in (16) to be able to cope with the case in which the outermost operator of the context is a replication or a parallel composition, and the substitution σ in (16) for the case in which the outermost operator is an input prefix.

In the base case, $C[\cdot] = [\cdot]$, and the fact that $Q \mid C[\cdot]$ is a (Γ,Δ)-context implies conditions (1) and (3) from the statement of the lemma; condition (2) is satisfied by assumption, and (16) follows.

For the induction part, we proceed by a case analysis on the outermost operator of $C[\cdot]$. In the barbed bisimulations we define, it is assumed that $P_1, P_2, Q, \Gamma, \Delta$, and σ satisfy the requirements in assertion (16). In each case, we shall only examine clause (1) of the definition of barbed bisimulation (5.2.1), since clause (3) is always trivial and (2) is similar to (1). Moreover, we shall explicitly write sort annotations only in the prefix case, since sorting does not play a central role in the other cases; we shall also feel free to write $\dot{\sim}$ instead of $\dot{\sim}_\Gamma$.

Prefix: $C[\cdot]$ is of the form $\alpha . D[\cdot]$. Let

$$\Re = \Big\{ \Big(Q \mid (\alpha . D[P_1])\sigma,\ Q \mid (\alpha . D[P_2])\sigma \Big) \Big\} \cup \dot{\sim}_\Gamma \ .$$

The relation $\Re$ is a barbed Γ-bisimulation. Consider the possible reductions of $Q \mid (\alpha . D[P_1])\sigma$; there are two cases to look at:

- The interaction occurs inside Q:

  $$Q \mid (\alpha . D[P_1])\sigma \longrightarrow Q' \mid (\alpha . D[P_1])\sigma$$

  This is matched by

  $$Q \mid (\alpha . D[P_2])\sigma \longrightarrow Q' \mid (\alpha . D[P_2])\sigma,$$

  since $(Q' \mid (\alpha . D[P_1])\sigma, Q' \mid (\alpha . D[P_2])\sigma) \in \Re$.

- The interaction is between Q and $(\alpha . D[P_1])\sigma$. There are two cases: α is an output or an input; the former is simpler, so we only look at the latter. For simplicity we assume that the object part of α has length one, say $\alpha = a(b : S)$ and that σ does not modify α, i.e. $(\alpha.D[\cdot])\sigma = \alpha.(D[\cdot]\sigma)$. We denote by $\overline{a}\langle c\rangle . Q'$ the subterm of Q which interacts with $(\alpha . D[P_1])\sigma$. By Lemma A.6, $Q \equiv (\boldsymbol{\nu}\, \tilde{y} : \tilde{T})\,(Q'' \mid \overline{a}\langle c\rangle . Q')$. Then, assuming $\tilde{y}$ fresh, we have:

  $$\begin{array}{rcl} Q \mid a(b:S) . (D[P_1]\sigma) & \equiv & \\ (\boldsymbol{\nu}\, \tilde{y} : \tilde{T}) \big(Q'' \mid \overline{a}\langle c\rangle . Q' \mid a(b:S) . (D[P_1]\sigma) \big) & \longrightarrow & (\boldsymbol{\nu}\, \tilde{y} : \tilde{T}) \big(Q'' \mid Q' \mid D[P_1]\sigma\{c/b\} \big) \end{array}$$

  Similarly, from the process $Q \mid a(b:S) . (D[P_2]\sigma)$ we can infer:

  $$\begin{array}{rcl} Q \mid a(b:S) . (D[P_2]\sigma) & \equiv & \\ (\boldsymbol{\nu}\, \tilde{y} : \tilde{T}) \big(Q'' \mid \overline{a}\langle c\rangle . Q' \mid a(b:S) . (D[P_2]\sigma) \big) & \longrightarrow & (\boldsymbol{\nu}\, \tilde{y}) \big(Q'' \mid Q' \mid D[P_2]\sigma\{c/b\} \big) \end{array}$$

  Now, by Lemma 2.6.4, $(\Gamma, \tilde{y} : \tilde{T})(c) \leq S$ and therefore, using the induction assumption, we have

  $$(Q'' \mid Q') \mid D[P_1]\sigma\{c/b\} \ \dot{\sim}_{\Gamma, \tilde{y}:\tilde{T}} \ (Q'' \mid Q') \mid D[P_2]\sigma\{c/b\}.$$

  From this, apply Lemma A.7 to get

  $$(\boldsymbol{\nu}\, \tilde{y} : \tilde{T}) \big(Q'' \mid Q' \mid D[P_1]\sigma\{c/b\} \big) \ \dot{\sim}_\Gamma \ (\boldsymbol{\nu}\, \tilde{y} : \tilde{T}) \big(Q'' \mid Q' \mid D[P_2]\sigma\{c/b\} \big),$$

  which closes up the bisimulation, since $\dot{\sim}_\Gamma \ \subseteq \Re$.

Restriction: $C[\cdot]$ is of the form $(\boldsymbol{\nu}\, a)\, D[\cdot]$. We want to show that

$$Q \mid \big((\boldsymbol{\nu}\, a)\, D[P_1] \big)\sigma \ \dot{\sim} \ Q \mid \big((\boldsymbol{\nu}\, a)\, D[P_2] \big)\sigma.$$

We can assume that a does not occur in Q and that σ does not modify a; therefore we have

$$Q \mid \big((\boldsymbol{\nu}\, a)\, D[P_1] \big)\sigma \equiv (\boldsymbol{\nu}\, a)\, (Q \mid D[P_1]\sigma) \ \dot{\sim} \ (\boldsymbol{\nu}\, a)\, (Q \mid D[P_2]\sigma) \equiv Q \mid \big((\boldsymbol{\nu}\, a)\, D[P_2] \big)\sigma,$$

where the use of $\dot{\sim}$ is derived from the induction assumption and Lemma A.7.

Parallel composition: $C[\cdot]$ is of the form $Q' \mid D[\cdot]$. We have to show that

$$Q \mid (Q' \mid D[P_1])\sigma \stackrel{\centerdot}{\sim}_\Gamma Q \mid (Q' \mid D[P_2])\sigma. \tag{17}$$

We have

$$Q \mid (Q' \mid D[P_1])\sigma \equiv (Q \mid Q') \mid D[P_1]\sigma \qquad \text{and} \qquad Q \mid (Q' \mid D[P_2])\sigma \equiv (Q \mid Q') \mid D[P_2]\sigma,$$

from which (17) follows from the induction assumption.

Replication: $C[\cdot]$ is of the form $!D[\cdot]$. We show that

$$\Re = \{(Q \mid !(D[P_1]\sigma), Q \mid !(D[P_2]\sigma))\}$$

is a barbed bisimulation up to $\stackrel{\centerdot}{\sim}$. We abbreviate $D[P_1]\sigma$ as D_A and $D[P_2]\sigma$ as D_B. We show that if $Q \mid !D_A \longrightarrow A$, then for some B, we have $Q \mid !D_B \longrightarrow B$ and

$$A \; \Re \stackrel{\centerdot}{\sim} B. \tag{18}$$

The processes which might cause an interaction in $Q \mid !D_A$ are Q and D_A. Therefore, if

$$H_A = Q \mid D_A,$$

then by Lemma A.5, modulo $\equiv$, any interaction in $Q \mid !D_A$ is of the form[1]

$$Q \mid !D_A \longrightarrow H'_A \mid !D_A, \tag{19}$$

where H'_A is a derivative of H_A, i.e.

$$H_A \longrightarrow H'_A. \tag{20}$$

Let us now turn to $Q \mid !D_B$. By the induction assumption of the lemma, for any process K,

$$K \mid D_A \stackrel{\centerdot}{\sim} K \mid D_B.$$

Using this fact, we get

$$Q \mid !D_B \equiv Q \mid D_B \mid !D_B \stackrel{\centerdot}{\sim} Q \mid D_A \mid !D_B = H_A \mid !D_B.$$

Now, by (20),

$$H_A \mid !D_B \longrightarrow H'_A \mid !D_B,$$

and by definition of $\Re$ it holds that $H'_A \mid !D_A \; \Re \; H'_A \mid !D_B$.
Summarizing, we have: $Q \mid !D_B \stackrel{\centerdot}{\sim} H_A \mid !D_B \longrightarrow H'_A \mid !D_B$ and $H'_A \mid !D_A \; \Re \; H'_A \mid !D_B$. From the former, by definition of $\stackrel{\centerdot}{\sim}$, for some B we also have

$$Q \mid !D_B \longrightarrow B \stackrel{\centerdot}{\sim} H'_A \mid !D_B.$$

This proves that $H'_A \mid !D_A \; \Re \stackrel{\centerdot}{\sim} B$, which for $A = H'_A \mid !D_A$ is precisely (18). □

B Proofs of the replicator theorems

B.1 Some results on names used only as triggers

Recall from Definition 5.3.1 that the expression "m is used only as a trigger in P" is understood with respect to a judgement of the form $\Gamma \vdash C[P] : \circ$. Sometimes we shall omit this judgement if it is clear from the context. We write $T^{\uparrow \mathsf{w}}$ for the sort obtained from T by replacing its outermost tag with w. Also, $(T)^{\leq \mathsf{w}}$ denotes a sort whose outermost tag is less than or equal to w.

[1] Had we included summation in the syntax of our π-calculus language, then two copies of D_A in the definition of H_A would be needed in order to capture all possible interactions in $!D_A$. For instance, if $D_A = \overline{a}\langle y\rangle . P + a(x) . Q$, then $!D_A \longrightarrow (P \mid Q\{y/x\}) \mid !D_A$, but D_A alone has no interaction.

B.1.1. Lemma: Suppose $\mathcal{I}(T) \leq \mathsf{w}$ and $S \leq T$. Then $S^{\uparrow\mathsf{w}} \leq T^{\uparrow\mathsf{w}}$.

Proof: By Corollary 2.4.9, we may assume that we are given an algorithmic derivation of $S \leq T$. The result follows by inspection of this derivation. $\square$

B.1.2. Lemma: Suppose $\Gamma, m : (S)^{\mathsf{r}} \vdash P : \circ$ and $\Gamma, m : (S)^{\mathsf{w}} \vdash P : \circ$. Then m is not free in P.

Proof: By induction on the structure of P. The interesting case is when $P = \overline{a}\langle c\rangle\,.\,Q$ (for notational convenience, we suppose that the tuple of arguments has length one). By the induction assumption, m is not free in Q. Therefore, we only have to show that $m \notin \{a, c\}$. Clearly, $a \neq m$, for otherwise $\Gamma, m : (S)^{\mathsf{r}} \nvdash P : \circ$. Now, suppose $a = c$. Then, from the hypothesis of the lemma and the definition of the typing system, it must be that

$$\Gamma(a) \leq ((S)^{\mathsf{r}})^{\mathsf{w}} \text{ and } \Gamma(a) \leq ((S)^{\mathsf{w}})^{\mathsf{w}}.$$

From this and the subtyping algorithm, we deduce that for some T, $\Gamma(a) \lesseqgtr (T)^{\leq\mathsf{w}}$ and

$$(S)^{\mathsf{r}} \leq T \tag{21}$$

$$(S)^{\mathsf{w}} \leq T. \tag{22}$$

But this is impossible because, by Lemma 2.4.12, from (21) we get that the outermost tag of T is r, and from (22) that the outermost tag of T is w. $\square$

B.1.3. Lemma: Suppose that $\Gamma, a : T \vdash P : \circ$, that m is used only as a trigger in P, and that $S \leq T$. Then $\Gamma, a : S \vdash P : \circ$, and m is used only as a trigger in P with respect to the narrower sorting.

Proof: For $a \neq m$, the result follows from the hypothesis $S \leq T$ by narrowing (2.6.2). Now, suppose $a = m$. We know that

$$\Gamma, a : T^{\uparrow\mathsf{w}} \vdash P : \circ \tag{23}$$

and we want to show that also

$$\Gamma, a : S^{\uparrow\mathsf{w}} \vdash P : \circ \tag{24}$$

holds. If $\mathcal{I}(T) \in \{\mathsf{w}, \mathsf{b}\}$, then (24) again follows by narrowing. If $\mathcal{I}(T) = \mathsf{r}$, then from $\Gamma, a : T \vdash P : \circ$, (23), and Lemma B.1.2, we infer that m is not free in P; therefore (24) is trivially true. $\square$

B.1.4. Corollary: Suppose that $\Delta \vdash P : \circ$, that m is used only as a trigger in P, and that $\Gamma <_{\mathrm{E}} \Delta$. Then $\Gamma \vdash P : \circ$ and m is used only as a trigger in P with respect to Γ.

Proof: By the previous lemma and weakening. $\square$

B.1.5. Lemma: Suppose that $\Gamma, a : S, b : S \vdash P : \circ$, and that a and b are used only as triggers in P. Then $\Gamma, a : S \vdash P\{a/b\} : \circ$ and a is used only as a trigger in $P\{a/b\}$.

Proof: By induction on the structure of P and the definitions of typing and subtyping. $\square$

B.1.6. Corollary: Suppose that $\Gamma, a : S, b : T \vdash P : \circ$ with $S \leq T$, and that a, b are used only as triggers in P. Then $\Gamma, a : S \vdash P\{a/b\} : \circ$ and a is used only as a trigger in $P\{a/b\}$.

Proof: By B.1.3 and B.1.5. $\square$

B.1.7. Lemma: Suppose $\Gamma \vdash a(b_1 : S_1, \ldots, b_n : S_n)\,.\,P_1 \,|\, \overline{a}\langle c_1, \ldots, c_n\rangle\,.\,P_2 : \circ$ and c_i is used only as a trigger in $\overline{a}\langle c_1, \ldots, c_n\rangle\,.\,P_2$. Then $\mathcal{I}(S_i) = \mathsf{w}$.

Proof: If $a \neq c_i$, then by the definitions of trigger and of the typing relation we have:

$$\Gamma(a) \leq (\Gamma(c_1), \ldots, \Gamma(c_i)^{\uparrow\mathsf{w}}, \ldots, \Gamma(c_n))^{\mathsf{w}} \quad \text{and} \quad \Gamma(a) \leq (S_1, \ldots, S_n)^{\mathsf{r}}.$$

By Lemma 2.4.10, we derive $\Gamma(c_i)^{\uparrow\mathsf{w}} \leq S_i$. Now, $\mathcal{I}(S_i) = \mathsf{w}$ follows from Lemma 2.4.12. The only difference in the case $a = c_i$ is that the proof of $\Gamma(a) \leq (\Gamma(c_1), \ldots, \Gamma(c_i)^{\uparrow\mathsf{w}}, \ldots, \Gamma(c_n))^{\mathsf{w}}$ requires two steps, namely $\Gamma(a) \leq \Gamma(a)^{\uparrow\mathsf{w}}$ and $\Gamma(a)^{\uparrow\mathsf{w}} \leq (\Gamma(c_1), \ldots, \Gamma(c_i)^{\uparrow\mathsf{w}}, \ldots, \Gamma(c_n))^{\mathsf{w}}$. These are derived from the definitions of subtyping and typing and the hypothesis of the lemma. $\square$

B.1.8. Lemma: Suppose $\Gamma \vdash P : \circ$ for $P = a(b_1 : S_1, \ldots, b_n : S_n)\,.\,P_1 \,|\, \overline{a}\langle c_1, \ldots, c_n\rangle\,.\,P_2$, and that, for some i, c_i is used only as a trigger in P_1 and $\overline{a}\langle c_1, \ldots, c_n\rangle\,.\,P_2$. Then c_i is used only as a trigger in $P_1\{c_1, \ldots, c_n/b_1, \ldots, b_n\} \,|\, P_2$.

Note, by the way, that Lemma B.1.8 is not a corollary of the preservation of sorting under reduction. It would be so if the hypothesis were that c_i is used only as a trigger in P: But this does not hold when a is equal to c_i, since a occurs in P in input position.

Proof: For notational convenience, take $n = i = 1$ and abbreviate c_1, b_1, and S_1 as c, b, and S, respectively. By hypothesis, we know that c is used only as a trigger in P_1 and P_2. Therefore, by Corollary B.1.6, c is used only as a trigger in $P_1\{^c/_b\} \,|\, P_2$ if $\Gamma(c) \leq S$ and b is used only as a trigger in P_1. The former holds by Lemma 2.6.4; the latter holds because, by Lemma B.1.7, we have $\mathcal{I}(S) = \mathsf{w}$. $\square$

B.1.9. Lemma: Suppose that $\Gamma \vdash C[P] : \circ$ and m is used only as a trigger in P. Moreover, let $\sigma = \{^b/_a\}$ with $\Gamma(b) \leq \Gamma(a)$ and $a, b \neq m$. Then $\Gamma \vdash C[P]\sigma : \circ$ and m is used only as a trigger in P with respect to this judgement.

Proof: Straightforward using Lemma 2.6.3. $\square$

B.2 The core of the proofs

In the two lemmas that follow, we suppose that m and m' are names of the same sort and write P^m and $P^{m'}$ for $P\{^m/_{m'}\}$ and $P\{^{m'}/_m\}$, respectively.

B.2.1. Lemma: Suppose that m, m' are used only as triggers in P^m and P.

1. If $P^m \longrightarrow Q_1$, then there exits Q such that $P \longrightarrow Q$ and $Q_1 = Q^m$.
2. If $P \longrightarrow Q$, then $P^m \longrightarrow Q^m$.

Moreover, in both cases, m and m' are used only as triggers in Q^m and Q.

Proof: Straightforward transition induction. Since m and m' are used only as triggers, they cannot appear in input position and consequently no interaction along m or m' is possible in P or P^m. Hence, the substitution of m' with m does not add new derivatives. Moreover, the names m and m' are used only as triggers in the derivatives of P and P^m, because sorting is preserved under reduction. $\square$

In the following, when talking about barbed Γ-congruence, we shall often omit the sorting Γ when it is not needed or is clear from the context. Moreover, when proving barbed bisimilarities, we shall only examine clause (1) of Definition 5.2.1, since clause (3) is always trivial and (2) is similar to (1).

B.2.2. Lemma: Suppose that m is used only as a trigger in P_1, P_2, and R. Then

$$Q \,|\, (P_1 \,|\, P_2)\{m(\tilde{x}) := R\} \stackrel{\bullet}{\sim}_\Gamma Q \,|\, P_1\{m(\tilde{x}) := R\} \,|\, P_2\{m(\tilde{x}) := R\}.$$

Proof: We shall show below that if m and m' are used only as triggers in P and R, and $m' \notin \mathrm{fn}(R)$, then

$$(\boldsymbol{\nu}\, m)\,(P^m \,|\, !m(\tilde{x})\,.\,R) \stackrel{\bullet}{\sim} (\boldsymbol{\nu}\, m, m')\,(P \,|\, !m(\tilde{x})\,.\,R \,|\, !m'(\tilde{x})\,.\,R^{m'}). \tag{25}$$

Using this, the assertion of the lemma is proved as follows, assuming that m, m' are not free in Q and that m' does not appear in $Q \,|\, (P_1 \,|\, P_2)\{m(\tilde{x}) := R\}$.

$$\begin{array}{rcl}
Q \,|\, (P_1 \,|\, P_2)\{m(\tilde{x}) := R\} & = & \\
Q \,|\, (\boldsymbol{\nu}\, m)\,(P_1 \,|\, P_2 \,|\, !m(\tilde{x})\,.\,R) & \equiv & \\
(\boldsymbol{\nu}\, m)\,(Q \,|\, P_1 \,|\, P_2 \,|\, !m(\tilde{x})\,.\,R) & \stackrel{\bullet}{\sim} & \text{(by (25))} \\
(\boldsymbol{\nu}\, m, m')\,(Q \,|\, P_1 \,|\, P_2^{m'} \,|\, !m(\tilde{x})\,.\,R \,|\, !m'(\tilde{x})\,.\,R^{m'}) & \equiv & (*) \\
Q \,|\, (\boldsymbol{\nu}\, m)\,(P_1 \,|\, !m(\tilde{x})\,.\,R) \,|\, (\boldsymbol{\nu}\, m')\,(P_2^{m'} \,|\, !m'(\tilde{x})\,.\,R^{m'}) & \equiv & \text{(by } \alpha\text{-conversion)} \\
Q \,|\, (\boldsymbol{\nu}\, m)\,(P_1 \,|\, !m(\tilde{x})\,.\,R) \,|\, (\boldsymbol{\nu}\, m)\,(P_2 \,|\, !m(\tilde{x})\,.\,R) & = & \\
Q \,|\, P_1\{m(\tilde{x}) := R\} \,|\, P_2\{m(\tilde{x}) := R\} & &
\end{array}$$

The transformation $(*)$ is obtained using the structural laws for parallel composition and restriction, in particular the fact that m is only free in P_1 and $m(x)\,.\,R$, whereas m' is only free in $P_2^{m'}$ and $m'(x)\,.\,R^{m'}$. The transformation (25) can be applied because the names m and m' are used only as triggers in P_1, $P_2^{m'}$ and R (by the hypothesis of the lemma and Lemma B.1.5) and m' is not free in R. Thus, we are left with only (25) to prove.

We show that

$$\Re = \Big\{\big((\boldsymbol{\nu}\, m)\,(P^m \,|\, !m(\tilde{x})\,.\,R),\ (\boldsymbol{\nu}\, m, m')\,(P \,|\, !m(\tilde{x})\,.\,R \,|\, !m'(\tilde{x})\,.\,R^{m'})\big) \\ \text{such that } m, m' \text{ are used only as triggers in } P \text{ and } R \\ \text{and } m' \notin \mathrm{fn}(R) \Big\}$$

is a barbed bisimulation. Let $(A, B) \in \Re$, and consider the possible moves by A; the case in which B moves first can be handled in a similar way. Suppose $A \longrightarrow A'$. By Lemma A.5 it is enough to consider only once each pair of possible interacting subterms in A. There are two cases to examine, according to whether the interacting subterms are both in P^m or one is in P^m and the other is $m(x)\,.\,R$. For the former case, use Lemma B.2.1. Let us consider now the latter.

In order to see better how B matches the move by A it is convenient to write a subterm of P^m as P_1^m, where P_1 is the corresponding subterm of P. Thus, let

$$H_P^m = (\overline{m''}\langle\tilde{c}\rangle\,.\,K_P)^m \qquad \text{for } m'' \in \{m, m'\}$$

be the subterm in P^m which interacts with $m(x)\,.\,R$. Using Lemma A.6, for some $\tilde{y}$ and L_P^m,

$$P^m \equiv (\boldsymbol{\nu}\,\tilde{y})\left(L_P^m \,|\, (\overline{m''}\langle\tilde{c}\rangle\,.\,K_P)^m\right)$$

and therefore,

$$A = (\boldsymbol{\nu}\, m)\,(P^m \,|\, !m(\tilde{x})\,.\,R) \longrightarrow \equiv (\boldsymbol{\nu}\, m)\left((\boldsymbol{\nu}\,\tilde{y})\,(L_P^m \,|\, K_P^m \,|\, (R\{\tilde{c}/\tilde{x}\})^m) \,|\, !m(\tilde{x})\,.\,R\right) \equiv A',$$

where the last use of $\equiv$ is derived from Lemma A.5. Now we can show that B can match the move $A \longrightarrow A'$. Suppose $m'' = m$; the case $m'' = m'$ is analogous. We have

$$P \equiv (\boldsymbol{\nu}\,\tilde{y})\left(L_P \,|\, \overline{m}\langle\tilde{c}\rangle\,.\,K_P\right)$$

from which we get

$$\begin{array}{rcl} B & = & (\boldsymbol{\nu}\, m, m')\,(P \,|\, !m(\tilde{x})\,.\,R \,|\, !m'(\tilde{x})\,.\,R^{m'}) \\ & \longrightarrow & (\boldsymbol{\nu}\, m, m')\left((\boldsymbol{\nu}\,\tilde{y})\,(L_P \,|\, K_P \,|\, R\{\tilde{c}/\tilde{x}\}) \,|\, !m(\tilde{x})\,.\,R \,|\, !m'(\tilde{x})\,.\,R^{m'}\right) \\ & = & B'. \end{array}$$

To conclude that $A' \,\Re\, B'$, we also have to guarantee that the side condition in the definition of $\Re$ is satisfied, namely m and m' are used only as triggers in $(\boldsymbol{\nu}\,\tilde{y})\,(L_P \,|\, K_P \,|\, R\{\tilde{c}/\tilde{x}\})$ and R. This follows from the hypothesis $A \,\Re\, B$, which guarantees that m and m' are used only as triggers in R and L_P, and from Lemma B.1.8, which guarantees that they are used only as triggers in $K_P \,|\, R\{\tilde{c}/\tilde{x}\}$. □

B.2.3. Lemma: If m is used only as a trigger in P and R, then

$$Q \,|\, !(P\{m(\tilde{x}) := R\}) \;\dot{\sim}_\Gamma\; Q \,|\, (!P)\{m(\tilde{x}) := R\}.$$

Proof: We show that

$$\Re = \{(Q \,|\, !(P\{m(\tilde{x}) := R\}), Q \,|\, (!P)\{m(\tilde{x}) := R\}) \mid m \text{ is used only as a trigger in } P, R\}$$

is a barbed bisimulation up to $\dot{\sim}$. Let $(A, B) \in \Re$ and consider a possible interaction in A. The terms that might participate in the interaction are Q and $P\{m(\tilde{x}) := R\}$. Therefore, if

$$H_A = Q \,|\, P\{m(\tilde{x}) := R\},$$

then by Lemma A.5, modulo $\equiv$, any interaction in A can be written in the form

$$A \longrightarrow H'_A \,|\, !(P\{m(\tilde{x}) := R\}) = A', \tag{26}$$

where H'_A is a derivative of H_A, i.e.

$$H_A \longrightarrow H'_A. \tag{27}$$

Let us see how B can match the move (26), up to $\dot{\sim}$. We have

$$B \equiv Q \mid (P \mid !P)\{m(\tilde{x}) := R\},$$

which, using Lemma B.2.2 is barbed bisimilar with

$$Q \mid P\{m(\tilde{x}) := R\} \mid (!P)\{m(\tilde{x}) := R\} = B_1.$$

Now, by (27), we have

$$B_1 \longrightarrow H'_A \mid (!P)\{m(\tilde{x}) := R\} = B'_1,$$

and from (26) we get $(A', B'_1) \in \Re$. Summarising, we have $B \mathrel{\dot{\sim}} B_1 \longrightarrow B'_1$ and $(A', B'_1) \in \Re$. From the latter, by definition of $\dot{\sim}$, for some B' we also have

$$B \longrightarrow B' \mathrel{\dot{\sim}} B'_1.$$

This proves that $A' \Re \dot{\sim} B'$, which is enough because $\Re$ is a bisimulation up to $\dot{\sim}$. □

Now we are ready to prove the two replicator theorems.

Theorem 5.3.2: *If* $\Gamma \vdash (P \mid Q)\{m(\tilde{x}) := R\} : \circ$ *and* m *is used as a trigger in* P, Q, *and* R, *then*

$$(P \mid Q)\{m(\tilde{x}) := R\} \sim_\Gamma P\{m(\tilde{x}) := R\} \mid Q\{m(\tilde{x}) := R\}.$$

Proof: By Lemma 5.2.2 it is enough to show that for every Γ, σ, and Q such that

1. $\Gamma <_{\mathrm{E}} \Delta$,
2. if $\sigma = \{\tilde{c}/\tilde{b}\}$, then $\Gamma(\tilde{c}) \leq \Gamma(\tilde{b})$,
3. $\Gamma \vdash Q : \circ$,

it holds that

$$Q \mid \big((P_1 \mid P_2)\{m(\tilde{x}) := R\}\big)\sigma \mathrel{\dot{\sim}_\Gamma} Q \mid \big(P_1\{m(\tilde{x}) := R\} \mid P_2\{m(\tilde{x}) := R\}\big)\sigma. \tag{28}$$

We may assume that σ does not modify m or $\tilde{x}$, so we have

$$Q \mid \big((P_1 \mid P_2)\{m(\tilde{x}) := R\}\big)\sigma = Q \mid (P_1\sigma \mid P_2\sigma)\{m(\tilde{x}) := R\sigma\}$$

and

$$Q \mid \big(P_1\{m(\tilde{x}) := R\} \mid P_2\{m(\tilde{x}) := R\}\big)\sigma = Q \mid (P_1\sigma)\{m(\tilde{x}) := R\sigma\} \mid (P_2\sigma)\{m(\tilde{x}) := R\sigma\}.$$

Using Corollary B.1.4 and Lemma B.1.9, we derive that m is used only as a trigger in $P_1\sigma$, $P_2\sigma$, and $R\sigma$. Then (28) follows from Lemma B.2.2. □

Theorem 5.3.3: *If* $\Gamma \vdash (!P)\{m(\tilde{x}) := R\} : \circ$ *and* m *is used as a trigger in* P *and* R, *then*

$$(!P)\{m(\tilde{x}) := R\} \sim_\Gamma !(P\{m(\tilde{x}) := R\}).$$

Proof: Similar to the proof of the first replicator theorem, using Lemma B.2.3 in place of Lemma B.2.2. □

C Validity of β-reduction for the encoding $\mathcal{V}'$

The following are some obvious laws:

C.1. Lemma:

1. $(\boldsymbol{\nu}\, a)\left(\overline{a}\langle\tilde{c}\rangle \,.\, P \mid a(\tilde{b}) \,.\, Q\right) \approx (\boldsymbol{\nu}\, a)\,(P \mid Q\{\tilde{c}/\tilde{b}\})$.
2. Suppose a not free in P. Then $P\{a(\tilde{x}) := R\} \sim P$.
3. Suppose a does not occur in the prefix α. Then it holds that $(\alpha\,.\,P)\{a(\tilde{x}) := R\} \sim \alpha\,.\left(P\{a(\tilde{x}) := R\}\right)$.

The schema of the proof of validity of β reduction for $\mathcal{V}'$ is the same used by Milner to show the analogous result for the lazy λ-calculus encoding [Mil91]. The sorts of the names for $\mathcal{V}'$ have been described in Section 4; for simplicity, explicit sort annotations and Γ subscripts (e.g. on bisimulations) are omitted.

C.2. Theorem: $\mathcal{V}'[\![(\lambda x.M)(\lambda y.N)]\!]\langle p\rangle \approx \mathcal{V}'[\![M\{{}^{\lambda y.N}/x\}]\!]\langle p\rangle$.

Proof: We can reduce the process

$$\mathcal{V}'[\![(\lambda x.M)(\lambda y.N)]\!]\langle p\rangle = (\boldsymbol{\nu}\, q, r)\left(\mathcal{V}[\![\lambda x.M]\!]\langle q\rangle \mid \mathcal{V}[\![\lambda y.N]\!]\langle r\rangle \mid App\langle p,q,r\rangle\right)$$

to

$$\mathcal{V}'[\![M]\!]\langle p\rangle\{x(w) := w(x,q)\,.\,\mathcal{V}'[\![N]\!]\langle q\rangle\}$$

by repeated expansions (Lemma C.1(1)) which consume the actions of the arbiter App, plus Lemma C.1(2) to garbage collect the environment entries which cannot be accessed any more. In the following, let R abbreviate $w(x,q)\,.\,\mathcal{V}'[\![N]\!]\langle q\rangle$. We show, by induction on M, that for any p,

$$A = \mathcal{V}'[\![M]\!]\langle p\rangle\{x(w) := R\} \sim \mathcal{V}'[\![M\{{}^{\lambda y.N}/x\}]\!]\langle p\rangle = B.$$

We have to distinguish the three cases in which M is a variable, an abstraction or an application. But first, observe that the name x is used only as a trigger in $\mathcal{V}'[\![M]\!]\langle p\rangle$ and R. This is trivially true for R because x does not occur free in R. As regards $\mathcal{V}'[\![M]\!]\langle p\rangle$, name x may only occurs free in $\mathcal{V}'[\![M]\!]\langle p\rangle$ in subterms of the form $\overline{q}\langle x\rangle$. This agrees with the hypothesis that x is used only as a trigger because, following the terminology in Section 4, q is a location name; hence, the sort assigned to the name carried by q is $(S_p)^{\mathsf{w}}$, which has w as outermost tag. The fact that x is used only as a trigger in $\mathcal{V}'[\![M]\!]\langle p\rangle$ and R will authorize us to use both the replicator theorems 5.3.2 and 5.3.3 in the transformations below.

Variable: $M = z$.

Both the case $z = x$ and the case $z \neq x$ are easy.

Abstraction: $M = \lambda z.M'$. We have

$$\begin{aligned}
A = \quad & (\boldsymbol{\nu}\, y)\left(\overline{p}\langle y\rangle \mid !\, y(w)\,.\,w(z,q)\,.\,\mathcal{V}[\![M']\!]\langle q\rangle\right)\{x(w) := R\} & \equiv \quad & (29)\\
& (\boldsymbol{\nu}\, y)\left(\overline{p}\langle y\rangle \mid \left(!\, y(w)\,.\,w(z,q)\,.\,\mathcal{V}[\![M']\!]\langle q\rangle\right)\{x(w) := R\}\right) & \sim \quad & (30)\\
& (\boldsymbol{\nu}\, y)\left(\overline{p}\langle y\rangle \mid !\left((y(w)\,.\,w(z,q)\,.\,\mathcal{V}[\![M']\!]\langle q\rangle)\{x(w) := R\}\right)\right) & \sim \quad & (31)\\
& (\boldsymbol{\nu}\, y)\left(\overline{p}\langle y\rangle \mid !\, y(w)\,.\,w(z,q)\,.\left(\mathcal{V}[\![M']\!]\langle q\rangle\{x(w) := R\}\right)\right) & \sim \quad & (32)\\
& (\boldsymbol{\nu}\, y)\left(\overline{p}\langle y\rangle \mid !\, y(w)\,.\,w(z,q)\,.\,\mathcal{V}'[\![M'\{{}^{\lambda y.N}/x\}]\!]\langle q\rangle\right) & = \quad & B,
\end{aligned}$$

where, in (29), the scope of the local environment $\{x(w) := R\}$ has been reduced, since x does not occur in $\overline{p}\langle y\rangle$; the equality (30) pushes the local environment inside the replicator and is derived from the second replicator theorem; the equality (31) pushes the local environment inside the two prefixes, and is derived from Lemma C.1(3); finally, the equality (32) is obtained from the induction assumption.

Application: $M = M_1M_2$. We have

$$\begin{aligned}
A = (\boldsymbol{\nu}\, q, r)\left(\mathcal{V}'[\![M_1]\!]\langle q\rangle \mid \mathcal{V}'[\![M_2]\!]\langle r\rangle \mid App\langle p,q,r\rangle\right)\{x(w) := R\} & \quad \equiv \quad & (33)\\
(\boldsymbol{\nu}\, q, r)\left(\mathcal{V}'[\![M_1]\!]\langle q\rangle\{x(w) := R\} \mid \mathcal{V}'[\![M_2]\!]\langle r\rangle\{x(w) := R\} \mid App\langle p,q,r\rangle\,\{x(w) := R\}\right) & \quad \sim \quad & (34)\\
(\boldsymbol{\nu}\, q, r)\left(\mathcal{V}'[\![M_1]\!]\langle q\rangle\{x(w) := R\} \mid \mathcal{V}'[\![M_2]\!]\langle r\rangle\{x(w) := R\} \mid App\langle p,q,r\rangle\right) & \quad \sim \quad & (35)\\
(\boldsymbol{\nu}\, q, r)\left(\mathcal{V}'[\![M_1\{{}^{\lambda y.N}/x\}]\!]\langle q\rangle \mid \mathcal{V}'[\![M_2\{{}^{\lambda y.N}/x\}]\!]\langle r\rangle \mid App\langle p,q,r\rangle\right) & \quad = \quad & B.
\end{aligned}$$

The equality (33) is obtained by first enlarging the scope of the restrictions on q and r and then applying the first replicator theorem to distribute the local environment $\{x(w) := R\}$ over the three processes of the parallel composition. The equality (34) is obtained from Lemma C.1(2), since x is not free in $App\langle p, q, r\rangle$; finally equality (35) is obtained from the induction assumption. $\square$

References

[Abr89] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1989.

[AC91] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages*, pages 104–118, Orlando, FL, January 1991. Also available as DEC Systems Research Center Research Report number 62, August 1990. To appear in TOPLAS.

[Car86] Luca Cardelli. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, pages 21–47. Springer-Verlag, 1986. Lecture Notes in Computer Science No. 242.

[Cou83] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.

[dB72] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[EN86] U. Engberg and M. Nielsen. A calculus of communicating systems with label-passing. Report DAIMI PB-208, Computer Science Department, University of Aarhus, Denmark, 1986.

[Gay93] Simon J. Gay. A sort inference algorithm for the polyadic π-calculus. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages*, January 1993.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[GR93] P. Glavan and D. Rosenzweig. Communicating evolving algebras. In E. Börger, S. Martini, G. Jäger, H. Kleine Büning, and M.M. Richter, editors, *Computer Science Logic, Selected Papers from CSL'92*, volume 702 of *Lecture Notes in Computer Science*, pages 186–215. Springer Verlag, 1993.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mil90] R. Milner. Functions as processes. Research Report 1154, INRIA, Sophia Antipolis, 1990. Final version in *Journal of Mathem. Structures in Computer Science* 2(2):119–141, 1992.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, LFCS, Dept. of Comp. Sci., Edinburgh Univ., October 1991. Also in *Logic and Algebra of Specification*, ed. F.L. Bauer, W. Brauer and H. Schwichtenberg, Springer Verlag, 1993.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[MS92] R. Milner and D. Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *19th ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer Verlag, 1992.

[Plo75] G.D Plotkin. Call by name, call by value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[PS93] J. Parrow and D. Sangiorgi. Algebraic theories for name-passing calculi. Technical Report ECS–LFCS–93–262, LFCS, Dept. of Comp. Sci., Edinburgh Univ., 1993. To appear in *Information and Compuation.* Short version to appear in the *Proc. REX Summer School/Symposium 1993*, LNCS, Springer Verlag.

[Rey74] John Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, New York, 1974. Springer-Verlag LNCS 19.

[Rey85] John Reynolds. Three approaches to type structure. In *Mathematical Foundations of Software Development.* Springer-Verlag, 1985. Lecture Notes in Computer Science No. 185.

[Rey88] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.

[San92] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis CST–99–93, Department of Computer Science, University of Edinburgh, 1992.

[San93] D. Sangiorgi. A theory of bisimulation for the π-calculus. Technical Report ECS–LFCS–93–270, LFCS, Dept. of Comp. Sci., Edinburgh Univ., 1993. Extended Abstract in *Proc. CONCUR '93*, volume 715 of Lecture Notes in Computer Science, Springer Verlag.

[Tur94] David N. Turner, 1994. Ph.D. thesis, LFCS, University of Edinburgh. In preparation.

[VH93] Vasco T. Vasconcelos and Kohei Honda. Principal typing schemes in a polyadic pi-calculus. In *Proceedings of CONCUR '93*, July 1993. Also available as Keio University Report CS-92-004.

[Wal92] David Walker. Objects in the pi-calculus. To appear in Information and Computation, 1992.