Typing and Subtyping for Mobile Processes

Benjamin Pierce[*]
INRIA, Projet Formel
BP 105
79153 Le Chesnay Cedex, France
Benjamin.Pierce@inria.fr

Davide Sangiorgi
Department of Computer Science
King's Buildings
Edinburgh, EH9 3JZ, U.K.
sad@dcs.ed.ac.uk

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 some 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, for which β-reduction does not hold 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 [14] achieved a remarkable simplification and generalization of its predecessors (including CCS [11] and ECCS [7]) by focusing on the notion of *naming* and allowing the data values communicated along channels to themselves be channels. 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 [12] 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 the atomic unit of communication is finite tuples of names instead of single names [13]. 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 $P_t = b(t,f)\,.\,\overline{t}\langle\rangle\,.\,P_t$ 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, $P_f = b(t,f)\,.\,\overline{f}\langle\rangle\,.\,P_f$ 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 the 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; in parallel with P_f, Q_f is activated.

Milner observed that the channels used by this collection 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 only to communicate empty tu-

*Current address: Department of Computer Science, King's Buildings, Edinburgh EH9 3JZ, U.K.

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

ples. This situation can be described by the following *sorting*:

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

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 [17, 16] 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. (It cannot describe a channel that is used to carry an alternating sequence of two- and three-element tuples.) Although it might be possible to develop sorting disciplines that take into account 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 [25, 23]; it has been further studied by Turner [24] and Gay [8], who consider 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.

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 simply by reading print requests from p and throwing them away: $C_2 = p(j{:}T)\,.\,C_2$. We can prevent this kind of 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: $-$ for input, $+$ for output, or $\pm$ 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 $(T)^+$. The printer, on the other hand, should only read from p; from its point of view, p has sort $(T)^-$. Such an approach is reminiscient 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 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 $\pm$ or $+$. Similarly, an input operation $x(y{:}T)$ is well sorted if $S_x \leq T$, where $x{:}(S_x)^I$ and I is either $\pm$ or $-$. 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 combining subtyping and recursive types in typed λ-calculi [2]. 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 in the ordinary π-calculus).

2 Basics

This section defines the syntax, operational semantics, typing rules, and subsort relation of our calculus and develops some of its properties.

2.1 Notational Preliminaries

Since our purpose in this paper is to study basic theoretical properties of our type system rather than to propose a pragmatic notation for programming, we

[2] We learned recently that a by-structure presentation of Milner's π-calculus sorting has been studied independently by Turner [24]. His system is essentially identical to the fragment of ours in which subtyping is omitted.

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

Figure 1: Syntax

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. [24, 8]). 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 well-formed processes is *ok*; more interesting process types are considered in Section 6. $\tilde{S}$ stands for a sequence of sorts $S_1..S_n$; 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 [13], we introduce an explicit replication operator ! instead of allowing systems of recursive definitions of process expressions; $!P$ stands for the parallel composition of any finite number of copies of P. To simplify the presentation, we omit the notational devices of *abstraction* and *concretion* [13] 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 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, 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 $\Gamma \vdash P : ok$, 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 the 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 DeBruijn indices [6], 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, which 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.

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: $\mathcal{I}((S_1..S_n)^I) = I$ and $\mathcal{I}(\mu A.\ S) = \mathcal{I}(S\{^{\mu A.\ S}/_A\})$. Similarly, $\mathcal{I}_i$ extracts the I/O annotation of the ith component of a sort: $\mathcal{I}_i((S_1..S_n)^I) = \mathcal{I}(S_i)$ $and \mathcal{I}_i(\mu A.\ S) = \mathcal{I}_i(S\{^{\mu A.\ S}/_A\})$.

2.2 Reduction Semantics

Following Milner [13], 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:

1. abelian monoid laws for composition:
   $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$, $P \mid \mathbf{0} \equiv P$;
2. $(\boldsymbol{\nu}\, x)\, \mathbf{0} \equiv \mathbf{0}$, $(\boldsymbol{\nu}\, x)\, (\boldsymbol{\nu}\, y)\, P \equiv (\boldsymbol{\nu}\, y)\, (\boldsymbol{\nu}\, x)\, P$;
3. $((\boldsymbol{\nu}\, x)\, P) \mid Q \equiv (\boldsymbol{\nu}\, x)\, (P \mid Q)$, if x not free in Q;
4. $!P \equiv P \mid !P$;
5. $P \mid \mathit{wrong} \equiv \mathit{wrong}$, $!\mathit{wrong} \equiv \mathit{wrong}$, and $(\boldsymbol{\nu}\, a{:}S)\ \mathit{wrong} \equiv \mathit{wrong}$.

(Note that the side condition on rule 3 can be viewed as a consequence of our convention of regarding names as Debruijn indices.)

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

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

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

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \quad (\text{R-Par})$$

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

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

2.3 Subsorting

The subsort relation is generated by a basic *sub-tag* relation $I \leq J$, the least preorder containing $\pm \leq -$ and $\pm \leq +$. An *I/O-tree* is a finitely branching tree whose nodes are labelled with symbols from $\{-, +, \pm\}$. The metavariables $\mathcal{S}$, $\mathcal{T}$ and $\mathcal{U}$ range over trees. The tree whose root is labelled I and whose subtrees are $\mathcal{T}_1, \ldots, \mathcal{T}_n$ is written $[\mathcal{T}_1, \ldots, \mathcal{T}_n]^I$. 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 $\mathit{Tree}(S) = \mathit{Tree}(S'\{\mu A.S'/A\})$.

2.3.1. Definition: *Tree simulation* is the largest relation $\leq_{tr}$ on the class of I/O-trees such that $\mathcal{S} \leq_{tr} \mathcal{T}$ implies:

1. if $\mathcal{T} = [\mathcal{T}_1, \ldots \mathcal{T}_n]^{\pm}$ then $\mathcal{S} = [\mathcal{S}_1, \ldots \mathcal{S}_n]^{\pm}$ and, for all $1 \leq i \leq n$, both $\mathcal{S}_i \leq_{tr} \mathcal{T}_i$ and $\mathcal{T}_i \leq_{tr} \mathcal{S}_i$.
2. if $\mathcal{T} = [\mathcal{T}_1, \ldots \mathcal{T}_n]^{-}$ then $\mathcal{S} = [\mathcal{S}_1, \ldots \mathcal{S}_n]^I$ with $I \leq -$ and, for all $1 \leq i \leq n$, $\mathcal{S}_i \leq_{tr} \mathcal{T}_i$.
3. if $\mathcal{T} = [\mathcal{T}_1, \ldots \mathcal{T}_n]^{+}$ then $\mathcal{S} = [\mathcal{S}_1, \ldots \mathcal{S}_n]^I$ with $I \leq +$ and, for all $1 \leq i \leq n$, $\mathcal{T}_i \leq_{tr} \mathcal{S}_i$.

We write $\vdash S \lessgtr T$ when $\vdash S \leq T$ and $\vdash T \leq S$.

There is a close analogy between channel sorts and the *reference types* found in some programming languages. In Reynolds' language Forsythe [22], 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 modeled 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 $\pm$ tag, is constrained by both requirements and thus 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 [3, 4, 21]. 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 $((S)^-, (T)^+)^+$. Note that S occurs in a contravariant position (under both a $-$ tag and a $+$ tag) and T occurs in a covariant position (under two $+$ tags). Thus, the intuitive "compilation" $(S \rightarrow T) \mapsto ((S)^-, (T)^+)^+$ 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. But in practice, it is also useful to have a finitary characterization of subsorting in terms of a deterministic algorithm.

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

$$\frac{\text{for each } i,\ \Sigma \vdash S_i \gtrless_{T_i}}{\Sigma \vdash (S_1..S_n)^{\pm} \leq (T_1..T_n)^{\pm}} \qquad \text{(A-BB)}$$

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

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

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

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

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

(Here Σ ranges over finite sets of subsorting assumptions of the form $S \leq T$.) These rules can be read as an algorithm by imposing an ordering on the rules. It attempts 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.

2.3.3. Theorem: [Soundness and completeness of the algorithm] For all sorts S and T, $\vdash S \leq T$ iff $\mathit{Tree}(S) \leq_{tr} \mathit{Tree}(T)$.

Proof sketch: The proof uses techniques similar to those developed by Amadio and Cardelli for λ-calculi with subtyping and recursive types [2, 5]. We first show that the algorithm is terminating on all inputs by arguing that the size of Σ cannot increase without bound. Semi-completeness — the fact that the algorithm cannot fail when presented with inputs whose trees are related by the tree-subsort relation — follows from the fact that its behavior at each step mimics the definition of tree-subtyping, so that, when called on two types whose trees are in the tree-subsort relation, it must either succeed or be able to continue. The two together give completeness; soundness is straightforward.

This equivalence yields an easy proof of an additional technical lemma. which is used later in the proof of soundness of typing.

2.3.4. Lemma: If $\vdash S \leq (S_1..S_m)^-$ and $\vdash S \leq (T_1..T_n)^+$, then $m = n$ and, for each i, $\vdash T_i \leq S_i$.

2.4 Typing

The typing judgement $\Gamma \vdash P : ok$ asserts that "P is a well-behaved process under assumptions Γ" — i.e., if P is placed in an execution context where its free names obey this protocol described by Γ, then its 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 dead process $\mathbf{0}$ is well behaved in any context.

$$\Gamma \vdash \mathbf{0} : ok \qquad \text{(T-Nil)}$$

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

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

$$\frac{\Gamma \vdash P : ok}{\Gamma \vdash !P : ok} \qquad \text{(T-Repl)}$$

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.

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

The interesting cases are the rules for input and output. In order to be sure that the input expression $a(b_1{:}S_1\ ..\ b_n{:}S_n)\,.\,P$ 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 $S_1\ ..\ S_n$. These two conditions can be combined by requiring that $\Gamma(a) \le (S_1..S_n)^-$. (Note that the use of the subsort relation validates the cases where $\Gamma(a) = (S_1..S_n)^{\pm}$ or when $\Gamma(a) = (S'_1..S'_n)^-$ for some $S'_1 \le S_1\ ..\ S'_n \le S_n$.) Furthemore, we must check that the body P is well behaved assuming that the channels $b_1\ ..\ b_n$ behave consistently with the sorts $S_1\ ..\ S_n$.

$$\frac{\vdash\ \Gamma(a)\ \le\ (S_1..S_n)^- \qquad \Gamma, b_1{:}S_1\,..\,b_n{:}S_n\ \vdash\ P\ :\ ok}{\Gamma\ \vdash\ a(b_1{:}S_1\,..\,b_n{:}S_n)\,.\,P\ :\ ok} \qquad \text{(T-In)}$$

The case for output is parallel. To verify that the output expression is well behaved, we must check that the sort of a in Γ permits a to be used for outputting n-tuples of values. 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.

$$\frac{\vdash\ \Gamma(a)\ \le\ (\Gamma(b_1)..\Gamma(b_n))^+ \qquad \Gamma\ \vdash\ P\ :\ ok}{\Gamma\ \vdash\ \overline{a}\langle b_1..b_n\rangle\,.\,P\ :\ ok} \qquad \text{(T-Out)}$$

Note that the typing relation is syntax-directed: the rules can be read backwards to form a deterministic algorithm for checking the well-formedness of process expressions under a given typing.

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

2.4.2. Lemma: [Narrowing] If $\Gamma, a{:}S\ \vdash\ P\ :\ ok$ and $\vdash T \le S$ then $\Gamma, a : T\ \vdash\ P\ :\ ok$.

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

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.

This semantics refines the standard reduction semantics given above, in the sense that every process development that does not result in *wrong* in the tagged semantics can be mirrored in the original semantics. Thus, the soundness theorem for the tagged semantics implies the soundness of the original semantics as an easy corollary.

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

$$\begin{array}{lcl} E & ::= & \mathbf{0} \\ & | & E \mid F \\ & | & (\boldsymbol{\nu}\, a{:}S)\, E \\ & | & a^I(a_1{:}S_1\ ..\ a_n{:}S_n)\,.\,E \\ & | & \overline{a^I}\langle a_1^{I_1}\ ..\ a_n^{I_n}\rangle\,.\,E \\ & | & !E \\ & | & \mathit{wrong} \end{array}$$

We use E, F, and G to range over process expressions with tagged names. *Erase* is the function mapping tagged process expressions to ordinary process expressions by erasing tags.

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

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

There is a simple "compilation function" that translates any well-typed process into a statically consistent tagged process. Furthermore, it is easy to check that static consistency is invariant under certain substitutions:

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

The tagged equivalence and reduction relations are nearly identical to the original ones, with the exception of the rule for communication

$$\frac{\begin{array}{c} m = n \qquad I \leq - \qquad J \leq + \\ \text{for each } i, \;\; J_i \leq \mathcal{I}(S_i) \end{array}}{\begin{array}{l} a^I(b_1{:}S_1 \mathinner{..} b_m{:}S_m) \,.\, E \mid \overline{a^J}\langle c_1^{J_1} \mathinner{..} c_n^{J_n}\rangle \,.\, F \\ \longrightarrow E\{^{c_1..c_n}/_{b_1..b_n}\} \mid F \end{array}} \qquad \text{(TR-Comm)}$$

and the corresponding rule yielding *wrong* when an attempted communication fails to satisfy one of these premises. Using the properties of typing and subsorting established in this section and the previous one, we obtain a straightforward proof of the soundness of our typing rules with respect to the tagged semantics.

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

Moreover, the tagged semantics refines the ordinary semantics, in the sense that reductions in the tagged semantics can always be mirrored

3.5. Theorem: Suppose that $\Gamma \vdash \mathit{Erase}(E) : \mathit{ok}$ 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 $\mathit{Erase}(E') = P$.

4 Lambda calculus encodings

The use of I/O tags arises naturally in applications. As an example, we consider Milner's encoding of the lazy λ-calculus [1], an untyped λ-calculus with a deterministic reduction strategy that evaluates only redexes at the extreme left of the term. 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.

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* to 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, etc.). 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.

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

Using the sorts $S_a = (S_t, S_a)^-$ and $S_t = (S_a)^+$, we can annotate the translation so that the intended use of triggers and argument 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 $P\{m(x{:}S_t) := R\} \stackrel{def}{=} (\boldsymbol{\nu}\, m{:}(S_t)^\pm)\,(P \mid !(m(x{:}S_t) \,.\, R))$, the translation becomes:

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

5 Behavioral equivalence

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

The most popular way of defining behavioral equivalence on processes is via the notion of bisimulation. In [15, 23], Milner and Sangiorgi proposed *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 a natural congruence for the newborn calculus.

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 $a(x_1..x_n)$ or $\overline{a}\langle x_1..x_n\rangle$ 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 is proved in [23] that barbed congruence coincides in both CCS and π-calculus with the ordinary bisimilarity congruences.

In a sorted calculus, the processes being compared must obey the same sorting and the contexts employed must be compatible with this sorting. Since a process context may involve binding constructs, we introduce the term (Γ,Δ)-*context* to express this compatibility: C is said to be a (Γ,Δ)-context if C is well typed under Γ and Δ is pointwise greater (in the subsort relation) than Γ_C, the extension of Γ with the binders around the hole in $C[]$.

5.1. Definition: *Barbed* Δ-*bisimulation* is the largest symmetrical relation $\stackrel{\cdot}{\sim}_\Delta$ on the set of processes well typed under Δ such that $P \stackrel{\cdot}{\sim}_\Delta Q$ implies:

1. if $P \longrightarrow P'$ then $Q \longrightarrow Q'$ and $P' \stackrel{\cdot}{\sim}_\Delta Q'$;
2. for each channel a, if $P\downarrow_a$ then $Q\downarrow_a$.

Two processes P and Q are *barbed* Δ-*congruent*, written $P \sim_\Delta Q$, if $C[P] \stackrel{\cdot}{\sim}_\Gamma C[Q]$ for each sorting Γ and (Γ,Δ)-context C.

In the basic π-calculus, 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.

5.2. Theorem: Let the sorting $\Gamma_\pm$ and the processes $P_\pm$ and $Q_\pm$ be obtained from the sorting Γ and the processes P and Q by replacing all the I/O tags in Γ, P, and Q with the tag $\pm$. Then $P_\pm \sim_{\Gamma_\pm} Q_\pm$ implies $P \sim_\Gamma Q$.

(Here, we are regarding Milner's sorting as a fragment of our system; the connection is developed more precisely in the full version of the paper.) Thus, any equality that can be proved in the basic π-calculus — e.g., the equality 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: processes that are not equivalent in the basic π-calculus may be made equivalent by replacing $\pm$ tags with $-$ or $+$.

5.1 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 [12] as well as in the proof of representability of higher-order π-calculus in first-order π-calculus [23]. 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:

$$\begin{aligned} P_1 &\stackrel{def}{=} (\overline{b}\langle m\rangle|Q)\{m := R\} \\ P_2 &\stackrel{def}{=} \overline{b}\langle m\rangle\{m := R\}|Q\{m := R\} \end{aligned} \quad (1)$$

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 by 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 b a sort such that values received from b can only be used for output, as we did in our encoding of the lazy λ-calculus. Formally, let P be a subexpression of a process expression $C[P]$, where $\Gamma \vdash C[P] : ok$. We say that m *is used (only) as a trigger in* P if $\Delta \vdash P : ok$, where Δ is obtained from Γ_C by replacing the outermost tag of m's sort with $+$. We maintain the notation $P\{m(\tilde{x}) := R\}$ in our sorted calculus, eliding the sorts of m and $\tilde{x}$.

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

5.1.2. Theorem: [Second replicator theorem] If 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 outside of the syntactic scope of P, Q, and R as long as each such transmission places an appropriate static restriction on their possible use by the receiver.

5.2 Encoding the call-by-value λ-calculus

In the original version of his paper on Functions as Processes [12], Milner presented two candidates, $\mathcal{V}$ and $\mathcal{V}'$, for the encoding of call-by-value λ-calculus [19] in π-calculus. The encoding $\mathcal{V}'$ is more efficient than $\mathcal{V}$, since in $\mathcal{V}$ the number of steps required to simulate a β-reduction may increase as the computation proceeds. But the proofs in [12] were only given for $\mathcal{V}$; the analysis of $\mathcal{V}'$ was left open. Later, Sangiorgi [23] showed that in the standard π-calculus, β-reduction is not valid for $\mathcal{V}'$ — that is, $\mathcal{V}'[\![(\lambda x.M)\lambda y.N]\!]\langle p\rangle$ and $\mathcal{V}'[\![M\{{}^{\lambda y.N}/x\}]\!]\langle p\rangle$ are not necessarily equivalent. (The final version of [12], which appeared in the *Journal of Mathematical Structures*, was written after the results in [23] were known and presents only the encoding $\mathcal{V}$.)

Intuitively, the problems with $\mathcal{V}'$ are similar to those arising in equation (1) above: the external environment may obtain a trigger and then use it in incorrectly (i.e. in input position). Aside from this possibility, $\mathcal{V}'$ yields a precise operational correspondence between λ-terms and their process encodings, and one intuitively expects $\mathcal{V}'$ to be correct. The use of I/O information gives us a way to formally express this intuition by refining the translation as we did in Section 4 for the lazy λ-calculus; our refinement of the encoding $\mathcal{V}'$ validates β-reduction. The proof of this fact, which uses our improved replicator theorems, appears in the full version of the paper.

6 Extensions

Most 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 $[a = b]P$, which allows P to proceed only if a and b 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 [14, 18].) 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. A better 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; this refinement appears to be sufficient to recover the properties of the calculus without matching.

A more significant extension involves adding higher-order communication — communication of processes and abstraction of processes on both channels and processes. Following Sangiorgi [23], we can enrich our language of sorts to include descriptions of channels carrying processes — e.g. $(ok)^{\pm}$ — channels carrying processes abstracted on processes — e.g. $(ok{\rightarrow}ok)^{\pm}$ — 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 [3]. We conjecture that the soundness of typing can be shown via a straightforward translation from the enriched system of sorts into the first-order sorts described above, following the structure of Sangiorgi's translation of the higher-order π-calculus into first-order π-calculus.

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 [10] or the polymorphic λ-calculus [9, 20] 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:

$$\begin{array}{lcl} S & ::= & (A_1..A_m;\, S_1..S_n)^I \\ & | & A \\ & | & \mu A.\ S \end{array}$$

In a channel sort of the form $(A_1..A_m;\ S_1..S_n)^I$, the type parameters $A_1..A_n$ may appear in the sorts $S_1..S_n$ — 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 $a\ :\ (A;\, (A)^+,\, A)^-$ precisely describes the constraints respected by the process $a(A;\ x{:}(A)^+,\ y{:}A)\,.\,\overline{x}\langle y\rangle\,.\,\mathbf{0}$, 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 [24].)

Acknowledgements

This research was begun while both authors were visitors at INRIA-Roquencourt, in projects Formel and Para, and was partially supported by ESPRIT Basic Research Action "TYPES" and "CONFER." Our

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

References

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

[2] 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.

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

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

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

[6] 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.

[7] 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.

[8] 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.

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

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

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

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

[13] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, LFCS, Dept. of Comp. Sci., Edinburgh Univ., October 1991. *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991.

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

[15] 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.

[16] F. Nielson. The typed λ-calculus with first-class processes. In *Proc. PARLE '89*, volume 366 of *Lecture Notes in Computer Science*. Springer Verlag, 1989.

[17] O. Nierstrasz. Towards an object calculus. In M. Tokoro, O. Nierstrasz, P. Wegner, and A. Yonezawa, editors, *ECOOP '91 Workshop on Object Based Concurrent Programming,* Geneva, Switzerland, 1991, volume 612 of *Lecture Notes in Computer Science*. Springer Verlag.

[18] Joachim Parrow and Davide Sangiorgi. Algebraic theory for name-passing calculi. In preparation, 1993.

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

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

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

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

[23] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, Department of Computer Science, University of Edinburgh, 1992. To appear.

[24] David N. Turner, 1992. Ph.D. thesis, LFCS, University of Edinburgh. In preparation.

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