Linearity and the Pi-Calculus

Naoki Kobayashi, University of Tokyo
Benjamin C. Pierce, University of Cambridge
David N. Turner, University of Glasgow

July 18, 1995

Abstract

A static type system with "use-once" channel types, inspired by linear logic, is applied to a pure calculus of processes and message passing. Applications of this type system include static detection of errors in concurrent programs, compiler optimizations, and more natural program equivalences. After developing standard results such as soundness of typing, we focus on equivalences, adapting the standard notion of barbed bisimulation to the linear setting and showing how reductions on linear channels induce a useful "partial confluence" of process behaviors.

1 Introduction

A long line of formal systems, from Hewitt's Actors [Hew77, Agh86] to modern process calculi such as Milner, Parrow, and Walker's pi-calculus [MPW92, Mil91], have popularized the idea that a wide range of concurrent programming idioms can be modeled by simple processes exchanging messages on channels. Besides the applications of such calculi in specification and verification of concurrent systems, they have been used as the basis for concurrency features in numerous programming languages (e.g. [Car86, GMP89, Rep91]), as foundations for theoretical study of language features like concurrent objects (e.g. [Jon93, Wal95, HT91, Vas94]), and more recently as core programming languages in their own right [PT95a, PT95b].

The encoding of functions as processes in the pi-calculus (cf. [Mil90]) illustrates the "feel" of a pure message-passing calculus. Suppose that *plusone* is a communication channel and that there is a process where that repeatedly reads pairs $[i, r]$ from *plusone*, with i a number and r a channel, and responds to each pair by sending the message $[i + 1]$ on r. We may speak of the channel *plusone* as the "location" of the incrementing process, since *plusone* is the only means by which other processes can refer to it. To build another function using this one, suppose we have another channel *plustwo* to serve as the location for the new process. We create a process that repeatedly reads pairs $[j, s]$ from *plustwo* and uses *plusone* twice to calculate $j + 2$ and send it back along s:

$$\begin{array}{l}
\mathit{plustwo}?^*[j,s].\\
\quad (\nu\ r_1, r_2)\\
\quad\quad (\ \mathit{plusone}![j, r_1]\\
\quad\quad \mid r_1?[k].\\
\quad\quad\quad\quad \mathit{plusone}![k, r_2]\\
\quad\quad\quad \mid r_2?[l].\, s![l])
\end{array}$$

The replicated input expression $plustwo?^*[j, s]. \ldots$ describes a process that, each time it receives a pair $[j, s]$ on the channel *plustwo*, starts a fresh copy of its body running in parallel with all the other processes in the system. Each copy of the body (i.e., each invocation of the function) create two fresh channels, r_1 and r_2, using the channel-creation operator ν. Then it sends the message $[j, r_1]$ on the channel *plusone* (written $x![j, r_1]$) and, again in parallel, waits to receive a response $[k]$ on r_1 (written $r_1?[k]. \ldots$). When this arrives, it sends the message $[k, r_2]$ on *plusone* and waits for the response $[l]$ along r_2. Finally, it sends $[l]$ back along the result channel s to the original caller.

The flexibility of this small set of primitives gives the pi-calculus and related systems an attraction similar to that enjoyed by the lambda-calculus in the context of sequential computation, both as objects of theoretical study and as bases for language design and implementation. However, such generality has a cost: the pi-calculus is also an extremely low-level notation, so that encodings of higher-level structures must throw away potentially useful information.

In the example above, for instance, we used the convention that a function is a process reading messages from some channel f, while callers of the function are processes that send messages on f. But there is nothing in the notation that prevents another process from disrupting this protocol, either maliciously or accidentally, by *reading* from f instead of writing to it. The asymmetry of roles between function and caller has been lost. This lack of structure can lead to errors in more complex programs, and blocks many useful optimizations and program transformations.

A common response to this observation is to consider larger calculi with functions as primitives. This ensures, at some cost in parsimony, that functions can be compiled efficiently and that the expected high-level rules for reasoning (either formally or informally) actually apply. A more radical approach — the one we are exploring here — is to program in the pure pi-calculus and recover information about special "modes of usage" by static typing.

The simplest type systems for the pi-calculus [Mil91] just track the arities of channels, so as to prevent situations where the tuple of arguments provided by a sender is not the same width as the tuple of bound variables in the receiver, as in $x![y, z] \mid x?[a, b, c]. P$. Each channel x is assigned a type of the form $\updownarrow[T_1, \ldots, T_n]$, read "channel carrying $T_1, \ldots, T_n$," describing the number of values that must be included in each message sent along x, as well as their types. For example, the channel *plustwo* would be given type $\updownarrow[Num, \updownarrow[Num]]$, since it is used to communicate pairs where the first element is a number and the second is a channel that is used to send a single number.

This type system can be extended in many ways, for instance by adding polymorphism [Gay93, VH93, Tur95]. Of more interest here, however, is the possibility of *refining* it so that the types of channels carry more information about how they are used. For example, we can straightforwardly capture the directionality of communications on channels like *plustwo* by introducing two new channel types: $\uparrow[T_1, \ldots, T_n]$ for channels that (in a given context) can only be used to send tuples of type $[T_1, \ldots, T_n]$, and $\downarrow[T_1, \ldots, T_n]$ for channels that can only be used to receive $[T_1, \ldots, T_n]$-tuples [PS95, Ode95]. From the point of view of the process implementing the plus-two function, the type of *plustwo* now becomes $\downarrow[Num, \uparrow[Num]]$ — that is, it can only be used to read messages, where each message must consist of a number and a channel that can be used to send a response. From the point of view of the users of this function, *plustwo* has type $\uparrow[Num, \uparrow[Num]]$: it can only be used to send messages, and each message must again consist of a number and a channel that can be used for sending numbers.

This refinement is not only useful for preventing programming mistakes; it also yields more powerful techniques for reasoning about programs. For example, in an early version of [Mil90], Milner had proposed two encodings of the call-by-value lambda-calculus into the pi-calculus. Unfortunately, the simpler of the two encodings turned out not to preserve beta-equivalence. There were lambda-terms M and N such that M and N were equivalent, but their translations $[\![M]\!]$ and $[\![N]\!]$ did not behave the same in every process context: there were "silly" contexts that misused $[\![M]\!]$ and $[\![N]\!]$ by communicating with them in the wrong direction and breaking the invariant of the encoding. Pierce and Sangiorgi [PS95] showed how the validity of beta-equivalence could be established by typing the encoding using input-only and output-only channels. In essence, refining the type system reduces the number of contexts in which a given process can correctly be placed, thus making it easier for two processes to be equivalent.

However, some useful information cannot be captured by simple directional channel types. For example, we cannot express the fact that the result channel that is passed to a function is used just once. In other words, without knowing more about *plusone*, we are not justified in replacing the plus-two process above with a more efficient "tail-call optimized" form:

$$\begin{array}{l} plustwo?^{*}[j,s]. \\ \quad (\nu\ r)\ (plusone![j,r] \mid r?[k].\, plusone![k,s]) \end{array}$$

In this paper, we propose a further refinement, in which channel types carry not only *polarities*, which determine in which directions they can be used, but also *multiplicities*, which control how many times they can be used. We write types in the form $p^m[\tilde{T}]$, where p can be $\updownarrow$, $\downarrow$, or $\uparrow$ as before, or else $|$, indicating that the channel has been "fully consumed" and cannot be used for communication in either direction; and where m can be either ω, which allows the channel to be used an unlimited number of times (as before), or else 1, indicating that it should be used exactly once. We call channels with multiplicity 1 *linear*. For example, the type $\uparrow^1[Num]$ describes a channel that can be used exactly once to send a message containing a number, while $\updownarrow^1[Num]$ describes a channel that can be used once for input and once for output.

In a pi-calculus with linear channel types, we can take the type of *plusone* and *plustwo*, from the point of view of callers, to be $\uparrow^{\omega}[Num, \uparrow^1[Num]]$, capturing the fact that each of them can be invoked any number of times but that the result channel in each invocation is only used once. Under this more accurate typing, the original and optimized versions of the plus-two function are provably equivalent, in the sense that they behave the same in all well-typed contexts, as we show in Section 5.

Indeed, not only are some useful equivalences gained in the presence of linear types: even the mechanics of reasoning about processes are augmented with new tools. The key intuition behind these is that linear communication is *dyadic*: two processes communicating over a linear channel can neither interfere with nor be interfered with by any other communication going on in the system. This stands in sharp contrast to the usual situation in the pi-calculus, where the possibility that two senders can be in competition for one receiver or vice versa introduces an essential element of non-determinism. The expression $x![y] \mid x![z] \mid x?[a].\, a![\,]$ can reduce either to $x![z] \mid y![\,]$ or to $x![y] \mid z![\,]$, and these can never be reduced further so as to reach a common point again. On the other hand, there can never be two senders or two receivers on a linear channel, and so such race conditions can never arise. Indeed, linear communications exhibit a *partial confluence* property reminiscent of the Church-Rosser property of the lambda-calculus, as we show in Section 4.

Linear typing potentially enables a variety of compiler optimizations. For example, the heap space allocated to a linear channel can immediately be re-used by the receiver after a communication, instead of being reclaimed by garbage collection. Also, the implementation of communication on a linear channel can be simplified, since such channels can only be in one of two states when a communication occurs: there may be either zero or one processes of opposite polarity waiting to communicate, but never more than one and never another one of the same polarity. Work is underway on exploring these optimizations in a compiler for a linear variant of the Pict programming language [PT95a, PT95b].

Linear channel types can also help detect programming mistakes. Suppose, for example, that we have written a lock manager that mediates between a group of processes, each containing a "critical section" during which it needs to know that none of the other processes are executing their own critical sections. Before entering its critical section, each process sends a message on a channel *acquireLock* and waits for a response message from a central "lock manager," indicating that it is safe to proceed. This go-ahead message includes a channel *done*, on which the process must send a message when it completes its critical section, signalling to the lock manager that it may give the lock to another process. If the critical section is complex, it can be easy to forget to signal on *done* or to signal twice. If *done* is given a linear type, our rules will ensure that each critical section uses *done* exactly once. However, we must be slightly careful about what this assertion means. Because typing is preserved by reduction (as we show in Section 4), a process containing exactly one use of *done* must either communicate on *done* and evolve to a process that cannot use *done*, or else must make some other step and become a process that still contains one use of *done*. This guarantees that *done* will never be used twice, but not that *done* will eventually be used, since the subprocess in which *done* occurs might diverge or deadlock before it actually communicates on *done*. For example, the process $(\nu z{:}\updownarrow^{\omega}[\,])\, z?[\,].\, \mathit{done}![\,]$ creates a fresh channel z, reads from it, and then sends on *done*. Since z is fresh, there can never be any sender on z, and the send on *done* will never be reached. This possibility of *partial* deadlock or divergence in the pi-calculus gives linear typing a slightly different force than in lambda-calculus, where a linear value *must* be used unless the whole program fails to yield any result.

Our goals in this paper are (1) a precise definition of a pi-calculus with linear types and (2) formal justifications of the claims that we have sketched here about typing and program equivalences. The most novel aspects of this development are the technical treatment of linear channel types (which, since they have both input and output ends, actually comprise *two* linear capabilities) and the study of behavioral equivalences in the presence of linear types.

After introducing some notational conventions in Section 2, we define the typing relation (Section 3) and show that it is preserved by reduction (Section 4); along the way, we remark on properties such as confluence. In Section 5, we adapt the standard notion of *barbed congruence* to the present setting. Section 6 discussed related work. We omit proofs in this summary.

2 Notational Preliminaries

The operators introduced in the examples above constitute a fragment of the polyadic pi-calculus [Mil91] lacking the choice and matching operators (like Milner's mini pi-calculus [Mil90]) and in which communication is asynchronous in the sense that an output expression is a bare message with no "body" to be executed after the message has been sent (as in Honda and Tokoro's ν-calculus [HT91]). We have argued elsewhere for the pragmatic virtues of this calculus [PT95a], but the techniques we develop should be applicable to any similar system. (The details of the behavioral properties will vary with the choice of calculus, of course.) We also provide booleans and conditional expressions as primitives, since their typing behavior interacts with linearity information in a slightly special way.

The metavariable m ranges over the *multiplicities* 1 and ω. The metavariable p ranges over *polarities*. Formally, these are subsets of the set of capabilities {i,o}, but for readability we use the abbreviations $\updownarrow = \{i, o\}$, $\uparrow = \{o\}$, $\downarrow = \{i\}$, and $| = \{\}$. The set of types is:

T	$::=$	$p^m[\tilde{T}]$	channel types
		Bool	booleans

Sequences of types, etc. are usually written $\tilde{T}$ instead of $T_1, \ldots, T_n$, for brevity. A type of the form $p^1[\tilde{T}]$, where $p \neq |$, is called *limited*. All other types are *unlimited*.

The set of process expressions is:

P	$::=$	$P \mid Q$	parallel composition
		$x![\tilde{y}]$	output atom
		$x?[\tilde{y}].\,P$	input prefix
		$x?^*[\tilde{y}].\,P$	replicated input prefix
		$(\nu x{:}T)\,P$	channel creation
		if x then P else Q	conditional

To avoid writing too many parentheses, we give ν a higher precedence than $|$ and make the bodies of input expressions extend as far to the right as possible, so that $x?^*[\,].\,(\nu y{:}\updownarrow^\omega[\,])\,x![\,] \mid y![\,]$ means $x?^*[\,].\,(((\nu y{:}\updownarrow^\omega[\,])\,x![\,]) \mid y![\,])$. The "inert process" is defined as $\mathbf{0} = (\nu x{:}\updownarrow^\omega)\,x![\,]$.

The free and bound variables of a process expression are defined in the usual way, with x having scope P in $(\nu x{:}T)\,P$, and $\tilde{y}$ having scope P in $x?[\tilde{y}].\,P$ and $x?^*[\tilde{y}].\,P$.

A *type environment* Γ is a set of bindings of distinct variables to types, containing at least the bindings true:Bool and false:Bool. By convention, we write type environments as $x_1{:}T_1, \ldots, x_n{:}T_n$ or just $\tilde{x}{:}\tilde{T}$, omitting the bindings of the booleans. The domain of a type environment, written $dom(\Gamma)$, is the set of variables it binds. Type environments are extended with bindings for new variables by writing $\Gamma, x{:}T$ or $\Gamma, \tilde{x}{:}\tilde{T}$. We write $\Gamma(x) = T$ to mean that $\Gamma = \Gamma_1, x{:}T$. A type environment is unlimited if each of its bindings is.

We always consider a process relative to a particular type environment, whose domain must contain all its free variables. We regard names of bound variables as inessential, and perform silent alpha-conversion as necessary to ensure that the name of a bound variable is always different from the names of free variables and other bound variables. This view (which amounts to regarding named variables as shorthand for an underlying "deBruijn-indexed" calculus [dB72]) requires some notational work to maintain, but helps avoid confusion.

The *combination* of two types T_1 and T_2, written $T_1 + T_2$, is defined as follows:

$$\begin{array}{lcll} \text{Bool} + \text{Bool} & = & \text{Bool} & \\ p^\omega[\tilde{T}] + q^\omega[\tilde{T}] & = & (p \cup q)^\omega[\tilde{T}] & \\ p^1[\tilde{T}] + q^1[\tilde{T}] & = & (p \cup q)^1[\tilde{T}] & \text{if } p \cap q = \emptyset \end{array}$$

Similarly, the *difference* of two types T_1 and T_2, written $T_1 - T_2$, is:

$$\begin{array}{lcll} \text{Bool} - \text{Bool} & = & \text{Bool} & \\ p^\omega[\tilde{T}] - q^\omega[\tilde{T}] & = & p^\omega[\tilde{T}] & \text{if } q \subseteq p \\ p^1[\tilde{T}] - q^1[\tilde{T}] & = & (p \setminus q)^1[\tilde{T}] & \text{if } q \subseteq p \end{array}$$

(Note that $T - T$ is not necessarily a type with $|$ polarity: we use $-$ to remove a *single instance* of a capability; if the capability can be used an unlimited number of times — i.e., if the type is Bool or an unlimited channel type — then it is unchanged by the operation.) The $+$ operator is extended pointwise to type environments with equal domains:

$$(\Gamma_1, x{:}T_1) + (\Gamma_2, x{:}T_2) = (\Gamma_1 + \Gamma_2), x{:}(T_1 + T_2)$$

We also use $+$ and $-$ to "add and remove capabilities" from a single existing binding in an environment. When $\Gamma = \Gamma', x{:}U$, we write $\Gamma + x{:}T$ to mean $\Gamma, x{:}U + T$ and $\Gamma - x{:}T$ to mean $\Gamma, x{:}U - T$. To account for the effect of "using up" a value of type T, we write $Rem(T) = T - T$ for the *remnant* of T. The *remnants* of a type environment Δ, written $Rems(\Delta)$, is the set of all the type environments that can be obtained from Δ by replacing some subset $x_1{:}T_1, \ldots, x_n{:}T_n$ of its bindings by their remnants $x_1{:}Rem(T_1), \ldots, x_n{:}Rem(T_n)$.

3 Typing

The technical intuitions underlying linear channel types are based on Girard's linear logic [Gir87, GLT89], a "resource conscious" refinement of classical logic, and on numerous proposals for functional and logic-programming languages based on linear logic (e.g. [Abr93, Wad91, Hod92, Bak92, Mac94, TWM95]). A crucial element in all of these systems is the careful treatment of the typing environments under which expressions are judged to be well typed. For example, to form a pair of two values v and w under some type environment Γ, we do not simply check that v and w are well typed in Γ, as in conventional type systems:

$$\frac{\Gamma \vdash v \in V \qquad \Gamma \vdash w \in W}{\Gamma \vdash \langle v, w\rangle \in V \times W}$$

Instead, we split the type environment in the conclusion into two parts, representing the resources consumed in typing v and w separately:

$$\frac{\Gamma_1 \vdash v \in V \qquad \Gamma_2 \vdash w \in W}{\Gamma_1 + \Gamma_2 \vdash \langle v, w\rangle \in V \times W}$$

Similarly, in a linear type system for the pi-calculus, we must split the type environment in the rule for parallel composition,

$$\frac{\Gamma_1 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 + \Gamma_2 \vdash P \,|\, Q} \qquad \text{(E-PAR)}$$

so that typing $P \,|\, Q$ consumes the sum of the resources used in the proofs that P and Q are well typed. The typing judgement $\Gamma \vdash P$ thus acquires additional force: "P is well typed under the typing assumptions in Γ and, moreover, uses each linear capability in Γ exactly once."

Our presentation of the typing relation differs in one important technical detail from the standard formulation of linear logics and type systems. Normally, when a type environment is "split," only those variable bindings that are actually used in the right and left subderivations are included in the respective "halves" of Γ. For example, to check $x{:}\updownarrow^1[],\ y{:}\updownarrow^1[] \vdash x![] \,|\, y![]$, we check $x{:}\updownarrow^1[] \vdash x![]$ and $y{:}\updownarrow^1[] \vdash y![]$. But this mixes two issues that are better kept distinct: resolving the scopes of variables and tracking their usage. We find it clearer to split type environments by removing just capabilities, leaving variable bindings in place, so that the combination operator $+$ is always applied to environments with equal domains. To type $x![] \,|\, y![]$ in the environment $x{:}\updownarrow^1[],\ y{:}\updownarrow^1[]$, we check $x![]$ in the environment $x{:}\updownarrow^1[],\ y{:}|^1[]$ and $y![]$ in the environment $x{:}|^1[],\ y{:}\updownarrow^1[]$.

To ensure that all the linear bindings in the environment get used somewhere, we check, at all the leaves of the typing derivation that all the bindings in the current environment have either multiplicity ω or else polarity $|$:

$$\frac{\Gamma \;\; \text{unlimited}}{\Gamma + x{:}\uparrow^m[\tilde{T}] + \tilde{y}{:}\tilde{T} \vdash x![\tilde{y}]} \tag{E-Out}$$

That is, if all the bindings in Γ are unlimited, then $x![\tilde{y}]$ is well typed in Γ augmented (in case x or any of the $\tilde{y}$ are linear) with the capabilities consumed by the output itself. Note that the capabilities expressed by the types $\tilde{T}$ are among those consumed, since the $\tilde{y}$ are being passed to the receiver of the communication. Conversely, an input expression is well typed if its body is well typed in an environment where the capability to input from x has been removed (if it is linear) and the capabilities associated with the values $\tilde{z}$ to be read from x have been added.

$$\frac{\Gamma, \tilde{z}{:}\tilde{T} \vdash P}{\Gamma + x{:}\downarrow^m[\tilde{T}] \vdash x?[\tilde{z}].\, P} \tag{E-In}$$

The rule for replicated input is nearly the same, except that the type of x must be unlimited and, since the body may be instantiated many times, we must ensure that the common part of the type environment does not contain any linear bindings at this point.

$$\frac{\Gamma, \tilde{z}{:}\tilde{T} \vdash P \qquad \Gamma \;\; \text{unlimited}}{\Gamma + x{:}\downarrow^\omega[\tilde{T}] \vdash x?^*[\tilde{z}].\, P} \tag{E-RIn}$$

The rule for ν-expressions simply moves the binding for the new variable from the ν to the type environment, first verifying that it is a reasonable type for a channel declaration .

$$\frac{\Gamma, x{:}p^m[\tilde{T}] \vdash P \qquad p = | \text{ or } p = \updownarrow}{\Gamma \vdash (\nu x{:}p^m[\tilde{T}])\, P} \tag{E-New}$$

The rule for checking conditional expressions is a little different from the others. (It corresponds to one of the "additive" operators of linear logic, while the rest of our operators belong to the "multiplicative" part.) We know that the "then" and "else" parts of an if-clause will never both be executed, so any linear bindings in the environment should be used up by both parts, instead of being allocated to one part or the other.

$$\frac{\Gamma \vdash P_1 \qquad \Gamma \vdash P_2}{\Gamma + b{:}\text{Bool} \vdash \text{if } b \text{ then } P_1 \text{ else } P_2} \tag{E-If}$$

The typing relation $\Gamma \vdash P$, then, is the least relation closed under the foregoing rules. A process well typed under Γ is called a Γ-process.

Although we are not concerned with algorithmic issues in this paper, it is worth mentioning that these rules give rise to a typechecking algorithm in a fairly straightforward way. Instead of "guessing" how to split the capabilities in the environment in rules like E-Par, we pass the whole environment to the lefthand premise, "crossing off" capabilities as they are used up and passing whatever is left to the right-hand premise.

4 Operational Semantics

Following [Mil91], the operational semantics of processes is presented in two steps. First, we define a *structural congruence* relation $P \equiv Q$, capturing the fact that, for example, the order of the branches in a parallel composition has no effect on its behavior. Then we define a *reduction relation* $\Gamma \vdash P \longrightarrow Q$, specifying how processes evolve through communications between their subprocesses.

Structural congruence plays an important technical role in simplifying the definition of the reduction relation. For example, we intend that the processes $x![] \mid x?[].\,P$ and $x?[].\,P \mid x![]$ both reduce to P. Since these two are structurally congruent, it suffices to write the reduction rule only for the first case and to stipulate, in general, that if P contains some possibility of communication, then so does any expression structurally congruent to P.

The first two structural congruence rules state that $\mid$ is commutative and associative.

$$P \mid Q \equiv Q \mid P \qquad \text{(S-Commut)}$$

$$(P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad \text{(S-Assoc)}$$

The third rule is called *scope extrusion* in the pi-calculus literature:

$$((\nu x{:}T)\,P) \mid Q \equiv (\nu x{:}T)\,(P \mid Q) \quad \text{if } x \text{ not free in } Q \qquad \text{(S-Extr)}$$

Informally, it says that the scope of the channel x, which starts out private to the process P, can be extended to include Q. The side-condition ensures that Q does not already have a free channel named x. (This condition can always be satisifed by renaming x, if necessary, before applying the scope extrusion rule.) For example, the process $((\nu x{:}\updownarrow^{\omega}[])\,w![x]) \mid w?[z].\,z![]$ may be transformed to $(\nu x{:}\updownarrow^{\omega}[])\,(w![x] \mid w?[z].\,z![])$. Finally, for technical convenience, we allow adjacent ν-bindings to be switched:

$$(\nu x{:}T)\,(\nu y{:}U)\,P \equiv (\nu y{:}U)\,(\nu x{:}T)\,P \qquad \text{(S-Switch)}$$

Formally, the relation $\equiv$ is the least congruence closed under these four rules, meaning that $P \equiv Q$ if Q can be obtained by applying the rules any number of times, in either direction, to arbitrary subexpressions of P.

Normally, the reduction relation is written $P \longrightarrow Q$, meaning that P can evolve to Q by making a single, atomic step of communication. We use the same basic relation, but annotate it with some extra information that we will want to use later for stating properties of the semantics: a type environment (to remind ourselves that we are interested in reduction of well-typed processes, and so that we can see the way communication uses up capabilities of linear channels), a multiplicity tag m that records whether the communication that leads from P to Q occurs on a linear or an unlimited channel, and a tag α, which can be either the name of a channel, indicating where the communication occurs, or else τ, indicating that the channel on which the communication occurs is bound by a ν that is not in scope at this point. We write the whole reduction relation $\Gamma \vdash P \xrightarrow{m,\alpha} Q$.

The most important reduction rule is the one for communication. In an environment where x has a channel type of multiplicity m, the parallel composition of an output and an input on x can evolve to the body of the input-process with the arguments to the output substituted for the formal parameters. We record m and x on the arrow.

$$\Gamma, x{:}\updownarrow^m[\tilde{T}] \vdash x![\tilde{y}] \mid x?[\tilde{z}].\, P \xrightarrow{m,x} [\tilde{y}/\tilde{z}]P \tag{R-Com}$$

Similarly, an output in parallel with a replicated input can reduce to the appropriate instance of the body of the input, *plus* a fresh copy of the replicated input itself.

$$\Gamma, x{:}\updownarrow^m[\tilde{T}] \vdash x![\tilde{y}] \mid x?^*[\tilde{z}].\, P \xrightarrow{m,x} [\tilde{y}/\tilde{z}]P \mid x?^*[\tilde{z}].\, P \tag{R-RCom}$$

Next, we state that if P can reduce to Q in isolation, then it can still do so when it is placed in parallel with some other process R.

$$\frac{\Gamma_1 \vdash P \xrightarrow{m,\alpha} Q \qquad \Gamma_2 \vdash R}{\Gamma_1 + \Gamma_2 \vdash P \mid R \xrightarrow{m,\alpha} Q \mid R} \tag{R-Par}$$

Similarly, if P reduces to Q, then it can still do so when placed under a ν. However, we must divide this rule into two cases. If the channel bound by the ν is the very channel x on which the communication in P has occurred, then we change the label x on the arrow to τ and change the polarity of the type of x in the ν-binding to its "remainder"; this has the effect of changing its polarity to $\mid$ if its multiplicity is linear, and has no effect if its multiplicity is unlimited.

$$\frac{\Gamma, x{:}T \vdash P \xrightarrow{m,x} Q}{\Gamma \vdash (\nu x{:}T)\, P \xrightarrow{m,\tau} (\nu x{:}\mathit{Rem}(T))\, Q} \tag{R-New1}$$

On the other hand, if the communication in P occurs on some other channel, then we simply pass the communication through the ν unaltered.

$$\frac{\Gamma, x{:}T \vdash P \xrightarrow{m,\alpha} Q \qquad x \neq \alpha}{\Gamma \vdash (\nu x{:}T)\, P \xrightarrow{m,\alpha} (\nu x{:}T)\, Q} \tag{R-New2}$$

An if-expression reduces to either its then-branch or its else-branch, depending on its guard.

$$\Gamma \vdash \text{if true then } P \text{ else } Q \xrightarrow{\omega,\tau} P \tag{R-IfT}$$

$$\Gamma \vdash \text{if false then } P \text{ else } Q \xrightarrow{\omega,\tau} Q \tag{R-IfF}$$

Finally, if P can be rearranged to some P' using the structural congruence laws so that a communication is possible in P', then the same communication is possible in P.

$$\frac{P \equiv P' \qquad \Gamma \vdash P' \xrightarrow{m,\alpha} Q' \qquad Q' \equiv Q}{\Gamma \vdash P \xrightarrow{m,\alpha} Q} \tag{R-Cong}$$

When we want to be explicit about the typing environment in force after a reduction has occurred, we write $\Gamma, x{:}T \vdash P \xrightarrow{m,\alpha} P' \dashv \Gamma'$, where

$$\Gamma' = \begin{cases} \Gamma & \text{if } \alpha = \tau \\ \Gamma - x{:}T & \text{if } \alpha = x. \end{cases}$$

The *sine qua non* of an operational semantics for a typed language is that well-typedness must be preserved under reduction.

4.1 Theorem [Subject Reduction]: If $\Gamma \vdash P$ and $\Gamma \vdash P \xrightarrow{m,\alpha} P' \dashv \Gamma'$, then $\Gamma' \vdash P'$.

Because our operational semantics is careful to "cross out" linear capabilities as it uses them, the subject reduction theorem together with the following theorem implies that that linear channels are used at most once, in addition to the standard observation that there can never be any arity mismatch during the execution of a well-typed program.

4.2 Theorem [Run-time safety]: A well-typed term contains no immediate possibilities for communication failure or misuse of capabilities. Suppose $\Gamma \vdash P$ and $P \equiv (\nu \tilde{w}{:}\tilde{U})\,(Q \mid R)$. Then:

1. If $Q = x![y_1, \ldots, y_m] \mid x?[z_1, \ldots, z_n].\,Q$ or $Q = x![y_1, \ldots, y_m] \mid x?^*[z_1, \ldots, z_n].\,Q$, then $m = n$ and the binding of x (in either P or Γ) has polarity $\updownarrow$.
2. If $Q = x![\tilde{y}] \mid x![\tilde{z}]$, then the binding of x has multiplicity ω.
3. If $Q = x?[\tilde{y}].\,Q_1 \mid x?[\tilde{z}].\,Q_2$, then the binding of x has multiplicity ω.
4. If $Q = x?^*[\tilde{y}].\,Q_1$, then the binding of x has multiplicity ω.

Finally, it is worth mentioning that the type annotations and other labels in this semantics are used just for bookkeeping, and do not play any role in determining the possible behaviors of well-typed process expressions.

4.3 Theorem [Correspondence with untyped semantics]: On well-typed processes, our operational semantics allows the same reductions as the standard untyped semantics, which can be obtained by erasing all types, type environments, and labels on arrows from ours.

Now we come to the first result that shows how linearity affects reasoning about program behavior. If P is a process that contains two possible communications, once of which is on a linear channel, then they may be performed in either order to reach the same result. In a word, linear reduction is *confluent* with respect to all other reductions.

4.4 Theorem [Partial confluence]: If $\Gamma \vdash P \xrightarrow{1,x} P'$, then for any $\Gamma \vdash P \xrightarrow{m,\alpha} Q \dashv \Gamma'$ with $\alpha \neq x$, there is some R such that $\Gamma' \vdash Q \xrightarrow{1,x} R$. Moreover, $\Gamma \vdash P \xrightarrow{1,x} P' \xrightarrow{m,\alpha} R$.

5 Bisimilarity

Our last job is to formalize process equivalence in the presence of linear types, so that we can check that equivalences like those claimed in the introduction really hold. We adapt Milner and Sangiorgi's notion of *barbed bisimilulation* [MS92, San92], which we find suited to the job for several reasons. First, it is naturally a relation on *typed* processes, which is crucial here. Second, it is a fairly "modular" definition, in the sense that barbed bisimilarities for many different calculi can be formulated almost identically; this facilities comparison, and helps give confidence that the definition is natural. And finally, like all varieties of bisimilarity, it comes with a convenient coinductive proof technique (after some work to establish necessary properties, such as Theorem 5.5).

We begin by defining what "tests" can be made of a process by an external observer, relative to a certain type environment Γ.

5.1 Definition: We say that P *exhibits the barb a under type environment* Γ, written $P \Downarrow_a^{\Gamma}$, if P has the immediate possibility of performing an input or output action on the channel a and if the type of a in Γ is such that the observer can supply the other side of the communication — i.e., it is not the case that a is linear and both its input and output capabilities are used by P. Formally, $P \Downarrow_a^{\Gamma, a:T}$ iff either

1. $P \equiv (\nu \tilde{x}{:}\tilde{U})\,(a![\tilde{y}] \mid Q)$ and either $T = p^{\omega}[\tilde{T}]$ or $T = \uparrow^1[\tilde{T}]$; or
2. $P \equiv (\nu \tilde{x}{:}\tilde{U})\,(a?[\tilde{y}].\,R \mid Q)$ and either $T = p^{\omega}[\tilde{T}]$ or $T = \downarrow^1[\tilde{T}]$; or
3. $P \equiv (\nu \tilde{x}{:}\tilde{U})\,(a?^*[\tilde{y}].\,R \mid Q)$ and $T = p^{\omega}[\tilde{T}]$.

(In each case, our assumption that all bound variables have distinct names ensures that the channel a cannot be one of the $\tilde{x}$.)

Next, we say that two processes are bisimilar if they have the same sets of observable actions and if, for each step that one can make, the other can make a step and reach a state that is again bisimilar. The only slight complication is that we cannot require that the two processes be well typed in exactly the same type environment, since each linear communication removes capabilities from the type environment, and we want to allow the possibility that two processes are bisimilar even though they may use up their linear resources in different orders. We deal with this by allowing the two to be typed in different environments, provided that both environments can be obtained by removing capabilities from some common environment. Formally, we first define what it means for an arbitrary relation on pairs of processes (and associated type environments) to be a bisimulation. Two processes are then said to be bisimilar if they are related by any bisimulation.

5.2 Definition [Strong barbed bisimulation]: A relation

$$\mathcal{R} \subseteq \{\, ((\Delta_P, P), (\Delta_Q, Q)) \;\mid\; \Delta_P, \Delta_Q \in \mathit{Rems}(\Delta) \;\wedge\; \Delta_P \vdash P \;\wedge\; \Delta_Q \vdash Q \,\}$$

is a Δ-*bisimulation* if, whenever $((\Delta_P, P), (\Delta_Q, Q)) \in \mathcal{R}$,

1. $P \Downarrow_a^{\Delta}$ iff $Q \Downarrow_a^{\Delta}$;
2. $\Delta_P \vdash P \longrightarrow P' \dashv \Delta'_P$ implies $\Delta_Q \vdash Q \longrightarrow Q' \dashv \Delta'_Q$ and $((\Delta'_P, P'), (\Delta'_Q, Q')) \in \mathcal{R}$ for some Q'; and
3. $\Delta_Q \vdash Q \longrightarrow Q' \dashv \Delta'_Q$ implies $\Delta_P \vdash P \longrightarrow P' \dashv \Delta'_P$ and $((\Delta'_P, P'), (\Delta'_Q, Q')) \in \mathcal{R}$ for some P'.

Two Δ-remnant processes P and Q are Δ-*bisimilar*, written $P \stackrel{\bullet}{\sim}_{\Delta} Q$, if $((\Delta_P, P), (\Delta_Q, Q)) \in \mathcal{R}$ for some $\Delta_P, \Delta_Q \in \mathit{Rems}(\Delta)$ and some Δ-bisimulation $\mathcal{R}$.

By itself, barbed bisimulation is a rather weak relation. In order to make it interesting, we need to close it under substitution and parallel composition.

5.3 Definition: Given two type environments Δ and Γ, a Δ-Γ-substitution σ is a function from $dom(\Delta)$ to $dom(\Gamma)$ such that, for each $y \in dom(\Gamma)$, the type $\Gamma(y)$ can be written $(\Sigma_{x \in dom(\Delta) \wedge \sigma(x) = y}\, \Delta(x)) + U$ for some unlimited U.

Intuitively, a Δ-Γ substitution is one that, whenever it maps some channels $x_1, \ldots, x_n \in dom(\Delta)$ to the same channel $y \in dom(\Gamma)$, ensures that y has at least the combination of the capabilities of the $\tilde{x}$.

5.4 Definition: P and Q are Δ-*precongruent*, written $P \sim^p_\Delta Q$, if, for any Δ-Γ substitution σ and $\Theta \vdash R$ such that $\Theta + \Gamma$ is well-defined, $R \,|\, \sigma P \overset{\bullet}{\sim}_{\Theta+\Gamma} R \,|\, \sigma Q$. That is, $\sim^p_\Delta$ is the largest bisimulation closed under well-typed substitutions and parallel composition. Two Δ-processes P and Q are Δ-*congruent*, written $P \sim_\Delta Q$, if $P \sim^p_\Delta Q$. That is, two processes are congruent if they are precongruent and are well typed in the same type environment.

As a simple example, it is easy to check that $x![] \,|\, y![] \,|\, x?[].y?[].\mathbf{0}$ and $x![] \,|\, y![] \,|\, y?[].x?[].\mathbf{0}$ are congruent under the environment $\Delta = x{:}\updownarrow^1[], y{:}\updownarrow^1[]$. (And that they are *not* congruent under the environment $\Delta = x{:}\updownarrow^\omega[], y{:}\updownarrow^\omega[]$.)

Fortunately, closing under parallel composition and substitution is enough: if two processes are congruent, then placing them in any context again yields congruent processes.

5.5 Theorem [Congruence]: The relation $\sim_\Delta$ is preserved by all the process constructors.

As usual, the definitions of strong barbed bisimulation and congruence have a more useful, but somewhat more complex, counterpart, called *weak bisimulation*, where we drop the requirement that the two processes being compared must proceed in lock-step and instead allow one to take zero or many steps to match a single step of the other. The definitions of *weak barbs*, *weak bisimulation*, and *weak congruence* are obtained from the strong ones in the standard way. The weak barbs of a process are the strong barbs of all of its derivatives. Weak bisimulation ($P \overset{\bullet}{\approx}_\Delta Q$) replaces the second $\longrightarrow$ in clauses (2) and (3) of Definition 5.1 with its reflexive, transitive closure, $\longrightarrow^*$. Weak precongruence ($P \approx^p_\Delta Q$) and congruence ($P \approx_\Delta Q$) are formed by closing under substitution and parallel composition.

Now the partial confluence of linear reduction (Theorem 4.4) can be expressed in a more useful way:

5.6 Theorem: If $\Delta \vdash P \xrightarrow{1,\alpha} Q$, then $P \approx^p_\Delta Q$.

This theorem essentially says that linear reduction does not change the weak congruence class of a process, so that, to show that $P \approx_\Delta Q$, it suffices to show that $P' \approx_\Delta Q$ for some linear reduct P' of P. This observation can be used to check, for instance, that the two versions of the plus-two function in the introduction are equivalent.

5.7 Example: Let $\Gamma = \mathit{plustwo}{:}\downarrow^m[\mathit{Num}, \uparrow^1[\mathit{Num}]], \mathit{plusone}{:}\uparrow^m[\mathit{Num}, \uparrow^1[\mathit{Num}]]$. Then

$$\begin{array}{ll} & \mathit{plustwo}?^*[j,s].\,(\nu r_1, r_2{:}\updownarrow^1[\mathit{Num}])\,(\mathit{plusone}![j,r_1] \,|\, r_1?[k].\,(\mathit{plusone}![k,r_2] \,|\, r_2?[l].\,s![l])) \\ \approx_\Gamma & \mathit{plustwo}?^*[j,s].\,(\nu r{:}\updownarrow^1[\mathit{Num}])\,(\mathit{plusone}![j,r] \,|\, r?[k].\,\mathit{plusone}![k,s]). \end{array}$$

6 Related Work

Our type system shares a great deal with linear type systems for lambda-calculi and related programming languages [Abr93, Wad91, Hod92, Bak92, Mac94, TWM95]. The main differences here are the extra technicalities of dealing with linear channels (which essentially comprise *two* linear capabilities), and our results about behavioral equivalences.

In the world of process calculi, there have been several papers on analyzing channel usage using *effect systems*. Nielson and Nielson [NN94] proposed a method for inferring the maximum usages of channels, while Kobayashi, Nakade, and Yonezawa [KNY95] proposed a method for approximating how many receivers can try to read from a channel at the same time, which in particular can be used to detect channels whose maximum queue length is 1. Our type-based analysis and these effect-based analysis both seem to have advantages and disadvantages. The effect-based methods can *infer* usage information, whereas currently our type system only *checks* consistency of the usage information it is given. On the other hand, in approaches based on effect analysis, channel usage is analyzed with respect to "regions," which may be aliased to an infinite number of channels, degrading the accuracy of the analysis. Although Kobayashi et al. tried to overcome this defect to some degree, the resulting analysis is difficult, and some algorithmic aspects (especially time complexity) are left open.

Finally, a type system reminiscent of ours, though not explicitly incorporating the notion of linearity, has been developed by Nestmann and Steffen [NS95] for the purpose of analyzing confluence in pi-calculus processes arising in the semantics of concurrent object-oriented programs. The same problem has also been tackled by Liu and Walker, using purely semantic techniques [LW95].

The immediate spur for our work came from a paper by Takeuchi, Honda, and Kubo [THK94] describing a modified pi-calculus whose syntax guarantees that certain channels can only be shared between two processes. This suggested that one might try to achieve the same effect in a standard pi-calculus by refining the type system.

Acknowledgements

Pierce is supported by the EPSRC. Turner is supported by EPSRC grant number (GR/J 527099), 'Save space with linear types'. We have profited from lively discussions with Robin Milner, Phil Wadler, and the Monday "Interruption Club" at Cambridge.

References

[Abr93] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1–2):3–57, April 12 1993.

[Agh86] Gul A. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.

[Bak92] Henry G. Baker. Lively linear lisp – look ma, no garbage! *ACM Sigplan Notices*, 27(8):89–98, 1992.

[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.

[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.

[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.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, Cambridge, 1989.

[GMP89] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. Facile: A Symmetric Integration of Concurrent and Functional Programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.

[Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.

[Hod92] J. S. Hodas. Lolli: An extension of λProlog with linear context management. In D. Miller, editor, *Workshop on the λProlog Programming Language*, pages 159–168, Philadelphia, Pennsylvania, August 1992.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 512 of *Lecture Notes in Computer Science*, Geneva CH, 1991. Springer-Verlag , Berlin, Heidelberg, New York, Tokyo.

[Jon93] Cliff B. Jones. A pi-calculus semantics for an object-based design notation. In E. Best, editor, *Proceedings of CONCUR'93*, LNCS 715, pages 158–172. Springer-Verlag, 1993.

[KNY95] Naoki Kobayashi, Motoki Nakade, and Akinori Yonezawa. Static analysis of communication for asynchrnous concurrent programming languages. to appear in Proceedings of International Static Analysis Symposium, Springer LNCS, 1995.

[LW95] Xinxin Liu and David Walker. Confluence of processes and systems of objects. In *Proceedings of CAAP'95*, pages 217–231. Springer, 1995.

[Mac94] Ian Mackie. Lilac: A functional programming language based on linear logic. *JFP*, 4(4):395–433, October 1994.

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

[Mil91] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991. *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991. Reprinted 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.

[NN94] Hanne Riis Nielson and Flemming Nielson. Higher-order concurrent programs with finite communication topology. In *Proceedings of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon*, pages 84–97, January 1994.

[NS95] Uwe Nestmann and Martin Steffen. Typing confluence, July 1995. Draft technical report.

[Ode95] Martin Odersky. Polarized name passing. Draft technical report, 1995.

[PS95] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 1995. To appear. A summary was presented at LICS '93.

[PT95a] Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Theory and Practice of Parallel Programming (TPPP), Sendai, Japan (Nov. 1994)*, number 907 in Lecture Notes in Computer Science, pages 187–215. Springer-Verlag, April 1995.

[PT95b] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical report in preparation; available electronically, 1995.

[Rep91] John Reppy. CML: A higher-order concurrent language. In *Programming Language Design and Implementation*, pages 293–259. SIGPLAN, ACM, June 1991.

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

[THK94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *Proceedings of PARLE'94*, pages 398–413. Springer-Verlag, 1994. Lecture Notes in Computer Science number 817.

[Tur95] David N. Turner. The π-calulus: Types, polymorphism and implementation, 1995. Ph.D. thesis, LFCS, University of Edinburgh. In preparation.

[TWM95] David N Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *FPCA*, San Diego, California, 1995.

[Vas94] Vasco T. Vasconcelos. Typed concurrent objects. In *Proceedings of the Eighth European Conference on Object-Oriented Programming (ECOOP)*, volume 821 of *Lecture Notes in Computer Science*, pages 100–117. Springer-Verlag, July 1994.

[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.

[Wad91] Philip Wadler. Is there a use for linear logic? In *Proceedings of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 255–273, 1991.

[Wal95] David Walker. Objects in the π-calculus. *Information and Computation*, 116:253–271, 1995.