Linearity and the Pi-Calculus

Naoki Kobayashi
University of Tokyo
koba@is.s.u-tokyo.ac.jp

Benjamin C. Pierce
University of Cambridge
bcp1000@cl.cam.ac.uk

David N. Turner
University of Glasgow
dnt@dcs.gla.ac.uk

Abstract

The economy and flexibility of the pi-calculus make it attractive both as an object of theoretical study and as a basis for concurrent language design and implementation. However, such generality has a cost: encoding higher-level features like functional computation in pi-calculus throws away potentially useful information. We show how a linear type system can be used to recover important static information about a process's behaviour. In particular, we can guarantee that two processes communicating over a linear channel cannot interfere with other communicating processes. This enables more aggressive optimisation of communications over linear channels and allows useful refinements to the usual notions of process equivalence for pi-calculus.

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 like Milner, Parrow, and Walker's pi-calculus [MPW92, Mil91], have popularized the idea that a range of concurrent programming idioms can be modeled by simple processes exchanging messages on channels. Besides their immediate applications in specification and verification of concurrent systems, these calculi 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, KY94, KY95]), and more recently as core programming languages in their own right [PT95a, PT95b, FG96].

The small set of message-passing primitives used by the pi-calculus makes it attractive both as an object of theoretical study and as a basis for language design and implementation. However, such generality has a cost: the pi-calculus is an extremely low-level notation, and encoding higher-level features (such as functional computation) in pi-calculus can throw away potentially useful information. A common response to this observation is to consider larger calculi with, for example, 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 avoid adding new primitives and instead recover information about special "modes of usage" from static type information.

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee.
POPL '96, St. Petersburg FLA USA
©1996 ACM 0-89791-769-3/95/01

1.1 Encoding functional computation

The following example illustrates how one can encode functions and function application (cf. [Mil90, San93]) in a pure message-passing calculus, and also highlights some of the pitfalls of such encodings.

Suppose that *plusone* is a communication channel and that there is some running process that repeatedly reads pairs $[i, r]$ from *plusone*, with i a number and r a channel, and responds in each case by sending the value $i + 1$ on r. (Numbers can themselves be implemented using processes and channels, but that need not concern us here.) 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 are given 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, returning the final result, $j + 2$, along the channel s:

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

The replicated input expression $\mathit{plustwo}?^*[j, s]. \ldots$ 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) creates 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 $\mathit{plusone}![j, r_1]$) and, in parallel, waits to receive a response k on the "continuation channel" 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 continuation channel s to the original caller.

In this example, 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. Unfortunately, there is nothing in the pi-calculus' 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.

1.2 Arities and polarities

The simplest type systems for the pi-calculus [Mil91] just track the arities of channels, in order 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 in each message sent along x, as well as their types. For example, the channel *plustwo* would be given the 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, LW95b]. 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 *polarity* (or directionality) of communications on channels like *plustwo* by introducing two new channel types: $![T_1,\ldots,T_n]$ for channels that (in a given context) can only be used to send tuples of type $[T_1,\ldots,T_n]$, and $?[T_1,\ldots,T_n]$ for channels that can only be used to receive $[T_1,\ldots,T_n]$-tuples [PS93, Ode95]. From the point of view of the process implementing the increment function, the type of *plusone* now becomes $?[Num,![Num]]$ — 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, *plusone* has type $![Num,![Num]]$: 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 draft 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, because of the possibility that the external environment might use channels generated by the encoding in the wrong direction [San92]. Pierce and Sangiorgi [PS93] showed how the validity of beta-reduction 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 polarized channel types. For example, we cannot express the fact that the result channel that is passed to *plusone* is used just once. Without this information, we are not justified in replacing our *plustwo* example with a more efficient "tail-call optimized" form:

$$plustwo?^*[j,s].\\ (\nu\ r)\,(plusone![j,r] \mid r?[k].\,plusone![k,s])$$

To see that this version does not have the same behavior as the original in all contexts, imagine that the increment process actually signals twice on its result channel. The tail-call optimized implementation above makes this fact visible to its own caller, generating two messages on the result channel s, whereas the original implementation sends just one message on s.

1.3 Linear types

We propose here a more refined pi-calculus type system, in which channel types have not only *polarities*, which determine the directions in which they can be used, but also *multiplicities*, which control how many times they can be used. For example, the type $!^1[Num]$ describes a channel that can be used exactly once to send a message containing a number, while $!^\omega[Num]$ describes a channel that can be used any number of times. 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 $!^\omega[Num,!^1[Num]]$, 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 tail-call optimized versions of our *plustwo* example are provably equivalent in the sense that they behave the same in all well-typed contexts, as we show in Section 6.2. Actually, the refined typings of *plusone* and *plustwo* are instances of a general observation: using techniques similar to those in [Tur95, PS93], it is easy to show that *every* result channel used in the pi-calculus encoding of a lambda-term can be assigned a linear type.

Not only are some useful equivalences gained in the presence of linear types; even the mechanics of reasoning about processes can be augmented, since (in the absence of non-local operators like choice) two processes communicating over a linear channel can neither interfere with nor be affected by other communicating processes. 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 cannot be reduced further so as to reach a common point. 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 4.4.

Linear typing potentially enables a variety of compiler optimizations. The impact of these optimizations in practice is not yet clear, but work is underway on evaluating several in a compiler for a linear variant of Pict, a programming language based on the pi-calculus [PT95a, PT95b]. For example, the implementation of communication on a linear channel can be simplified from the current Pict implementation, since a linear channel 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. In addition, 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.

Linear channel types can also help detect common programming mistakes. For example, it is often the case that a process must signal on some channel to indicate that it is leaving a "critical section." Suppose c is the channel upon on which we are supposed to signal completion. If the critical section is complex, it is easy to forget to signal completion on c, or to signal twice on c. However, if c is given a linear type, our typing rules will ensure that c is used exactly once.

Note that we must be slightly careful about what "exactly once" means here: typing is preserved by reduction (cf. Section 4.3), so a process containing exactly one use of c must either communicate on c and evolve to a process that cannot use c, or else make some other step and become a process that still contains one use of c, or else do nothing at all. This guarantees that c will never be used twice, but not that c will eventually be used, since the subprocess in which c occurs might diverge or deadlock before it actually communicates on c. For example, the process $(\nu z : \updownarrow^{\omega}[])\, z?[].\, c![]$ creates a fresh channel z, reads from it, and then sends on c. Since z is fresh, there can never be any sender on z, and the send on c 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.

1.4 Overview

Our goals in this paper are (1) a precise definition of a pi-calculus with linear types and (2) formal justifications of the claims sketched here about typing and program equivalences. The 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, more importantly, 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 extends the results on strong barbed congruence to the more useful weak case, which is used to prove the two versions of *plustwo* equivalent. Section 7 discusses related work and Section 8 sketches several variations on the system studied here. Proofs omitted in this summary can be found in an accompanying technical report [KPT95].

2 Notational Preliminaries

The operators introduced in the examples above constitute a fragment of the polyadic pi-calculus [Mil91], omitting the choice and matching operators (as in Milner's mini pi-calculus [Mil90]) and using only asynchronous communication (as in Honda and Tokoro's ν-calculus [HT91]). We have argued elsewhere for the pragmatic virtues of this calculus [PT95a], but many of the techniques we develop should be applicable to any similar system. (Of course, the behavioral properties studied in later sections are sensitive to the exact choice of calculus; in particular, partial confluence of linear communications fails in the presence of a general choice operator.) We also provide booleans and conditional expressions as primitives, since their typing behavior interacts with linearity in a slightly special way.

2.1 Syntax

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 $\updownarrow = \{i,o\}$, $! = \{o\}$, $? = \{i\}$, and $| = \{\}$. The set of types is:

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

For example, the type $!^1[Bool]$ (i.e. $\{o\}^1[Bool]$) describes a channel that can be used exactly once to send a message containing a boolean, while $\updownarrow^1[Bool]$ describes a channel that can be used once for input and once for output. For brevity, sequences of types, etc. are written $\tilde{T}$ instead of $T_1, \ldots, T_n$. A type of the form $p^1[\tilde{T}]$, where $p \neq |$, is called *limited*. All other types are *unlimited*. An example of an unlimited type is $!^{\omega}[Bool, !^1[Bool]]$, which describes a channel upon which we can send any number of messages, but where the components of each message must be a boolean and a linear output channel.

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
		$(\nu x : T)\, P$	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 inputs 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 $0 = (\nu x : \updownarrow^{\omega})\, x![]$.

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 $x?^*[\tilde{y}].\, P$.

2.2 Type environments

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 $\Gamma = \Gamma_1, x : T$ for some Γ_1. 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 the free variables of the process. 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.

2.3 Operations on types

The *combination* of two types T_1 and T_2, written $T_1 + T_2$, is defined as follows (in all cases other than those mentioned below, $T_1 + T_2$ is undefined):

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

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

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

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 $T - T$ for the *remnant* of T. The set of *remnants* of a type environment Δ, written $\mathit{Rems}(\Delta)$, comprises 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 $x_1 : T_1 - T_1, \ldots, x_n : T_n - T_n$.

3 Typing

Our linear channel types are inspired by Girard's linear logic [Gir87, GLT89], a "resource conscious" refinement of classical logic, and by numerous proposals for functional languages based on linear logic (e.g. [Abr93, Wad91, Bak92, Mac94, TWM95]). A crucial element in 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 in a linear lambda-calculus, we do not simply check that v and w are well typed in Γ:

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

Instead, we split the type environment in the conclusion into two parts, representing the resources consumed by 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. (We only consider instances of this rule scheme where $\Gamma_1 + \Gamma_2$ is well defined.) 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 just once."

Like some linear type systems, we omit the rule of *dereliction* from linear logic, which allows an unlimited channel to be coerced to a linear one. If our type system had a subtyping relation, we would similarly ban the coercion $p^{\omega}[\tilde{T}] \leq p^{1}[\tilde{T}]$. The worlds of linear and unlimited values are never allowed to mix.

Our typing rules differ in one other technical detail from the usual formulations 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 $x : !^{1}[\,],\ y : !^{1}[\,] \vdash x![\,] \,|\, y![\,]$, we check $x : !^{1}[\,] \vdash x![\,]$ and $y : !^{1}[\,] \vdash y![\,]$. 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 $x : !^{1}[\,],\ y : !^{1}[\,]$, we check $x![\,]$ in $x : !^{1}[\,],\ y : |^{1}[\,]$ and $y![\,]$ in $x : |^{1}[\,],\ y : !^{1}[\,]$.

To ensure that all the linear bindings in the environment get used somewhere, we check at all the leaves of the typing derivation — i.e., at all instances of the typing rule for output expressions — that all the remaining bindings in the current environment have either multiplicity ω or polarity $|$:

$$\frac{\Gamma \ \text{ unlimited}}{\Gamma + x : !^{m}[\tilde{T}] + \tilde{y}{:}\tilde{T} \vdash x![\tilde{y}]} \qquad \text{(E-Out)}$$

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 consumed by the output, since the arguments $\tilde{y}$ 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.

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

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.

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

The rule for ν-expressions adds the given type for x to the type environment. The side-condition verifies that the supplied type is a reasonable one for a channel declaration (i.e. not *Bool* and not a type that would allow only input or only output, which would be silly).

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

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.

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

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 noting 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 righthand premise. There is some cost associated with checking the "Γ unlimited" side conditions, but a few simple tricks keep this small in practice.

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.

4.1 Structural congruence

Structural congruence plays an important technical role in simplifying the definition of the reduction relation. For example, we intend that both the process $x![] \mid x?[].\, P$ and the process $x?[].\, P \mid x![]$ reduce to P. Since these two will be 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. For experts, our structural congruence relation is identical to the usual one, except that we drop some "garbage collection" rules.

The first two structural congruence rules state that $|$ 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:

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

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 satisfied 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 channel bindings to be switched:

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

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.

4.1.1 Lemma: If $P \equiv Q$, then $\Gamma \vdash P$ iff $\Gamma \vdash Q$.

4.1.2 Definition: A process $(\nu \tilde{x}{:}\tilde{T})\, (P_1 \mid \cdots \mid P_n)$ is in *normal form* if $P_1, \ldots, P_n$ are guarded processes, where a process is *guarded* if it is an input, output, replicated input, or conditional.

4.1.3 Lemma: For any process P there is some Q in normal form such that $P \equiv Q$.

Proof: By the associativity and commutativity of $|$ and the fact that any $(\nu \tilde{x}{:}\tilde{T})$ not guarded by an input can be moved to the outside of the expression using S-Extr. □

4.1.4 Lemma: If the processes $(\nu \tilde{x}{:}\tilde{T})\, (P_1 \mid \cdots \mid P_n)$ and $(\nu \tilde{y}{:}\tilde{U})\, (Q_1 \mid \cdots \mid Q_m)$ are normal forms that are both structurally congruent to P, then $\tilde{x}{:}\tilde{T}$ is a permutation of $\tilde{y}{:}\tilde{U}$, $m = n$, and, for some permutation $Q'_1, \ldots, Q'_n$ of $Q_1, \ldots, Q_n$, $P_i \equiv Q'_i$ for $1 \leq i \leq n$.

Proof: Structural congruence neither introduces nor eliminates guarded subexpressions or ν-bindings, and does not move subexpressions across input prefixes. □

4.2 Reduction

The pi-calculus reduction relation is usually 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 need later for stating properties of the semantics. We write $\Gamma \vdash P \xrightarrow{m,\alpha} Q$, where Γ is 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), m is a multiplicity tag (recording whether the communication that leads from P to Q occurs on a linear or an unlimited channel), and α can be either the name of a channel, indicating where the communication occurs, or else the special tag τ, indicating that the channel on which the communication occurs is bound by a ν that is not in scope at this point.

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.

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

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, ready to consume further outputs on x.

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

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:

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

Similarly, if P reduces to Q, then it can still do so when placed under a ν. However, because of our extra annotations, 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.

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

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

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

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

$$\Gamma \vdash \text{if } \mathit{true} \text{ then } P \text{ else } Q \xrightarrow{1,\tau} P \qquad \text{(R-IfT)}$$

$$\Gamma \vdash \text{if } \mathit{false} \text{ then } P \text{ else } Q \xrightarrow{1,\tau} Q \qquad \text{(R-IfF)}$$

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.

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

When we want to be explicit about the typing environment in force after a reduction has occurred, we write $\Gamma \vdash P \xrightarrow{m,\alpha} P' \dashv \Gamma'$, where

$$\Gamma' = \begin{cases} \Gamma & \text{if } \alpha = \tau \\ \Gamma - x : \Gamma(x) & \text{if } \alpha = x. \end{cases}$$

That is, if the label on the arrow is τ, then the binder of the channel x on which the communication occurred is within P and there is nothing to do; on the other hand, if the label is x, then we must remove an instance of the type of x from the capabilities in Γ.

As usual, we write $\Gamma \vdash P \longrightarrow^* Q$ or $\Gamma \vdash P \longrightarrow^* Q \dashv \Gamma'$ to show that P reduces to Q in zero or more steps.

4.3 Type soundness

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

4.3.1 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 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).

We use the shorthand $\{\mid Q\}$ to stand for either nothing or a parallel composition with a process Q.

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

1. If Q is an input $x![y_1, \ldots, y_m] \mid x?[z_1, \ldots, z_n].\, Q_1$ or a replicated input $x![y_1, \ldots, y_m] \mid x?^*[z_1, \ldots, z_n].\, Q_1$, then $m = n$ and the binding of x (in either P or Γ) has polarity $\updownarrow$.
2. If $Q = x![\tilde{y}]$, then the binding of x has a polarity including !.
3. If $Q = x?[\tilde{y}].\, Q_1$, then the binding of x has a polarity including ?.
4. If $Q = x![\tilde{y}] \mid x![\tilde{z}]$, then the binding of x has multiplicity ω.
5. If $Q = (x?[\tilde{y}].\, Q_1) \mid (x?[\tilde{z}].\, Q_2)$, then the binding of x has multiplicity ω.
6. 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.

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

4.4 Partial confluence

We now come to the first results that show how linearity affects reasoning about program behavior. The first theorem below says that communication on a linear channel is deterministic. The second says: If P is a process that contains two possible communications, one 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.

4.4.1 Theorem: If $\Gamma \vdash P \xrightarrow{1,x} Q$ and $\Gamma \vdash P \xrightarrow{1,x} R$, then $Q \equiv R$.

4.4.2 Theorem [Partial confluence]: If $\Gamma \vdash P$ and $\Gamma \vdash P \xrightarrow{1,\alpha} P' \dashv \Gamma'$, then, for any Q, m, and β such that $\Gamma \vdash P \xrightarrow{m,\beta} Q \dashv \Delta$ and $P' \not\equiv Q$, there is some R such that $\Delta \vdash Q \xrightarrow{1,\alpha} R$ and $\Gamma' \vdash P' \xrightarrow{m,\beta} R$.

The proofs of 4.4.1 and 4.4.2, which occupy the rest of this section, may safely be skipped on a first reading. We begin with one technical lemma.

4.4.3 Lemma: If $\Gamma \vdash P \xrightarrow{m,\alpha} P'$, then there are Q, Q' such that one of the following holds,

1. $Q = (\nu\,\tilde{w}{:}\tilde{W})\,(x![\tilde{z}] \mid x?[\tilde{y}].\,Q_1\{\mid R\})$ and
   $Q' = (\nu\,\tilde{w}:\tilde{W'})\,([\tilde{z}/\tilde{y}]Q_1\{\mid R\})$,
   where, if $w_i = x$, then $W_i' = W_i - W_i$, and otherwise $W_i' = W_i$;
2. $Q = (\nu\,\tilde{w}{:}\tilde{W})\,(x![\tilde{z}] \mid x?^*[\tilde{y}].\,Q_1\{\mid R\})$ and
   $Q' = (\nu\,\tilde{w}{:}\tilde{W})\,([\tilde{z}/\tilde{y}]Q_1 \mid x?^*[\tilde{y}].\,Q_1\{\mid R\})$;
3. $Q = (\nu\,\tilde{w}{:}\tilde{W})\,(\text{if } \mathit{true} \text{ then } Q_1 \text{ else } Q_2\{\mid R\})$ and
   $Q' = (\nu\,\tilde{w}{:}\tilde{W})\,(Q_1\{\mid R\})$; or
4. $Q = (\nu\,\tilde{w}{:}\tilde{W})\,(\text{if } \mathit{false} \text{ then } Q_1 \text{ else } Q_2\{\mid R\})$ and
   $Q' = (\nu\,\tilde{w}{:}\tilde{W})\,(Q_2\{\mid R\})$;

and such that $P \equiv Q$, $P' \equiv Q'$, and $\Gamma \vdash Q \xrightarrow{m,\tau} Q'$ is derivable without using R-Cong.

Proof: From the fact that R-Cong and R-New2 can be permuted after the other rules and successive applications of R-Cong can be combined into one R-Cong. □

Proof of Theorem 4.4.1: By the reduction rules, $\Gamma(x) = \updownarrow^1[\tilde{T}]$. From $\Gamma \vdash P \xrightarrow{1,x} Q$ and Lemma 4.4.3,

$$P \equiv (\nu\,\tilde{w}{:}\tilde{W})\,(x![\tilde{y}] \mid x?[\tilde{z}].\,P_1\{\mid P_2\})$$

and

$$Q \equiv (\nu\,\tilde{w}{:}\tilde{W})\,([\tilde{y}/\tilde{z}]P_1\{\mid P_2\}).$$

Also, from $\Gamma \vdash P \xrightarrow{1,x} R$ and Lemma 4.4.3,

$$P \equiv (\nu\,\tilde{w}'{:}\tilde{W'})\,(x![\tilde{y'}] \mid x?[\tilde{z'}].\,P_1'\{\mid P_2'\})$$

and

$$R \equiv (\nu\,\tilde{w}'{:}\tilde{W'})\,([\tilde{y'}/\tilde{z'}]P_1'\{\mid P_2'\}).$$

We can assume, without loss of generality, that

$$\begin{array}{ll} & (\nu\,\tilde{w}{:}\tilde{W})\,(x![\tilde{y}] \mid x?[\tilde{z}].\,P_1\{\mid P_2\}) \\ \text{and} & (\nu\,\tilde{w}'{:}\tilde{W'})\,(x![\tilde{y'}] \mid x?[\tilde{z'}].\,P_1'\{\mid P_2'\}) \end{array}$$

are normal forms, because otherwise we can obtain normal forms of P_2, P_2' and then move the ν to the outside. By Lemma 4.1.4, $\tilde{w}{:}\tilde{W}$ is a permutation of $\tilde{w}'{:}\tilde{W'}$. Moreover, let $P_2' = R_1 \mid \cdots \mid R_n$ for guarded processes $R_1, \ldots, R_n$. Then either $x?[\tilde{z}].\,P_1 \equiv x?[\tilde{z}].\,P_1'$ or $x?[\tilde{z}].\,P_1 \equiv R_i$ for some i. The latter case cannot happen; R_i must be in the form $x?[\tilde{u}].\,R_i'$, but this contradicts the fact that

$$\Gamma \vdash (\nu\,\tilde{w}'{:}\tilde{W'})\,(x![\tilde{y'}] \mid x?[\tilde{z'}].\,P_1' \mid P_2')$$

and $\Gamma(x) = \updownarrow^1[\tilde{T}]$. Therefore, we obtain $x?[\tilde{z}].\,P_1 \equiv x?[\tilde{z'}].\,P_1'$. Similarly, we have $x![\tilde{y}] \equiv x![\tilde{y'}]$, which implies

$$\begin{array}{rcl} Q & \equiv & (\nu\,\tilde{w}{:}\tilde{W})\,([\tilde{y}/\tilde{z}]P_1\{\mid P_2\}) \\ & \equiv & (\nu\,\tilde{w}'{:}\tilde{W'})\,([\tilde{y'}/\tilde{z'}]P_1'\{\mid P_2'\}) \\ & \equiv & R, \end{array}$$

as required. □

Proof of Theorem 4.4.2: Suppose that $\xrightarrow{1,\alpha}$ and $\xrightarrow{m,\beta}$ both come from communications with unreplicated inputs. (The cases for replicated input and conditional are similar.) From the assumption and Lemma 4.4.3,

$$\begin{array}{l} P \equiv (\nu\,\tilde{w}{:}\tilde{W})\,(x![\tilde{y}] \mid x?[\tilde{z}].\,P_1 \mid P_2) \\ P' \equiv (\nu\,\tilde{w}{:}\tilde{W'})\,([\tilde{y}/\tilde{z}]P_1 \mid P_2) \\ P \equiv (\nu\,\tilde{u}{:}\tilde{U})\,(v![\tilde{y'}] \mid v?[\tilde{z'}].\,P_3 \mid P_4) \\ Q \equiv (\nu\,\tilde{u}{:}\tilde{U'})\,([\tilde{y'}/\tilde{z'}]P_3 \mid P_4) \end{array}$$

where $W_i' = W_i - W_i$ if $w_i = x$, and otherwise $W_i' = W_i$, and where the binding of x in Γ or $\tilde{w}{:}\tilde{W}$ is $\updownarrow^1[\tilde{T}]$. Also, $U_i' = U_i - U_i$ if $u = v$, otherwise $U_i' = U_i$. We can assume, without loss of generality, both
$(\nu\,\tilde{w}{:}\tilde{W})\,(x![\tilde{y}] \mid x?[\tilde{z}].\,P_1 \mid P_2)$ and
$(\nu\,\tilde{u}{:}\tilde{U})\,(v![\tilde{y'}] \mid v?[\tilde{z'}].\,P_3 \mid P_4)$ are in normal forms. From the assumption $P' \not\equiv Q$, we have $x \neq v$, because otherwise by Lemma 4.1.4 and the linearity of x, we have $x![\tilde{y}] \equiv v![\tilde{y'}], x?[\tilde{z}].\,P_1 \equiv x?[\tilde{z'}].\,P_3, P_2 \equiv P_4$, from which we obtain $P' \equiv Q$.

By Lemma 4.1.4,

$$\begin{array}{l} P_2 \equiv v![\tilde{y'}] \mid v?[\tilde{z'}].\,P_3' \mid R_1 \mid \cdots \mid R_n \\ P_4 \equiv x![\tilde{y}] \mid x?[\tilde{z}].\,P_1' \mid R_1' \mid \cdots \mid R_n' \\ P_1 \equiv P_1', P_3 \equiv P_3', R_i \equiv R_i' (1 \leq i \leq n) \end{array}$$

and $\tilde{w}{:}\tilde{W}$ is a permutation of $\tilde{u}{:}\tilde{U}$. Let

$$R = (\nu\,\tilde{w}{:}\tilde{W''})\,([\tilde{y}/\tilde{z}]P_1 \mid [\tilde{y'}/\tilde{z'}]P_3 \mid R_1 \mid \cdots \mid R_n),$$

where $W_i'' = W_i - W_i$ if $w_i = x$ or $w_i = v$, and otherwise $W_i'' = W_i$. Then, $\Delta \vdash Q \xrightarrow{1,\alpha} R$, and $\Gamma' \vdash P' \xrightarrow{m,\beta} R$. □

5 Strong Bisimilarity

Our final task 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 bisimulation* [MS92, San92], which is 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 facilitates comparison and helps give confidence that the definition is natural. And finally, like all varieties of bisimilarity,

it comes with a coinductive proof technique. We begin by establishing the basic theory of strong bisimilarity in this section; Section 6 extends the results developed here to the more useful case of weak bisimilarity.

5.1 Basic definitions

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

5.1.1 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., if 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 one of the following holds

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

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

Experts should note that this definition of barbs agrees with the standard one for the full pi-calculus. Since we are working with an asynchronous fragment here, with output atoms instead of output prefixes, we do not actually need to allow observation of inputs: we could drop clauses 2 and 3 of the definition and obtain a notion of observation closer to "what processes can see about each other." We conjecture that the theory developed in the rest of the paper would not be affected by this change.

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 some bisimulation. Recall that $Rems(\Delta)$ is the set of environments obtained from Δ by changing some bindings in Δ of the form $x : p^1[\tilde{T}]$ to $x : |^1[\tilde{T}]$.

5.1.2 Definition [Strong barbed bisimulation]: A relation

$$\begin{aligned}\mathcal{R} \subseteq \{\ &((\Delta_P,\, P),\ (\Delta_Q,\, Q))\\ &\mid\ \Delta_P, \Delta_Q \in Rems(\Delta)\\ &\wedge\ \Delta_P \vdash P\\ &\wedge\ \Delta_Q \vdash Q\ \}\end{aligned}$$

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 $P \overset{\bullet}{\sim}_\Delta Q$, if $((\Delta_P, P), (\Delta_Q, Q)) \in \mathcal{R}$ for some $\Delta_P, \Delta_Q \in Rems(\Delta)$ 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. The resulting relation, called barbed congruence, will later be seen to be a full congruence for all of the other process constructors.

5.1.3 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 as $(\Sigma_{x \in dom(\Delta) \wedge \sigma(x) = y} \Delta(x)) + U$ 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 $\tilde{x}$.

5.2 Congruence

5.2.1 Definition: P and Q are Δ-*precongruent*, written $P \sim_\Delta^p Q$, if, for any Δ-Γ substitution σ and $\Theta \vdash R$ such that $\Theta + \Gamma$ is well-defined, $R \mid \sigma P \overset{\bullet}{\sim}_{\Theta+\Gamma} R \mid \sigma Q$. That is, $\sim_\Delta^p$ 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 $P \sim_\Delta^p Q$. 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 the processes $x![] \mid y![] \mid x?[].y?[].0$ and $x![] \mid y![] \mid y?[].x?[].0$ 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 process context again yields congruent processes.

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

6 Weak Bisimilarity

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 each to take zero or many steps to match a single step of the other. The main result in this section is Example 6.2.3, which formalizes the claim in the introduction that the two versions of the plus-two process are behaviorally equivalent.

6.1 Definitions

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 ($P \dot{\approx}_\Delta Q$) is obtained by replacing the second $\longrightarrow$ in clauses (2) and (3) of Definition 5.1.1 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. The refined definitions read as follows:

6.1.1 Definition [Weak 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 weak Δ-*bisimulation* if, whenever $((\Delta_P, P), (\Delta_Q, Q)) \in \mathcal{R}$,

1. $\Delta_P \vdash P \longrightarrow^* P' \Downarrow_a^\Delta$ iff $\Delta_Q \vdash Q \longrightarrow^* Q' \Downarrow_a^\Delta$;
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 *weakly* Δ-*bisimilar*, written $P \dot{\approx}_\Delta Q$, if the pair $((\Delta_P, P), (\Delta_Q, Q)) \in \mathcal{R}$ for some $\Delta_P, \Delta_Q \in \mathit{Rems}(\Delta)$ and weak Δ-bisimulation $\mathcal{R}$.

6.1.2 Definition: Two Δ-remnant processes P and Q are *weakly* Δ-*precongruent*, written $P \approx^p_\Delta Q$, if for any Δ-Δ' substitution σ and $\Gamma \vdash R$ such that $\Gamma + \Delta'$ is well-defined, $R \mid \sigma P \dot{\approx}_{\Gamma+\Delta'} R \mid \sigma Q$. Two Δ-processes P and Q are *weakly* Δ-*congruent*, written $P \approx_\Delta Q$, if $P \approx^p_\Delta Q$.

In the arguments that follow, it is convenient to use "weak bisimulation up to $\equiv$" to prove that two processes are bisimilar, instead of using $\dot{\approx}_\Delta$ directly.

6.1.3 Definition [Weak Bisimulation up to $\equiv$]:
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 weak Δ-*bisimulation up to* $\equiv$ if,
whenever $((\Delta_P, P), (\Delta_Q, Q)) \in \mathcal{R}$,

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

Two Δ-remnant processes P and Q are weak Δ-*bisimilar up to* $\equiv$, if $\Delta_P, \Delta_Q \in \mathit{Rems}(\Delta)$ and $((\Delta_P, P), (\Delta_Q, Q)) \in \mathcal{R}$ where $\mathcal{R}$ is a weak Δ-bisimulation up to $\equiv$.

6.1.4 Lemma: P and Q are weakly Δ-bisimilar iff they are weakly Δ-bisimilar up to $\equiv$.

When we prove that the given two processes are weakly barbed bisimilar, it is often hard to compare weak barbs since we need to consider all the possible reduction sequences. The following theorem allows us to compare strong barbs instead of weak barbs, which substantially simplifies proofs of weak barbed bisimilarity. Let $\mathit{Barbs}_\Delta(P)$ be the set of strong barbs of P, and $\mathit{WBarbs}_\Delta(P)$ the set of weak barbs of P.

6.1.5 Theorem: 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 weak Δ-*bisimulation* if, whenever $((\Delta_P, P), (\Delta_Q, Q)) \in \mathcal{R}$,

1. $\mathit{Barbs}_\Delta(P) \subseteq \mathit{WBarbs}_\Delta(Q)$ and $\mathit{Barbs}_\Delta(Q) \subseteq \mathit{WBarbs}_\Delta(P)$
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'.

Proof: It suffices to show that if $((\Delta_P, P), (\Delta_Q, Q)) \in \mathcal{R}$, then $\mathit{WBarbs}_\Delta(P) = \mathit{WBarbs}_\Delta(Q)$. Suppose that $a \in \mathit{WBarbs}_\Delta(P)$. Then there is some P' such that $\Delta_P \vdash P \longrightarrow^* P' \dashv \Delta'_P$ and $a \in \mathit{Barbs}_\Delta(P')$. By condition 2, there is a matching reduction sequence $\Delta_Q \vdash Q \longrightarrow^* Q' \dashv \Delta'_Q$ with $((\Delta'_P, P'), (\Delta'_Q, Q')) \in \mathcal{R}$. By condition 1, $a \in \mathit{WBarbs}_\Delta(Q')$. Since $\mathit{WBarbs}_\Delta(Q') \subseteq \mathit{WBarbs}_\Delta(Q)$, we know that a is in $\mathit{WBarbs}_\Delta(Q)$, and so $\mathit{WBarbs}_\Delta(P) \subseteq \mathit{WBarbs}_\Delta(Q)$. Similarly, $\mathit{WBarbs}_\Delta(P) \supseteq \mathit{WBarbs}_\Delta(Q)$. □

6.1.6 Theorem: $\approx_\Delta$ is a congruence.

Now the partial confluence of linear reduction (Theorem 4.4.2) can be expressed in a more useful way:

6.1.7 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 precongruence class of a process, so that, to show that $P \approx_\Delta Q$, it suffices to show that P and Q are both well typed in Δ and $P' \approx^p_\Delta Q$ for some linear reduct P' of P. (In the case of a linear τ-reduction, we just need to show that P' and Q are congruent.) We use the next lemma to prove the theorem.

6.1.8 Lemma: If $\Gamma \vdash P$, $\Gamma \vdash P \xrightarrow{1,\alpha} P'$, $\Gamma \in Rems(\Delta)$, and $P \Downarrow_a^\Delta$, then $P' \Downarrow_a^\Delta$.

Proof of Theorem 6.1.7: Suppose that σ is a Δ-Δ' substitution and $\Gamma \vdash R$, with $\Gamma + \Delta'$ well-defined. Then $\Gamma + \Delta' \vdash R \,|\, \sigma P \xrightarrow{1,(\sigma,\tau/\tau)\alpha} R \,|\, \sigma Q$. Therefore, we must show that if $\Delta \vdash P \xrightarrow{1,\alpha} Q$, then $P \dot{\approx}_\Delta Q$. Let

$$\begin{array}{rcl}\mathcal{R} & = & \{((\Delta', Q'), (\Delta', Q'')) \\ & & \mid \Delta \vdash Q \longrightarrow^* Q' \dashv \Delta' \ \wedge \ Q' \equiv Q''\} \\ & \cup & \{((\Delta', P'), (\Delta'', Q')) \\ & & \mid \Delta \vdash P \longrightarrow^* P' \dashv \Delta' \\ & & \wedge \Delta' \vdash P' \xrightarrow{1,\alpha} Q' \dashv \Delta''\}\end{array}$$

We show that $\mathcal{R}$ is a weak Δ-bisimulation. Since the first set in the definition of $\mathcal{R}$ is itself a weak Δ-bisimulation, it suffices to check only the case $((\Delta'_P, P'), (\Delta'_Q, Q')) \in \{((\Delta', P'), (\Delta'', Q')) \mid \Delta \vdash P \longrightarrow^* P' \dashv \Delta'$ and $\Delta' \vdash P' \xrightarrow{1,\alpha} Q' \dashv \Delta''\}$.

1. We must show that $Barbs_\Delta(P') \subseteq WBarbs_\Delta(Q')$ and $Barbs_\Delta(Q') \subseteq WBarbs_\Delta(P')$. Suppose that $Q' \Downarrow_a^\Delta$. Then, $\Delta' \vdash P' \xrightarrow{1,\alpha} Q' \Downarrow_a^\Delta$. Conversely, if $P' \Downarrow_a^\Delta$, then $Q' \Downarrow_a^\Delta$ by Lemma 6.1.8 and $\Delta' \vdash P' \xrightarrow{1,\alpha} Q'$.

2. Consider any reduction $\Delta'_P \vdash P' \xrightarrow{m,\beta} P''$. If $P'' \equiv Q'$, then it is matched by $\Delta'_Q \vdash Q' \longrightarrow^* Q'$. Otherwise, by Theorem 4.4.2, there is some Q'' such that $\Delta''_P \vdash P'' \xrightarrow{1,\alpha} Q'' \dashv \Delta''_Q$ and $\Delta'_Q \vdash Q' \xrightarrow{m,\beta} Q''$. Therefore, the reduction is matched by $\Delta'_Q \vdash Q' \xrightarrow{m,\beta} Q''$ and $((\Delta''_P, P''), (\Delta''_Q, Q'')) \in \mathcal{R}$.

3. If $\Delta'_Q \vdash Q' \longrightarrow Q'' \dashv \Delta''_Q$, then $\Delta'_P \vdash P' \xrightarrow{1,\alpha} Q' \longrightarrow Q'' \dashv \Delta''_Q$ and $((\Delta''_Q, Q''), (\Delta''_Q, Q''))$ appears in $\mathcal{R}$. $\Box$

6.2 Application

We now apply the foregoing theory to show that the two versions of the plus-two process presented in the introduction are weakly congruent. These processes are nontrivial, and the proof of their equivalence requires some care; we begin by introducing two technical lemmas to help structure the argument.

6.2.1 Lemma: If $P \dot{\approx}_\Delta Q$ and $\Delta \in Rems(\Gamma)$, then $P \dot{\approx}_\Gamma Q$.

We write $\Gamma +\!\!+ x : T$ to mean either $\Gamma, x : T$ if $x \notin dom(\Gamma)$ or $\Gamma + x : T$ if $x \in dom(\Gamma)$.

6.2.2 Lemma: If $\Delta, y : !^1[\tilde{T}] \vdash P$ and $\Delta +\!\!+ x : !^1[\tilde{T}]$ is well defined, then

$$[x/y]P \dot{\approx}_{\Delta +\!\!+ x:!^1[\tilde{T}]} (\nu y : \updownarrow^1[\tilde{T}])\,(P \,|\, y?[\tilde{z}].\, x![\tilde{z}]).$$

6.2.3 Example: Let

$$\begin{array}{rcl}\Gamma & = & plustwo : ?^\omega[Num, !^1[Num]], \\ & & plusone : !^\omega[Num, !^1[Num]].\end{array}$$

Then the two versions of the plus-two process given in the introduction are weakly congruent.

Proof: Let

$$\Delta = plusone : !^\omega[Num, !^1[Num]], s : !^1[Num], k : Num.$$

By Theorem 6.1.6, it suffices to show that

$$(\nu r : \updownarrow^1[Num])\,(plusone![k, r] \,|\, r?[l].\, s![l]) \approx_\Delta plusone![k, s].$$

Because both side of processes are well-typed under Δ, it suffices to show that

$$(\nu r : \updownarrow^1[Num])\,(plusone![k, r] \,|\, r?[l].\, s![l]) \approx^p_\Delta plusone![k, s].$$

Moreover, because substitution can only rename variables (note that all bindings in Δ are incompatible with each other), it suffices to show that if $\Theta \vdash Q$ and $\Theta + \Delta$ is well defined, then

$$\begin{array}{l}Q \,|\, (\nu r : \updownarrow^1[Num])\,(plusone![k, r] \,|\, r?[l].\, s![l]) \\ \dot{\approx}_{\Theta+\Delta}\ Q \,|\, plusone![k, s].\end{array}$$

We show that

$$\begin{array}{rcl}\mathcal{R} & = & \{((\Theta' + \Delta, \\ & & \quad Q' \,|\, (\nu r : \updownarrow^1[Num])\,(plusone![k, r] \,|\, r?[l].\, s![l])), \\ & & \quad (\Theta' + \Delta, Q' \,|\, plusone![k, s])) \\ & & \mid \Theta \vdash Q \longrightarrow^* Q' \dashv \Theta'\} \\ & \cup & \dot{\approx}_{\Theta+\Delta}\end{array}$$

is a weak $(\Theta + \Delta)$-bisimulation up to $\equiv$, from which the desired result follows, since

$$\begin{array}{l}((\Theta + \Delta, Q \,|\, (\nu r : \updownarrow^1[Num])\,(plusone![k, r] \,|\, r?[l].s![l])), \\ \ (\Theta + \Delta, Q \,|\, plusone![k, s])) \in \mathcal{R}.\end{array}$$

Take the pair

$$\begin{array}{l}((\Theta' + \Delta, Q' \,|\, (\nu r : \updownarrow^1[Num])\,(plusone![k, r] \,|\, r?[l].\, s![l])), \\ \ (\Theta' + \Delta, Q' \,|\, plusone![k, s]))\end{array}$$

from the first part of $\mathcal{R}$. We check the three conditions in Theorem 6.1.5.

1. $$\begin{array}{l}Barbs_{\Theta+\Delta}(Q' \,|\, (\nu r : \updownarrow^1[Num])\,(plusone![k, r] \,|\, r?[l].s![l])) \\ \quad = Barbs_{\Theta+\Delta}(Q') \cup \{plusone\} \\ \quad = Barbs_{\Theta+\Delta}(Q' \,|\, plusone![k, s])\end{array}$$

2. Consider any reduction from

   $$Q' \,|\, (\nu r : \updownarrow^1[Num])\,(plusone![k, r] \,|\, r?[l].\, s![l]))$$

   to R. There are two possibilities: either the reduction occurs inside Q', or an interaction occurs between Q' and $(\nu r : \updownarrow^1[Num])\,(plusone![k, r] \,|\, r?[l].\, s![l])$.

   In the former case,

   $$\begin{array}{l}R \equiv Q'' \,|\, (\nu r : \updownarrow^1[Num])\,(plusone![k, r] \,|\, r?[l].s![l]) \\ \Theta' \vdash Q' \longrightarrow Q''.\end{array}$$

   This reduction is matched by

   $$\Theta' + \Delta \vdash Q' \,|\, plusone![k, s] \longrightarrow Q'' \,|\, plusone![k, s].$$

   In the latter case, reduction occurs on the channel *plusone*:

   $$\begin{array}{rcl}Q' & \equiv & (\nu \tilde{w}{:}\tilde{W})\,(plusone?[n, t].P \,|\, O) \\ R & \equiv & (\nu \tilde{w}{:}\tilde{W})\,(\nu r : \updownarrow^1[Num]) \\ & & \quad ([k/n, r/t]P \,|\, O \,|\, r?[l].\, s![l])\end{array}$$

Then the reduction is matched by

$$\begin{array}{l}\Theta' + \Delta \vdash Q' \mid plusone![k,s] \\ \longrightarrow (\nu\, \tilde{w}{:}\tilde{W})\,([k/n, s/t]P \mid O) \\ = [s/r](\nu\, \tilde{w}{:}\tilde{W})\,([k/n, r/t]P \mid O)\end{array}$$

because

$$\begin{array}{l}(\nu\, \tilde{w}{:}\tilde{W})\,(\nu r : \updownarrow^1[Num])\,([k/n, r/t]P \mid O \mid r?[l].s[l]) \\ \dot{\approx}_{\Theta' + \Delta - plusone}\ (\nu\, \tilde{w}{:}\tilde{W})\,([k/n, s/t]P \mid O)\end{array}$$

by Lemma 6.2.2, which implies

$$\begin{array}{l}(\nu\, \tilde{w}{:}\tilde{W})\,(\nu r : \updownarrow^1[Num])\,([k/n, r/t]P \mid O \mid r?[l].s[l]) \\ \dot{\approx}_{\Theta + \Delta}\ (\nu\, \tilde{w}{:}\tilde{W})\,([k/n, s/t]P \mid O)\end{array}$$

by Lemma 6.2.1.

3. Consider any reduction $\Theta' + \Delta \vdash Q' \mid plusone![k,s] \longrightarrow R$. There are again two cases: either reduction occurs inside Q', or interaction occurs between Q' and $plusone![k,s]$. In the former case, $R \equiv Q'' \mid plusone![k,s]$ and $\Theta' \vdash Q' \longrightarrow Q''$. This is matched by:

$$\begin{array}{rcl}\Theta' + \Delta & \vdash & Q' \mid (\nu r : \updownarrow^1[Num])\,(plusone![k,r] \\ & & \quad \mid r?[l].\, s![l]) \\ & \longrightarrow & Q'' \mid (\nu r : \updownarrow^1[Num])\,(plusone![k,r] \\ & & \quad \mid r?[l].\, s![l]).\end{array}$$

The argument for the latter case is similar to part (2).

These three conditions imply that $\mathcal{R}$ is a weak $(\Theta{+}\Delta)$-bisimulation up to $\equiv$. □

7 Related Work

The 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 are shared between just two processes. The motivating intuitions behind this calculus have the flavor of a type system, suggesting that one might achieve the same effect in a standard pi-calculus by introducing a refined type system instead of altering the syntax. (In fact, earlier work by Honda [Hon93] did adopt a type-theoretic perspective related to ours.)

Our type system shares a good deal with linear type systems for lambda-calculi and related programming languages [Abr93, Wad91, Hod92, Bak92, Mac94, TWM95]. In particular, linear typing has recently been proposed as a framework for *syntactic control of interference* in sequential programming languages (cf. [OTPT95]).

In the world of process calculi, there have been several papers on analyzing channel usage using *effect systems*. Nielson and Nielson [NN95] proposed a method for inferring the maximum usages of channels by applying the effect system for CML [NN94], while Kobayashi, Nakade, and Yonezawa [KNY95] proposed a method for approximating the number of receivers that can ever try to read from a channel simultaneously, which in particular can be used to detect channels whose maximum queue length is 1. Our type-based analysis and these effect-based analyses 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.

Abramsky, Gay, and Nagarajan [AGN94, GN95] describe a typed calculus of synchronous processes. Although their calculus ensures deadlock-freedom, its expressive power is much more limited than ours in the sense that there must be exactly one sender and receiver for each channel, and mobility (passing channels as data along channels) cannot be expressed.

Finally, a type system reminiscent of ours — though not explicitly incorporating the notion of linearity — has been developed by Steffen and Nestmann [SN95] for the purpose of analyzing confluence in pi-calculus processes arising in the semantics of concurrent object-oriented programs. (The same problem has been tackled by Liu and Walker using purely semantic techniques [LW95a].) Our analysis is also related to the one used by Niehren [Nie96] to guarantee "uniform confluence" in a pi-calculus fragment with only replicated inputs.

8 Variants and Extensions

We have focused here on a fragment of the original polyadic π-calculus of [Mil91], omitting output guards (processes of the form $x![\tilde{y}].P$), and the choice operator $+$. Although our type system and reduction semantics can easily be extended to the full π-calculus, some of the results we developed must be treated carefully. For example, in the presence of output guards, Example 6.2.3 is not valid. (However, if we allow output guards only with nonlinear channels, the example does remain valid.) Similarly, if we add choice to our calculus, Theorems 4.4.2 and 6.1.7 do not hold, but we can recover them by allowing choice only on processes guarded by nonlinear communications.

A great variety of other "resource-aware" type systems, tracking different kinds of usage information, can be formulated using similar techniques. Indeed, many of these systems can actually be *encoded* in our simple system of linear types.

One trivial variant of the present system is obtained by replacing linear by *affine* channels, which must be used *at most* once instead of exactly once. Since, as we have seen, it is actually possible to throw away linear channels without using them (by placing them in a deadlocked subprocess), the linear and affine variants of the system have nearly identical properties. One advantage of the affine variant is that it allows garbage collection: if the process $(\nu x : \updownarrow^{\omega}[\tilde{T}])\, x?[\tilde{y}].Q$ is well typed in some context Γ, then 0 is well typed in the same context.

A more interesting variant introduces a type of *replicated channels*, which can be used arbitrarily often for output but only allow a single, replicated input. Communication on replicated channels enjoys the same partial confluence property as linear communication. We conjecture that this property can be used to show, for example, that β-reduction preserves weak congruence in the encoding of the call-by-value λ-calculus. To handle replicated channels formally, we introduce a new multiplicity r (either replacing or augmenting

the linear multiplicity 1). We define the operator + for the new multiplicity by

$$p^r[\tilde{T}] + q^r[\tilde{T}] = (p \cup q)^r[\tilde{T}] \quad \text{if } i \notin p \cap q$$

and modify the side conditions of the rules E-In and E-RIn as follows:

$$\frac{\Gamma, \tilde{z}{:}\tilde{T} \vdash P \qquad m \neq r}{\Gamma + x : ?^m[\tilde{T}] \vdash x?[\tilde{z}].\, P} \qquad \text{(E-In)}$$

$$\frac{\Gamma, \tilde{z}{:}\tilde{T} \vdash P \qquad \Gamma \text{ unlimited} \qquad m = r \text{ or } \omega}{\Gamma + x : ?^m[\tilde{T}] \vdash x?^*[\tilde{z}].\, P} \qquad \text{(E-RIn)}$$

Another variation on linear channels is *linearized* channels, which can be used multiple times but only in a sequential manner: an output $x![\tilde{y}].P$ can reuse x again for output in P, while an input $x?[\tilde{y}].P$ can reuse x for input in P. Linearized channels can be encoded using linear channels and recursive types. Write $\updownarrow^*[\tilde{T}]$ for the type of linearized channels repeatedly carrying tuples of type $\tilde{T}$. This type is encoded as follows in terms of linear types:

$$[\![\updownarrow^*[\tilde{T}]]\!] = \updownarrow^1[\tilde{T}, !^1[\,], (\mu X.\, ?^1[\tilde{T}, !^1[\,], X])]$$

That is, a linearized channel is encoded as a linear channel carrying tuples with types $\tilde{T}$ plus two extra parameters: a channel used to trigger the continuation of the output (needed because we are translating from a language with output guards to one without) and another channel of nearly the same type as the original linearized channel (more precisely, of the type of the "input end" of the original linearized channel), which replaces the original in the continuation of both sender and receiver. Input and output expressions on a linearized channel $x \in \updownarrow^*[\tilde{T}]$ are encoded as follows:

$$\begin{array}{rcl} [\![x![\tilde{y}].\, P]\!] & = & (\nu x' : [\![\updownarrow^*[\tilde{T}]]\!])\,(\nu c : \updownarrow^1[\,]) \\ & & \quad x![\tilde{y}, c, x'] \mid c?[\,].\, [\![x'/x]P]\!] \\ [\![x?[\tilde{y}].\, P]\!] & = & x?[\tilde{y}, c, x'].\, (c![\,] \mid [\![x'/x]P]\!]) \end{array}$$

Linearized channel types can be viewed as a variant on a fragment of Honda's "types for dyadic interaction" [Hon93].

Combining the intuitions behind replicated and linearized channel types yields another important usage analysis, which Milner calls "unique handling" [Mil91]: a channel is uniquely handled if it has a single replicated receiver and, at any given moment, at most one sender. That is, the capability to send must be explicitly passed from one "client" to another, so there can never be any contention between clients for access to the service provided by the receiver.

Of course, there is no reason in principle why our analyses should distinguish only between zero, one, and arbitrarily many uses of a channel: we could just as well introduce the additional multiplicities 2, 3, etc. But, at least in our present experimental testbed (the Pict language), there are few benefits to be gained from this refinement: profiling of Pict programs indicates that the great majority of channels are either used unboundedly or linearly. This is largely due to the fact that functions are literally compiled as processes in Pict, so a very large number of channels are used either as locations of functions or as result channels.

Applying any of these type systems to a full-scale programming language raises a number of pragmatic issues. Although typechecking (even, with some help from programmer annotations, partial type inference) is relatively unproblematic, it is quite difficult to add linear type information without cluttering the language with extra distinctions that the programmer must explicitly bear in mind. Work is underway on exploring these issues in a linear variant of Pict.

Acknowledgements

Kobayashi is partially supported by Grant-in-Aid for Scientific Research of Japan No. 06452389 and 07780232. Pierce is supported by EPSRC grant GR/K 38403. Turner is supported by EPSRC grant 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. Cédric Fournet gave us useful comments on an earlier draft.

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.

[AGN94] Samson Abramsky, Simon J. Gay, and R. Nagarajan. Interaction categories and the foundations of typed concurrent programming. In *Deductive Program Design: Proceedings of the 1994 Marktoberdorf Summer School, NATO ASI Series F.* Springer-Verlag, 1994.

[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.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Principles of Programming Languages*, January 1996.

[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.

[GN95] Simon Gay and Rajagopal Nagarajan. A typed calculus of synchronous processes. In *Proceedings of IEEE Symposium on Logic in Computer Science*, 1995.

[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.

[Hon93] Kohei Honda. Types for dydadic interaction. In *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523, 1993.

[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 asynchronous concurrent programming languages. In *Second International Static Analysis Symposium (SAS'95)*, volume 983 of *Lecture Notes in Computer Science*, pages 225–242. Springer-Verlag, 1995.

[KPT95] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus, 1995. Technical report, Department of Information Science, University of Tokyo and Computer Laboratory, University of Cambridge.

[KY94] Naoki Kobayashi and Akinori Yonezawa. Type-theoretic foundations for concurrent object-oriented programming. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94)*, pages 31–45, 1994.

[KY95] Naoki Kobayashi and Akinori Yonezawa. Towards foundations for concurrent object-oriented programming – types and language design –. *Theory and Practice of Object Systems, John Wiley & Sons*, 1995. to appear.

[LW95a] Xinxin Liu and David Walker. Confluence of processes and systems of objects. In *Proceedings of CAAP'95*, pages 217–231. Springer, 1995.

[LW95b] Xinxin Liu and David Walker. A polymorphic type system for the polyadic π-calculus. In *CONCUR'95: Concurrency Theory*, pages 103–116. Springer, 1995.

[Mac94] Ian Mackie. Lilac: A functional programming language based on linear logic. *Journal of Functional Programming*, 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.

[Nie96] Joachim Niehren. Functional computation as concurrent computation. In *Principles of Programming Languages*, January 1996.

[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.

[NN95] Hanne Riis Nielson and Flemming Nielson. Static and dynamic processor allocation for higher-order concurrent languages. In *TAPSOFT'95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 590–604. Springer-Verlag, 1995.

[Ode95] Martin Odersky. Polarized name passing. In *Proc. FST & TCS*, LNCS. Springer Verlag, December 1995.

[OTPT95] P. W. O'Hearn, M. Takayama, A. J. Power, and R. D. Tennent. Syntactic control of interference revisited. In *MFPS XI, conference on Mathematical Foundations of Program Semantics*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, March 1995.

[PS93] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993. Full version to appear in *Mathematical Structures in Computer Science*.

[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. To appear, 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.

[San93] Davide Sangiorgi. An investigation into functions as processes. In *Proc. Ninth International Conference on the Mathematical Foundations of Programming Semantics (MFPS'93)*, volume 802 of *Lecture Notes in Computer Science*, pages 143–159. Springer Verlag, 1993.

[SN95] Martin Steffen and Uwe Nestmann. Typing confluence. Interner Bericht IMMD7-xx/95, Informatik VII, Universität Erlangen-Nürnberg, 1995.

[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. Forthcoming Ph.D. thesis, LFCS, University of Edinburgh.

[TWM95] David N Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Functional Programming Languages and Computer Architecture*, 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.