Activeness and Responsiveness

Maxime Gamboni

January 30, 2009

Abstract

In this paper we propose a new way of writing channel and process types for the π-calculus, as well as a semantic description of two liveness properties of channels: *activeness* (whether the process is able to receive or send a request on the channel) and *responsiveness* (if a receiver is able to eventually reply to a request).

Any combination of these properties can be associated independently to input and output occurrences of names in a process and to their parameters (thus specifying what a name polar may expect from its complement).

The type notation we propose puts an emphasis on *protocols*: A channel type segregates the server (input) side behaviour from the client (output) behaviour, and the type for a process explicitly specifies how third-party processes are permitted to interact and interfere with it.

We conclude the paper with a survey of some works on related topics along with encodings of the various type notations.

1 Introduction

This work originated from a need to model complex data exchange in π-calculus, in a way that "looks" atomic.

More specifically, we want to be able to encode in polyadic π-calculus (that only allows transmitting names) expressions such as $\overline{a}\langle\xi\rangle$ where ξ is a complex structure that may be of unbounded size (such as a list). The only way this can be achieved is by first sending a name containing a pointer to the data to be transmitted, and then transmitting the data small parts at a time.

One goal of such an encoding is to permit encoded processes to interact with arbitrary π-calculus processes that were not necessarily obtained through the encoding.

To see why we need to restrict which processes may interact with an encoded process, consider the two following processes:

> P receives a value v on channel a, sends a signal on channel s and then decodes value v (discarding the result).
>
> Q receives a value v on channel a, then decodes value v (discarding the result) and finally sends a signal on channel s.

Assuming value decoding is immediate, those two processes are indistinguishable for an external observer as both receive a value on a and then send a signal on s (in case value decoding takes a measurable time, they can be

made bisimilar again by inserting silent actions of corresponding lengths at the corresponding places).

However the encoded forms of these processes are, respectively, as follows[1]:

> $[\![P]\!]$ receives on channel a a name u holding an encoding of value v, then sends a signal on channel s, and finally sends a decoding request on channel u, discarding the reply.
>
> $[\![Q]\!]$ receives on channel a a name u holding an encoding of value v, then sends a decoding request on channel u. *After receiving the reply*, it sends a signal on channel s.

Now these two processes can be distinguished by a process R sending a (private) name u and ignoring any decoding requests: $[\![P]\!]$ will send the success signal but $[\![Q]\!]$ will not, as it will be blocked waiting for a reply to its decoding request.

More generally, to have bisimilarity of processes preserved by the encoding we need the following properties to hold for value encodings:

1. Once sent, the transmitted data is fully determined
2. The data can be accessed by the receiver as many times as it wants, and does not change from one access to the other
3. The sender has no way of knowing when and how many times the data is decoded by the receiver
4. A decoding of the data always succeeds and terminates after a finite time.

The goal of this paper is to provide a precise description of requirement 4.

We start by defining *channel types* that describe how a process may interact at a channel. A process which exhibits the behaviour declared in channel types is said *well-behaved*. We can then define an encoding producing *typed processes* that behave correctly as long as they only interact with well-behaved processes. In the example above, the channel u will be declared "*responsive*" by the encoding, making process R not well-behaved (as it refuses to reply decoding requests). Indeed, as we will show, no well-behaved process will be able to distinguish between P and Q.

The rest of this paper is structured as follows:

Section 2 gives an accurate definition of responsiveness, as well as a notation for channel and process types. We proceed by increments, starting with a very simple notation and gradually adding elements to the types. Subsections 2.7 and 2.8, introduce the property of *activeness* (availability of a server or a client at a channel). Subsection 2.9 introduces the concept of *dependency networks* and subsection 2.10 finally provides a definition for responsiveness. We then close the section with a number of refinements useful with polyadicity and recursive channel types. In section 3 we discuss related work characterising similar properties, and compare their relative expressive power.

[1] Although there are other ways to encode these processes, all exhibit a similar difficulty.

2 Types

We will now describe our needs for the expressiveness of types.

Before starting, a little vocabulary, as it is used in this paper:

"Channels" and "names" have their usual π-calculus meaning, a name being the syntactic element. Unless noted otherwise, lower case latin letters are names. Through renaming, it may happen that two initially different names are assigned to the same channel. A *port* of a channel a is either its input ("a") or output ("$\overline{a}$") half. The letter p stands for a port. A tilde $\tilde{}$ over a symbol stands for a (usually ordered) sequence of elements whose individual elements are represented by the same (tilde-less) symbol with numerical indexes. For instance $\tilde{x}$ stands for $x_1, x_2, \ldots, x_n$.

The calculus considered in this paper is the full synchronous polyadic π-calculus with replication, but not sums, recursion or matching:

$$P \ ::= \ (P|P) \ \mid \ (\boldsymbol{\nu} x)\, P \ \mid \ \mathbf{!}\, P \ \mid \ a(\tilde{x}).P \ \mid \ \overline{a}\langle\tilde{x}\rangle.P \ \mid \ \mathbf{0}$$

2.1 Non-Homogeneous Properties

A first important property is that in such an encoding there are many classes of channels, as the requirements for different channels differ depending on their role in the encoding. In the processes $[\![\, P \,]\!]$ and $[\![\, Q \,]\!]$ given before, there are strong requirements on channel u's input port, but a and s, have very few constraints, if at all, as they play precisely the same role as in the source calculus.

This is the reason for introducing *channel types* : rather than expressing properties of a process as a whole, we focus on channels, and associate channel types (written as σ) to names.

2.2 Parameter types

As names carried over channels can themselves be used as channels, it becomes quickly obvious that a channel type should include the types of its parameters.

For instance, consider the process $P = a(x).\bar{x} \,|\, \overline{a}\langle b\rangle.b(y).\bar{y}$, which reduces to $P \xrightarrow{\tau} \bar{b} \,|\, b(y).\bar{y}$. If a's type does not provide its parameter type, P does not show any immediate typing error, even though P exhibits an arity mismatch after the reduction: The parameterless output $\bar{x}$ just requires x to be parameterless, $b(y).\bar{y}$ requires b to have one parameter, and $\overline{a}\langle b\rangle$ requires a to carry one parameter.

This suggests the notation $\sigma \ ::= \ (\sigma, \sigma, \ldots, \sigma)$ for a channel type (whose recursion ends at parameterless channels, written ()). In the above example, the left component requires a to be of type $\sigma_a = (\sigma_x) = (())$, while the right component requires the type $\sigma_a = (\sigma_b) = ((\sigma_y)) = ((()))$ for a, making the type mismatch obvious.

2.3 Multiplicities

Consider the following situation:

A process A sends a value v to a process B, which then sends a reference to the same value to process C. As explained before, A actually creates a process $[\![\, v \,]\!]_u$ encoding the value v into channel u, and sends the name u to B. B then

sends the same name u to C. Both B and C, to decode the value, send a message on u, which is then replied to by A. Now C has the potential to change the value v as it appears to B, by creating another receiver on u. Now, if the scheduler is fair, on average one half of the decoding requests sent by B will actually be intercepted by C.

A simple way to solve this issue is with the concept of *multiplicities*, which, in their most general form, tell for a channel how many times it may appear in input (respectively, output) subject position. For instance, both ports of a appear in $a|(\boldsymbol{\nu} b)\, b.\bar{a}$ (even though one is deadlocked), and a's input is used once and b not used at all in $(\boldsymbol{\nu} cd)\,(\overline{c}\langle a\rangle \mid \overline{d}\langle b\rangle \mid c(x).x \mid d(y).\mathbf{0})$. We also need to distinguish whether an occurrence is replicated (as a in $!\, a(x).\bar{x}$) or not.

The above issue can now be solved simply by declaring that u's input port has precisely one (replicated) occurrence in subject position, rendering C unable to create one more occurrence without being rejected by a type checker.

The encoding scenario involves the following multiplicities:

1. *Uniform* or *omega* names such as u in the example have one replicated input and an arbitrary number of outputs, replicated or not.

2. A decoding request is a message of the form $\overline{u}\langle l\rangle$ where l is *linear*, meaning that it must occur exactly once in output (for the u-server to send a reply) and exactly once in input (for the request sender to receive the reply).

3. *Plain* names are those that do not have any requirement.

Other cases may occur, as in the internal choice $\bar{a} \,|\, a.P \,|\, a.Q$, where the output port must occur exactly once, and the input port at least once.

Rather than constructing a list of such channel classes we choose to define *port multiplicities* (ranged over by m), and record multiplicities independently for input and output ports. To cover the cases seen so far we need three multiplicities: 1, ω and $\bigstar$, standing respectively for "exactly one non-replicated use", "exactly one replicated use" and "no constraint".[2]

A natural way of writing this information is to put multiplicities as exponent: $\sigma \;::=\; (\tilde{\sigma})^{m_i,m_o}$, where m_i is the input multiplicities and m_o the output multiplicities. For instance $\left((\tilde{\sigma})^{1,1}\right)^{\omega,\bigstar}$ would be the type for u in the encoding example, where $\tilde{\sigma}$ describes how such a request is replied (and depends both on the particular encoding and the source calculus type of the encoded value).

2.4 Local and Remote uses

All examples we have considered so far have been *input-output-alternating*, in that input processes only output on their parameters. A counter-example is a "server creator" $!\, a(x).!\, x(y).Q$ which creates a one parameter server with body Q on all names sent to it. In that example, a type for a would be of the form $\left((\sigma)^{\omega,\bigstar}\right)^{\omega,\bigstar}$. However exactly the same type would be given in the case where the input on x is provided by the *output* of a (as in $\overline{a}\langle b\rangle.!\, b(y).Q$), and yet composing these two processes no longer respects the channel type.

Continuing with the calculus encoding example, we can't require the target processes to have input-output-alternation without requiring processes of the

[2]The "At least one" case is obtained using $\bigstar$ together with "activeness", as shown later.

source calculus to have that property (which is most of the time an unreasonable assumption).

This example shows that giving up the input-output-alternation property requires adding information to channel types as to how uses of the parameters are divided between the input and output side of the channel. One way of expressing this information is in terms of a local/remote separation, which is useful because it can easily be adapted to express the interface between a process and its environment, as we will see in the next section.

Instead of merely recording a total number of port uses for a channel, we write the local and the remote uses separately. To fix a notation, we write α/β to mean α is local and β is remote.

For the parameter uses, we take, as a convention, the point of view of the input process. For instance, consider a two linear parameter channel, whose first parameter is alternating and second is not, as in:

$$a(xy).(\bar{x}|y) \,|\, \overline{a}\langle bc\rangle.(b|\bar{c}) \tag{1}$$

The first parameter then has multiplicities $0, 1/1, 0$, while the second has $1, 0/0, 1$.

Note that this issue only applies to *parameter types*, not to (top-level) channel types. We will provide a similar extension for the channel types in the next section, but, for the moment, distinguish parameter types σ and channel types π ::= $(\tilde{\sigma})^{m_i,m_o}$. This gives the following syntax for parameter types: σ ::= $(\tilde{\sigma})^{m_{li},m_{lo}/m_{ri},m_{ro}}$, where l stands for "local" (i.e., channel's input port), r is remote (channel's output port), i is parameter input and o parameter output.

For instance, 1 has, as a type for a, $((\,)^{0,1/1,0}, (\,)^{1,0/0,1})^{1,1}$ (in order, a's input does zero input on x, one output on x, one input on y and zero output on y, while a's output does one input on b, zero output on b, zero input on c and one output on c. The outer $1,1$ exponent means that a is a linear name, i.e. is used once in input and once in output.

Note that, even though in this example the parameter multiplicities look very symmetric they need not be so. For instance the type $((\,)^{0,\bigstar/\bigstar,\bigstar})^{1,1}$ is for a channel whose input side may only use the parameter in output position, but whose output may use the parameter without restrictions.

2.5 Process Types

In this section we propose a way to use the channel type notation to describe entire processes, with *process types*.

To explain the similarity between channel and process types we consider the interface between a process P and its environment as a special kind of channel whose parameters are the names free in the process. For instance if $\tilde{z} = \mathrm{fn}(P)$ and a is a fresh name, then P's process type is a's channel type in $a(\tilde{z}).P$. E being a process representing the environment, interaction between P and E may then be modelled as τ-reductions following $\overline{a}(\boldsymbol{\nu}\tilde{z}).E \,|\, a(\tilde{z}).P \xrightarrow{\tau} (\boldsymbol{\nu}\tilde{z})\,(E \,|\, P)$.

Using the notation introduced previously, we get Γ ::= $(z_1 : \sigma_1, z_2 : \sigma_2, \ldots, z_n : \sigma_n)^{1,1}$ as a notation for a process type, where z_i covers $\tilde{z} = \mathrm{fn}(P)$.

Two things can be noted in that expression. The first is that the exponent $1,1$ is rather uninteresting (it just means "there is one process and there is one environment"). The second is that the σ_i are of the form $(\tilde{\sigma})^{m_{li},m_{lo}/m_{ri},m_{ro}}$ rather than $(\tilde{\sigma})^{m_i,m_o}$, i.e. they are parameter types rather than channel types.

Note that the local/remote terms make sense now, as these multiplicities tell how the channel usages are divided between local (P) and remote (E).

Consider for example the process $P = \mathbf{!}\,a(x).\bar{x}$. Wrapping it into an input as described above gives $b(a).P$. In that process, the channel type for b (and therefore the process type for P) is $(a : (()^{0,1/1,0})^{\omega,0/0,\bigstar})^{1,1}$. (The "$a$:" label is used because channel names are not numbered and ordered like channel parameters, but it remains essentially the same as a parameter type.) Now consider a process $E = \overline{a}\langle t\rangle.t$ acting as the environment for P. The interaction $P \xrightarrow{a(t)} P|\bar{t} \xrightarrow{\bar{t}} P$ with that process corresponds to the reduction $b(a).P \,|\, \bar{b}(\boldsymbol{\nu} a).E \;\rightarrow\; (\boldsymbol{\nu} a)\left((\mathbf{!}\,a(x).\bar{x}) \,|\, (\overline{a}\langle t\rangle.t)\right) \;\rightarrow\; (\boldsymbol{\nu} a)\left((P|\bar{t}) \,|\, t\right) \;\rightarrow\; (\boldsymbol{\nu} a)\left(P \,|\, \mathbf{0}\right)$. The process type for the intermediary form $P|\bar{t}$ would be $(a : \sigma_a, t : ()^{0,1/1,0})^{1,1}$, where σ_a is a's type seen before. Finally, after t has been consumed, we get $(a : \sigma_a, t : ()^{0,0/0,0})^{1,1}$, expressing the fact that t has been fully used. If, for completeness, we wanted to mention t in the type for P before the first transition, it would have been $t : ()^{0,0/1,1}$, expressing the fact that it may not be used in any way by the process, and the environment may use both ports exactly once. The first transition ($(0, 0/1, 1) \rightarrow (0, 1/1, 0)$ on p's multiplicities) can now be seen as E passing t's output capability to P.

2.6 Types as Triples

In this section we propose a change in channel type notation, to make it more natural and more extensible.

Although there is no serious problem in having channel types of that form in a process type, the issue is that, as the examples showed, multiplicities are not preserved by transitions, while the intuition would suggest that for a channel there should exist a single channel type which remains valid over time.

For instance, in $a|b \xrightarrow{a} b$, a's type (a being assumed linear) evolves as $()^{1,0/0,1} \rightarrow ()^{0,0/0,0}$, and in $a|\bar{a} \xrightarrow{\tau} \mathbf{0}$, a's type evolves as $()^{1,1/0,0} \rightarrow ()^{0,0/0,0}$.

Another issue, perhaps more serious, is that multiplicities are not preserved by composition. For instance, in $P = a|b|\bar{a}$, the first component has $()^{1,0/0,1}$ as a type for a, the second has $()^{0,0/1,1}$, the last has $()^{0,1/1,0}$, and in P, a has type $()^{1,1/0,0}$. So, in a single process, a single channel has four different types (plus $()^{0,0/0,0}$ which is a's type after the reduction on a).

Lastly, the notation introduced here, unlike the one used until now, is easily adaptable to the concept of "parameter protocols" which will be explained later.

All these considerations suggest separating channel types and channel multiplicities, while still keeping the same amount of information.

For a process type we use the notation $(\Sigma; \Xi_L; \Xi_R)$, where Σ maps names to channel types, Ξ_L contains the local channel usage information and Ξ_R contains the remote channel usage information. Similarly, for channel types we use the notation $(\tilde{\sigma}; \xi_I; \xi_O)$ where $\tilde{\sigma}$ is a set of channel types for the parameters and ξ_I, respectively ξ_O, gives the parameter multiplicities found in the channel input, respectively output. Note that it is now no longer necessary to distinguish between channel and parameter types.

A channel type $(\tilde{\sigma})^{m_{li},m_{lo}/m_{ri},m_{ro}}$ for a channel a is separated into $(\tilde{\sigma})$, $a^{m_{li},m_{lo}}$ and $a^{m_{ri},m_{ro}}$, and each parameter type $\sigma_i \in \tilde{\sigma}$ is similarly split into its own parameter sequence, input and output behaviour. i^{m_i,m_o} means that

parameter number i is used m_i times in input position and m_o times in output position.

For instance, all names being assumed linear, $\overline{a}\langle b\rangle.\overline{b}$ has as a process type $\Gamma = (a : \sigma_a, b : (); a^{0,1}, b^{1,1}; a^{1,0}, b^{0,0})$: both a and b are locally output once, b is locally input once (as a consequence of being sent to a) and a is remotely input once, with $\sigma_a = ((); 1^{1,0}; 1^{0,1})$ (the first parameter is parameterless, and a's input performs one input on it while a's output performs one output on it).

In the notation used before this section, the same process type would have been written as $(a : (()^{1,0/0,1})^{0,1/1,0}, b : ()^{1,1/0,0})$, omitting the $1,1$ process type exponent. For more clarity we will typically write $a^m, \bar{a}^{m'}$ instead of $a^{m,m'}$. In channel types, terms with zero exponent (such as 1^0) are usually omitted and so are exponents equal to one (writing for instance $\bar{a}$ instead of $\bar{a}^1$).

In process types, local terms with exponent zero and remote terms with exponent $\bigstar$ are omitted, so that, for instance, the channel a need not be mentioned in a type for process $\mathbf{0}$, as it has local multiplicity zero (in both ports) and remote multiplicity $\bigstar$ for both ports, expressing the fact that the environment has, by default, no constraints on the way it may use the channel.

In that simpler notation, the same process type Γ may be written $(a : \sigma_a, b : (); \bar{a}, b, \bar{b}; a, \bar{a}^0, b^0, \bar{b}^0)$ (the process does an output on a, an input on b and an output on b, while the environment an input on a, no output on a and no interaction on b), with $\sigma_a = ((); 1; \overline{1})$ (the channel carries a parameterless channel, its input does an input on the parameter and its output does an output on the parameter).

It should be clear that this new notation, although more extensible and more sound, is precisely as expressive as the previous one, in that any type can be translated from the old to the new notation and *vice versa*. Also note that the representation of a process type as the channel type of an imaginary process-environment communication channel still holds — we use different symbols to emphasise the fact that process types describe channel names while channel types describe parameter numbers.

2.7 Activeness

An important requirement which is at the centre of this work is the ability to specify in a type that a process should be listening (respectively, ready to send) at an input (resp., output) port. We call this property *activeness* at a port[3].

For example, consider a process decoding a value v and sending a signal on a channel s: $P = a(v).\mathsf{case}\, v\, \mathsf{of}\, (x, y) : \bar{s}$, and a process first sending a signal and then decoding v: $Q = a(v).\bar{s}.\mathsf{case}\, v\, \mathsf{of}\, (x, y) : \mathbf{0}$. These processes could be encoded as $[\![P]\!] = a(u).\overline{u}(\boldsymbol{\nu} r_1 r_2).r_1(x).r_2(y).\bar{s}$ and $[\![Q]\!] = a(u).\bar{s}.\overline{u}(\boldsymbol{\nu} r_1 r_2).r_1(x).r_2(y).\mathbf{0}$, where u holds an encoding of v.

As said in the introduction, $P \sim Q$ but $[\![P]\!] \not\approx [\![Q]\!]$ because they are distinguished by $R = \overline{a}(\boldsymbol{\nu} u).\bot.\mathbf{!}\, u(xy).(\overline{x}\langle b\rangle | \overline{y}\langle c\rangle)$ (where $\bot.P \stackrel{\text{def}}{=} (\boldsymbol{\nu} t)\, t.P$ with $t \notin \mathrm{fn}(P)$). Note that R does not violate any multiplicity constraint, as the receiver on u is present — it is merely deadlocked (*inactive*).

Before we propose a solution, it should be noted that requiring u to be active

[3] *Input* activeness is commonly called receptiveness.

is not enough, as is shown by

$$R = \overline{a}(\boldsymbol{\nu} u).\mathbf{!}\, u(xy).\bot.(\overline{x}\langle b\rangle|\overline{y}\langle c\rangle) \tag{2}$$

where u is active (after the transition $\overline{a}(\boldsymbol{\nu} u)$), but, after u receives a request $r_1 r_2$, the reply itself is not. This will be addressed in the section for *responsiveness*.

Moreover, in order to have a property which is meaningful for nonlinear names we add a *reliability* requirement to activeness.

Consider the process $P = p(x).\bar{x}$, where p is plain (i.e. has multiplicities $\bigstar, \bigstar$). At first sight it might seem natural to declare that p is active in P. However that input is not reliable because, composing P with a process $\overline{p}\langle b\rangle.\bar{s}$ will not necessarily trigger the success signal $\bar{s}$, if a third party $E = \overline{p}\langle c\rangle$. "steals" the input at p. In contrast, the replicated form $\mathbf{!}\, P = \mathbf{!}\, p(x).\bar{x}$ is reliable, because there is an infinite supply of inputs at p and no third party can steal them all (assuming fairness on the scheduler).

Finally, our target being encodings, there will be typically an overhead (in terms of extra τ-transitions) in an encoded process compared to the original one. Therefore we consider it acceptable if a number of τ-transitions are required before a receiver (or sender, for output-activeness) becomes available. Ruling out such "weak activeness" would give what we call *strong activeness* and is characterised by works such as [San99, ABL03].

This gives us an informal definition for activeness:

Definition 2.7.1 (Activeness) *A port p is said* active *in a process P if*

1. *P will eventually (i.e. possibly after a finite number of τ-reductions) contain an unguarded occurrence of p in subject position.*

2. *The port is "reliable", in the sense that no third party can interfere in a way that prevents p from being made available to a process attempting to communicate with that port.*

A more precise definition (in particular making "eventually" and "interference of a third party" more precise) is given in the next section.

We extend channel and process types as follows, to permit requiring activeness on channels: Instead of p^m, we may write $p^m_{\mathbf{A}}$, (where p is a port, a or $\bar{a}$) meaning that the port must be active (and have a multiplicity m).

Assuming σ_p is the type for b and c in the example, the reply channels r_1 and r_2 will have a type such as $\sigma_r = (\sigma_p; 1^{\bigstar}\bar{1}^{\bigstar}; 1^{\bigstar}\bar{1}^{\bigstar})$. The $\bigstar$ exponents and absence of **A**-index mean that both the input and output sides of reply channels are free to interact with the parameters b and c in any way. A type for u can then be written $\sigma_u = (\sigma_r\sigma_r; \bar{1}_{\mathbf{A}}, \bar{2}_{\mathbf{A}}; 1_{\mathbf{A}}, 2_{\mathbf{A}})$, telling that u's input side must provide one active output on both parameters, and u's output side must provide one active input on both parameters. Finally, the channel a will have a type such as $(\sigma_u; \bar{1}^{\bigstar}; 1^{\omega}_{\mathbf{A}}\bar{1}^{\bigstar})$, where both input and output side of a may send requests on the parameter u but a's *output* side must provide one replicated ("ω") and active ("**A**") input at the parameter.

Note that it makes little sense to specify activeness on the remote side of a process type, so we will usually have activeness marks on the local side only.

Some examples:
The type $(a : (), b : (); a_{\mathbf{A}}, b; \bar{a}, \bar{b})$ is a valid description of $a \,|\, b$, of $a.b$ and $a \,|\, \bot.b$, but not of $\bot.a \,|\, b$. It does however correctly describe $(\boldsymbol{\nu} t)\,(\bar{t}|t.a.\mathbf{0})$: the fact that

a is not immediately available is not an issue if it is guaranteed to eventually become so.

The type $(a : (); a^{\bigstar}_{\mathbf{A}}; a^0\bar{a}^{\bigstar})$ is a valid description of $!\, a.\mathbf{0}$, but not of $a.\mathbf{0}$, because the latter is unreliable. $(a : (); a^{\bigstar}_{\mathbf{A}}; a^0\bar{a}^1)$, on the other hand, is a valid description of both processes.

Finally, using the notation $?.P \stackrel{\text{def}}{=} (\boldsymbol{\nu} t)\,(\bar{t} \mid t \mid t.P)$ (t fresh) as a shortcut for an "unreliable prefix", $(a : ((); \bar{1}_{\mathbf{A}}; 1_{\mathbf{A}}); a_{\mathbf{A}}; a^0\bar{a})$ is a valid description of $a(x).\bar{x}$, but neither describes $?.a(x).\bar{x}$ (a is not active) nor $a(x).?.\bar{x}$ (x is not active). Weakening the process type to $(a : ((); \bar{1}_{\mathbf{A}}; 1_{\mathbf{A}}); a; a^0\bar{a})$ allows describing the first two processes, but still not the last: It is no longer required for a to be active, but if a request is received then it *must* be replied, because the parameter is declared active in the channel type.

2.8 A word about Fairness

In this section we clarify the meaning of the (ambiguous) word "eventually" used in the Definition 2.7.1 for activeness. That word hides a *fairness* assumption on the scheduler, and we now consider a series of processes with increasing requirements for activeness, and, for each, we discuss whether a port $\bar{s}$ should be considered active. This will help indicating more accurately what is required of the scheduler. We assume s is bilinear, so that reliability is not an issue.

The simplest example is $P_1 = \bar{s}$, where $\bar{s}$ is *strongly active*.

Now an unrelated but unfinishing computation should not affect activeness: In $P_2 = \bar{s}|\Omega$ (where Ω is any process with $\Omega \rightarrow \Omega$, for instance $(\boldsymbol{\nu} t)\,(\bar{t} \mid !\, t.\bar{t})$), $\bar{s}$ is still strongly active.

As already said, a finite number of transitions preserves activeness (but no longer strong activeness), as seen in $P_3 = \tau.\tau \ldots \tau.\bar{s} \mid \Omega$ (for a finite number of τ, and where $\tau.P \stackrel{\text{def}}{=} (\boldsymbol{\nu} t)\,(\bar{t}|t.P)$ for some fresh t).

The following example is more interesting in that the number of transitions is no longer bounded:

$$P_4 = !\, a(x).\bar{x} \mid !\, a(x).\overline{a}(\boldsymbol{\nu} y).y.\bar{x} \mid \overline{a}\langle s\rangle$$

In this process, every request sent to a may be received by the first or by the second input. The first input immediately responds to requests while the second one resends the request. So, the request $\overline{a}\langle s\rangle$ will, under a fair scheduler, loop in the second input for a while, and then eventually be passed to the first input, after which the requests to the second input cascade back until the $\bar{s}$ output is fired. It seems therefore reasonable to consider $\bar{s}$ to be active in this process, as it matches the accepted definition of fairness (see for instance [PT00], Section 2.9, as well as [CC04]), in that the first input is continuously available, and at each step there is a request to a available, so that the first input should eventually catch one request, after which we are back to P_3.

A last example, which, in our opinion, would be requiring too much from a scheduler (as even a stochastic scheduler would not satisfy it), is the following. This example shows that a naive activeness (of $\bar{s}$ in P) definition such as "$\forall Q$ s.t. $P \Rightarrow Q$, $Q\Downarrow_{\bar{s}}$" would be too weak.

$$P_5 = !\, a(x).\bar{x} \mid !\, a(x).\overline{a}(\boldsymbol{\nu} y).y.\overline{a}\langle x\rangle \mid \overline{a}\langle s\rangle \tag{3}$$

In this example, the second output, when receiving the reply to its own request $\overline{a}\langle y\rangle$, re-sends the request it received rather than replying to it, so that the global behaviour is analogous to a random walk. Although a stochastic scheduler (randomly and independently choosing one a-input for each a-output) will eventually reach the output at s, adding more copies of the second input to the program will have the probability of $\bar{s}$ being fired fall to zero.

The fundamental difference between P_4 and P_5 is that in the former, at any point, there is a possibility of *progress* towards strong activeness of $\bar{s}$. In other words, at any time, there exists a *strong* transition that brings the process "closer" to firing $\bar{s}$. In P_4 this progress is very simple, in that having the first input handle a request passes from a process where the number of required τ-transitions is not bounded, to one where it is bounded. A process P_4' where the progress is slightly more elaborate would be obtained by replacing $\overline{a}\langle s\rangle$ in that process by $\overline{a}\langle s_1\rangle.s_1.\overline{a}\langle s_2\rangle.s_2 \ldots s_{n-1}.\overline{a}\langle s\rangle$. In that case, the "distance" towards an output at s is n, and is reduced by one every time the first a-input is used. When that distance reaches zero, we are back to case P_3, with a finite number of transitions. The usual fairness assumption now works, because if at any point in time the scheduler has the possibility to make an (irreversible) progress towards strong activeness, and if the number of times such progress is required is bound (it is 1 for P_4 and n for P_4'), then $\bar{s}$ will eventually be fired. In P_5, no such irreversible progress occurs, because any diminution of the call stack can be cancelled by calling the second a-input a sufficient number of times.

In order to obtain a precise definition for activeness we introduce a "game" between two players (for brevity, this definition assumes s to be linear and its local multiplicities to be $s^0\bar{s}^1$). Combining this definition with 2.7.1 poses no difficulties and mainly consists in replacing Player 1's τ-transitions by arbitrary transitions, as long as the port being tested has a non-zero remote multiplicity.

Definition 2.8.1 (Fairness) *The linear port p is* active *in P if Player 2 has a winning strategy in the following game (where "current process" is initially P):*

Player 1 plays first, and, at each turn, may replace the current process P' with any process Q such that $P' \Rightarrow Q$.

Player 2, at each turn, may either do nothing or replace the current process P' with any process Q such that $P' \xrightarrow{\tau} Q$.

Player 2 has won if the current process P' ever satisfies the $P'{\downarrow_p}$ property.

In that definition, Player 2 models the "opportunities" the scheduler has to make progress, while player 1 models the times when the scheduler doesn't "take advantage" of those opportunities.

It is now clear that, in P_4, player 2 simply connects any existing a-output to the first input, and wins, while, in P_5, player 1 simply activates the second input at least once at every turn, preventing $\bar{s}$ to ever become available.

2.9 Conditional Activeness

Consider the process $P = a.b$ where both a and b are linear. According to the definition given above, b is not active, simply because it is blocked by a.

This makes $(a : (), b : (); a_{\mathbf{A}}, b; a^0\overline{a}, b^0\overline{b})$ a correct type for P (while $(a : (), b : (); a_{\mathbf{A}}, b_{\mathbf{A}}; a^0\overline{a}, b^0\overline{b})$ would be incorrect).

However, if P is composed with a process where $\bar{a}$ is active, b becomes active as well: $P|\bar{a}$ has a (weak) transition using a as subject. In order to permit a type system to infer that form of causality we represent in the process type the relation between a and b with the notation $\bar{a}_{\mathbf{A}} < b_{\mathbf{A}}$. The order relation symbol is used in the sense "after $\bar{a}$ is active, b will be active". A more accurate process type for P (which helps distinguishing it from $b\,|\,\bot.a$ where a can't be made active no matter what the process is composed with) is therefore $(a:(), b:(); \bar{b}_{\mathbf{A}} < a_{\mathbf{A}}, b_{\mathbf{A}}; a^0\bar{a}, b^0\bar{b})$.

More generally, an element in the *local* portion of a process type (the second component) is of the form $\varepsilon p^m_{\mathbf{A}}$ ($p = a$ or $p = \bar{a}$ for some name a), where ε is a predicate (called "dependency") obeying the following grammar:

$$\varepsilon \;::=\; p^m_{\mathbf{A}} < \;\;\big|\;\; p^m_{\mathbf{A}} \leq \;\;\big|\;\; (\varepsilon|\varepsilon) \;\;\big|\;\; (\varepsilon\&\varepsilon) \;\;\big|\;\; \neg \;\;\big|\;\; \emptyset$$

The symbol $<$ stands for a *strong dependency* as in the situation above, while $\leq$ stands for a *weak dependency* and will be described in section 2.13. The operators $|$ and $\&$ have their usual meanings as disjunction and conjunction, $\neg$ is the "unsatisfiable dependency" ($\neg p_{\mathbf{A}}$ means p is not active) and $\emptyset$, usually omitted, stands for unconditional activeness. We also typically omit the $\&$ operator, writing just $\varepsilon\varepsilon'$ for $\varepsilon\&\varepsilon'$. In the following sections we will first generalise the dependency system as far as our application requires, before applying the corresponding extension to channel types.

2.10 Responsiveness

In this section we study relationships between activeness of a channel and activeness of names carried in that channel. Specifically, consider again the process (2). After the transition $\overline{a}(\boldsymbol{\nu} u)$, all free names are active, but, as already discussed, that process is not acceptable because after a transition $u(r_1 r_2)$, it contains two inactive names r_1 and r_2. One way to solve this would be to strengthen the definition of activeness, but, instead, we choose to introduce a separate concept, *responsiveness*, so that dependencies of activeness and responsiveness may be computed separately, which permits typing more processes.

Definition 2.10.1 (Responsiveness) *Given its type σ, we say a port is* responsive *if, whenever it is consumed, all its parameters are themselves responsive, and the ones declared active in σ actually are active.*

For instance, in R as given in (2), u is active (after a transition $\overline{a}(\boldsymbol{\nu} u)$) but not responsive, because an additional transition labelled $u(r_1 r_2)$ leads to a process where neither r_i are active, though they are declared so in the channel type.

We now generalise process types to allow dependencies between activeness and responsiveness.

Just as ε in $\varepsilon p^m_{\mathbf{A}}$ gives the dependencies for p's activeness, $\varepsilon p^m_{\mathbf{R}}$ gives the dependencies for p's responsiveness. Note that specifying twice the multiplicities of a single port (as m in $\varepsilon p^m_{\mathbf{A}}, \varepsilon' p^m_{\mathbf{R}}$) is redundant, so we typically omit one of them, or, alternatively, use separate terms for multiplicities and dependencies, as in $p^m, \varepsilon p_{\mathbf{A}}, \varepsilon' p_{\mathbf{R}}$.

The following example illustrates this:

$$t.a(x).u.\bar{x}$$

As far as activeness is concerned, we have $\emptyset t_{\mathbf{A}}$, $\bar{t}_{\mathbf{A}} < a_{\mathbf{A}}$, $\bar{t}_{\mathbf{A}}\bar{a}_{\mathbf{A}} < u_{\mathbf{A}}$, and, after a has been consumed and x made visible, $\bar{u}_{\mathbf{A}} < \bar{x}_{\mathbf{A}}$.

By definition, $\bar{x}_{\mathbf{A}} \leq a_{\mathbf{R}}$ (a is responsive if $\bar{x}$ is active — the reason for using a weak dependency "$\leq$" rather than a strong one will be clarified later, but this distinction is not important at this point of the discussion), which gives us $\bar{u}_{\mathbf{A}} < a_{\mathbf{R}}$. Why doesn't a's responsiveness depend on $\bar{t}_{\mathbf{A}}$? The idea is that responsiveness's dependencies are those that are required to provide a reply *after a request has been received*. In this case, $\bar{t}_{\mathbf{A}}$ is no longer needed once a has received a request, but $\bar{u}_{\mathbf{A}}$ is required to answer it. Inversely, $\bar{t}_{\mathbf{A}}$ is required for a communication on a to take place, but $\bar{u}_{\mathbf{A}}$ is not needed for that.

The following process (where a is plain active) is another illustration of the duality between activeness and responsiveness:

$$t_1.a(x).u_1.\bar{x} \,|\, t_2.a(x).u_2.\bar{x}$$

Now we have $(\bar{t}_{1\mathbf{A}}|\bar{t}_{2\mathbf{A}}) < a_{\mathbf{A}}$ and $\bar{u}_{1\mathbf{A}}\bar{u}_{2\mathbf{A}} < a_{\mathbf{R}}$: any of the $\bar{t}_{i\mathbf{A}}$ must be provided for a to be active, but *both* $\bar{u}_{i\mathbf{A}}$ must be provided for a to be responsive. The reason is that the sender can't know for certain which input on a will receive the request, and therefore must provide both $\bar{u}_i$ to be certain the request gets replied.

Finally, the following process shows why keeping activeness and responsiveness separate when computing dependencies is interesting:

$$\overline{a}\langle t\rangle.!\, b(x).\bar{x} \mid !\, a(y).\bar{b}\langle y\rangle \tag{4}$$

We have both $a_{\mathbf{A}} < b_{\mathbf{A}}$ (because of the left-hand component) and $b_{\mathbf{R}} < a_{\mathbf{R}}$ (because of the right-hand component), and yet the process isn't deadlocked. However, not distinguishing $a_{\mathbf{A}}$ and $a_{\mathbf{R}}$ would result in the circularity "$a < b < a$" and have the process rejected.

2.11 Labelled Dependencies

In this section we describe more formally how computing responsiveness dependencies work, and more precisely how to describe the "dependencies after a request has been received" idea.

For simplicity (to avoid confusion arising from parameter binding inherent to inputs) we only consider *output* responsiveness for the moment. The output version of an earlier example is as follows:

$$t.\overline{a}\langle b\rangle.u.\bar{b}$$

The relevant dependencies are $\bar{t}_{\mathbf{A}} < \bar{a}_{\mathbf{A}}$, $\bar{t}_{\mathbf{A}}a_{\mathbf{A}}\bar{u}_{\mathbf{A}} < \bar{b}_{\mathbf{A}}$ and $\bar{b}_{\mathbf{A}} \leq \bar{a}_{\mathbf{R}}$. However, composing all these dependencies would result in $\bar{t}_{\mathbf{A}}a_{\mathbf{A}}\bar{u}_{\mathbf{A}} < \bar{a}_{\mathbf{R}}$, which is not correct, as explained in the previous section. In terms of dependencies, the reason is that not all $\bar{b}_{\mathbf{A}}$'s dependencies are relevant when computing the ones for $a_{\mathbf{R}}$.

We propose using a notation of *labelled dependencies* to solve this issue. (Note that this notation is only used as an intermediary construct while type checking). The syntax for dependencies is extended as follows:

$$\varepsilon \;::=\; \cdots \;\;\big|\;\; l : \varepsilon \;\;\big|\;\; \neg l : \varepsilon$$

The $l : \varepsilon$ dependency (l being a "label" from some infinite set) places a "marker" on part or all of a dependency for a resource, and $\neg l : \varepsilon$ indicates that said dependency is only relevant *outside* of regions marked with l. More concretely, performing the substitution $(l : \varepsilon)\{^{\varepsilon'}/_{\alpha}\}$ will replace, in ε', all sub-expressions labelled with $\neg l :$ by $\emptyset$.

We can now label dependencies for responsive resources, and have all its active resources labelled with the corresponding negative label, as follows: $(\neg l : \bar{t}_{\mathbf{A}} a_{\mathbf{A}}) \& \bar{u}_{\mathbf{A}} < \bar{b}_{\mathbf{A}}$ and $l : \bar{b}_{\mathbf{A}} \leq \bar{a}_{\mathbf{R}}$. Now, substituting $\bar{b}_{\mathbf{A}}$ by $(\neg l : \bar{t}_{\mathbf{A}} a_{\mathbf{A}}) \& \bar{u}_{\mathbf{A}} <$ in $\bar{a}_{\mathbf{R}}$'s dependencies results in $l : ((\neg l : \bar{t}_{\mathbf{A}} a_{\mathbf{A}}) \& \bar{u}_{\mathbf{A}}) < \bar{a}_{\mathbf{R}}$ which reduces to $l : (\emptyset \& \bar{u}_{\mathbf{A}}) \leq \bar{a}_{\mathbf{R}}$ and then to $l : \bar{u}_{\mathbf{A}} < \bar{a}_{\mathbf{R}}$, as required.

Such labels can be obtained automatically for any process, but, for brevity, we do not detail the algorithm here.

2.12 Parameter Protocols

Channel types, as described so far, indicate what resources the input and output side of a channel must provide to the other end. As already said, we are working in polyadic π-calculus, and as a consequence it can be interesting to permit the two ends of a channel to negotiate these resources, rather than requiring each end to provide everything without cooperation from the other end. In this section we describe *parameter protocols*, which indicate to what extend such a negotiation can occur over a given channel.

In the simple case where channel parameter don't themselves carry parameters, the parameter protocol indicates in what order the parameters are to be provided, as a partial order (where $x < y$ indicates y *may* wait for x to be provided before becoming itself available). One example is the "left to right protocol" where parameter resources are ordered according to their position, so that $a(xy).\bar{x}.\bar{y}$ and $a(xy).(\bar{x}|\bar{y})$ are valid but $a(xy).\bar{y}.\bar{x}$ is not, and similarly for the output, $\overline{a}\langle bc\rangle.b.c$ and $\overline{a}\langle bc\rangle.(b|c)$ are valid but $\overline{a}\langle bc\rangle.c.b$ is not). It is clear in this example that composing a valid input with a valid output does not result in a deadlock, but using an invalid input or output may result in a deadlock, as in $a(xy).\bar{x}.\bar{y} \,|\, \overline{a}\langle bc\rangle.c.b$, which reduces to $\bar{b}.\bar{c} \,|\, c.b$ which is deadlocked.

A more interesting example is the following, which creates on the first parameter a "cached copy" of a server passed as second parameter:

$$!\, a(bc).\overline{c}(\boldsymbol{\nu} x).x(t).!\, b(y).\overline{y}\langle t\rangle \tag{5}$$

The channel type for a (the protocol being omitted, for the moment) is $(\sigma\sigma; 1^{\omega}\bar{2}^{\bigstar}; \bar{1}^{\bigstar}2^{\omega})$, with σ being any single parameter channel. In this case the protocol indicates that activeness and responsiveness on the first parameter may depend on activeness and responsiveness on the second parameter (i.e. it is a sort of right-to-left protocol but also involving responsiveness. We could drop the right-to-left dependency on activeness by having the input on a immediately available, but fetch the data from b on first use).

As far as notation is concerned, we apply the "process type as a channel type" metaphor backwards, and simply put dependency statements in channel types. The first example can now be typed with the channel type $(\lambda\lambda; \bar{1}_{\mathbf{A}}, 1_{\mathbf{A}} < \bar{2}_{\mathbf{A}}; 1_{\mathbf{A}}, \bar{1}_{\mathbf{A}} < 2_{\mathbf{A}})$, and the second with $(\sigma\sigma; 2_{\mathbf{AR}} < 1^{\omega}_{\mathbf{AR}}\bar{2}^{\bigstar}; \bar{1}^{\bigstar}2^{\omega}_{\mathbf{AR}})$.

Note that a simple forwarder creator $!\, a(bc).!\, b(x).\bar{c}\langle x\rangle$ has precisely the same channel type as (5), showing that whether caching is performed or not is transparent as far as types are concerned.

There are two important differences compared to dependency network in process types, however:

- Both the second and third components of a channel type can contain dependencies, unlike process types where having dependencies on the remote side wouldn't make much sense.

- Only strong dependencies, and dependency conjunction are useful in channel types: Having a disjunction would be equivalent to having a conjunction, because "allowing a port to use α or β" is equivalent to "allowing a port to use α and β" — the other end will anyway have to provide both to cover all cases. Moreover, having a $\neg n_{\mathbf{A}}$ is equivalent to not mentioning it at all in the channel type, and $\neg n_{\mathbf{R}}$ would be invalid — such a statement in a process type indicates an error detected by the type system, and including it in a channel type would mean the corresponding receiver is *required* to have such an error. Labelled dependencies are only used as intermediary expression during type checking and therefore don't belong in channel types (which are meant as a specification).

2.13 Weak Dependencies

Weak dependencies briefly mentioned before differ from strong dependencies in that a loop containing only weak dependencies vanishes (for instance $\alpha \leq \beta$ and $\beta \leq \alpha$ reduces to $\emptyset\alpha$ and $\emptyset\beta$, meaning both are available), while a loop containing at least one strong dependency ($\alpha < \beta$ and $\beta < \alpha$ reduce to $\neg\alpha$ and $\neg\beta$, meaning neither is available) indicates a deadlock.

The concept of weak dependencies is not crucial to our needs (replacing them with strong dependencies still types a good number of processes), but it doesn't complicate the manipulation of types, and has a number of advantages:

- The programmer is not required to fully specify the parameter protocol for a channel type, as it can be "completed" by putting, for every pair $(\alpha; \beta)$ of parameter resources, bidirectional weak dependencies like $\{\alpha \leq \beta, \beta \leq \alpha\}$. That way, if the process attempts to create a (strong) dependency either way, a loop containing at least a strong dependency is created, resulting in a deadlock. This permits defining the "empty protocol", which simply forbids dependencies between the input and the output side of the channel.

- "Apparent circularities" in dependencies, connecting two different nesting levels of a recursive channel type, is not a deadlock, and will be accepted by the type system if weak dependencies are used to connect local responsiveness to activeness and responsiveness on local resources. For instance, $!\, a(x).\bar{x}\langle a\rangle$ is a server returning a pointer to a server of the same type, and is responsive, because it only contains the weak loop $a_{\mathbf{R}} \leq \bar{x}_{\mathbf{R}} \leq a_{\mathbf{R}}$. This

can be useful for encoding a sequential process containing loop, where invoking the encoding (a) of a line in the program returns (on x) a pointer to the next line to be executed. In this example, responsiveness means that an individual line will complete, not that the encoded program will terminate, and, in particular, such an apparent circularity appears when encoding a loop.

- Many natural extensions of the type systems can make use of weak dependencies. For instance, an extension for checking determinism will type a forwarder $!\, a(x).\bar{b}\langle x\rangle$ as having a's determinism *weakly* depend on b's determinism — indeed, loops in such relations do not prevent determinism.

3 Related work

In this section we present some related research, together with, when possible, an encoding of their notation into ours, to help comparison.

3.1 Sangiorgi — The Name Discipline of Uniform Receptiveness

This [San99] is one of the first papers to address the property of activeness (which they call "receptiveness"). It works on asynchronous monadic π-calculus with sums and matching (which we don't handle). A *linear receptive* name corresponds, in our terminology, to bi-linear names that are input active, like a in $a^1_{\mathbf{A}}\bar{a}^1$, and an *$\omega$-receptive* name is the same, but with ω multiplicity on input and plain multiplicity on output, like $a^\omega_{\mathbf{A}}\bar{a}^\star$.

Their $(\Gamma;\Delta)$ process types can then be translated into our process types by having a name a's local multiplicities be $\bar{a}^{\Gamma(a)}a^{\Delta(a)}$ for the linear type system (with $A(a) = 1$ if $a \in A$ and 0 otherwise), and the complement multiplicities $\bar{a}^{1-\Gamma(a)}a^{1-\Delta(a)}$ on the remote side. For the ω-receptiveness type system, we have, for each a, $\bar{a}^{\star\Gamma(a)}a^{\omega\Delta(a)}$ on the local side, and $\bar{a}^\star a^{\omega(1-\Delta(a))}$ on the remote one. Sangiorgi's *plain names* correspond to $a^\star\bar{a}^\star$, both locally and remotely (names plain on both ports, and without activeness).

Note however that his type system is typing strong activeness, so that it does not require dependency analysis, but also is not subsumed by ours. If however we weaken his soundness theorem to allow a weak input transition when using a receptive name, then our semantic definition matches his, and typability of our type system strictly implies his.

He also provides definitions for labelled bisimilarity and barbed equivalence that respect the concept of receptiveness. Generalising those definitions, in particular 5.3, the one for labelled bisimilarity, would however require some work, because if receptive names are allowed to carry receptive names, then the $x \triangleright v$ sub-process is not complete.

3.2 Pierce, Sangiorgi: Typing and Subtyping for Mobile Processes

This paper [PS93] studies input and output capabilities (in our terminology, types such as $\emptyset$, $a^\star$, $\bar{a}^\star$, and $a^\star\bar{a}^\star$), and establishes a *subtyping* relation, which

permits typing $\overline{a}\langle x\rangle$ while having x's type different from a's parameter type (using the subtyping relation covariantly or contravariantly depending on which capabilities of x are used by a's receiver).

Their types $(\tilde{S})^I$ with $I \in \{\mathbf{r}, \mathbf{w}, \mathbf{b}\}$ are easily encoded into our notation, as follows:

$$[\![\, a : (\tilde{S})^I \,]\!] \stackrel{\text{def}}{=} (a : ([\![\, \tilde{S} \,]\!]); a^{\bigstar I_{\mathbf{r}}} \overline{a}^{\bigstar I_{\mathbf{w}}}; a^{\bigstar \bar{I}_{\mathbf{r}}} \overline{a}^{\bigstar \bar{I}_{\mathbf{w}}})$$

where $\bigstar I_c$ is $\bigstar$ if $I \leq c$, 0 otherwise, where $\bigstar \bar{I}_c$ is the same but using $c \leq I$, and $[\![\, S_1, \ldots, S_n \,]\!]$ is an abbreviation of $[\![\, 1 : S_1 \,]\!], \ldots, [\![\, n : S_n \,]\!]$.

Their types are thus more specific (all names are plain and none can be declared active) but, with equivalent types, their type system accepts more processes than ours, thanks to subtyping.

3.3 Kobayashi, Pierce, Turner: Linearity and the π-calculus

That paper [KPT99] is a specialisation of our system in that they only have inert (multiplicity zero), linear (only one port is used, and linearly), bi-linear (both ports are linear) and plain names (which they call ω), and no behavioural property. They also introduce $(\omega; \bigstar)$ channels in section 7.3 (and call them $*$). Like in Section 3.2, we can encode their types as follows:

$$[\![\, a : p^m[\tilde{T}] \,]\!] \stackrel{\text{def}}{=} (a : ([\![\, \tilde{T} \,]\!]); a^{[\![\, m \,]\!]_{p_i}} \overline{a}^{[\![\, m \,]\!]_{p_o}}; a^{[\![\, m \,]\!]_{\bar{p}_i}} \overline{a}^{[\![\, m \,]\!]_{\bar{p}_o}})$$

where mp_c is m if $c \in p$, 0 otherwise, $[\![\, 1 \,]\!] \stackrel{\text{def}}{=} 1$, and $[\![\, \omega \,]\!] = \bigstar$. $[\![\, T_1, \ldots, T_n \,]\!]$ is an abbreviation of $[\![\, 1 : T_1 \,]\!], \ldots, [\![\, n : T_n \,]\!]$.

They provide definitions for barbed bisimilarity, and show some confluence results for linear channels.

3.4 Amadio et al.: The Receptive Distributed π-calculus

As the title suggests, this paper [ABL03] is on a distributed setting, where they have the additional issue that, for a communication to succeed, its two ends must be at the same site (which requires extra care when checking for deadlocks). They also have matching, on a special set of names called *keys*.

So, the setting is more complex, with the trade off that their types are very simple — all names are (in our terminology) active non-uniform ω input and plain output and, just like [San99], they guarantee *strong* activeness, where no internal action is tolerated between creation of a new name and it being ready to use). More importantly, as a consequence of having I/O alternation and only input activeness, they are only concerned about messaged being *received* — no reply is guaranteed.

Their work is mainly interesting in the distributed setting — restricting it to a local setting would reduce to the essentially syntactic check that all outputs have at least one corresponding unguarded input.

Also note that they concentrate on *non-uniform* activeness based on recursion (like a in $\mu X.a(x).(\overline{x}\langle t\rangle \mid a(y).(\overline{x}\langle t'\rangle \mid X))$ where $\mu X.P$ stands for a recursive process), which can't be characterised in our type system without modification, as the closest we have is *uniform* activeness obtained through replication.

3.5 Acciai, Boreale: Responsiveness in process calculi

This paper [AB08a] addresses concerns very close to ours, through two distinct type systems. Again, their setting is simpler than ours, in that it works on synchronous π, I/O alternating and doesn't consider combinations of active and non-active names. On the other hand, they present, with their system $\vdash_1$, an extension for recursive processes which is more powerful than our type system, in that it permits handling unbounded recursion such as a function computing the factorial of its parameter:, $!\, f(n,r).\,\mathrm{if}(n=0)\,\overline{r}\langle 1\rangle\,\mathrm{else}\,(\boldsymbol{\nu} r')\,(\overline{f}\langle n-1, r'\rangle \mid r'(m).\overline{r}\langle n*m\rangle)$. A naive dependency analysis would reject such a process, because the recursive call would create a dependency $f_{\mathbf{R}} < f_{\mathbf{R}}$.

We conjecture that their analysis, based on the well-foundedness of parameter domains, could be adapted to our dependency networks by having a $b_{\mathbf{R}} < a_{\mathbf{R}}$ dependency be *weak* if b's parameter tuple is "lighter" than a's. In the factorial example, $\langle n-1, r'\rangle$ being "lighter" than $\langle n, r\rangle$ (because $n-1 < n$), the self-dependency becomes $f_{\mathbf{R}} \leq f_{\mathbf{R}}$ and cancels out.

3.5.1 Types

A channel type can be *responsive*, *ω-receptive* or *$+$-responsive*. For the last case they use a concept mostly equivalent to our multiplicities, which they call "capabilities". Their channel types can then be encoded into ours as follows:

- Inert type: $[\![\, a : I \,]\!] = (a : (); ; a^0 \bar{a}^0)$
- Responsive name: $[\![\, a : T^{[\rho,k]} \,]\!] = (a : ([\![\, 1 : T \,]\!]); a_{\mathbf{A}} \bar{a}_{\mathbf{A}}; a^0 \bar{a}^0)$
- Responsive parameter: $[\![\, 1 : T^{[\rho,k]} \,]\!] = (1 : ([\![\, 1 : T \,]\!]); \bar{a}_{\mathbf{A}}; a_{\mathbf{A}})$
- ω-receptive name: $[\![\, a : T^{[\omega,k]} \,]\!] = (a : ([\![\, 1 : T \,]\!]); a^{\omega}_{\mathbf{A}} \bar{a}^{\bigstar}; a^0)$
- ω-receptive parameter: $[\![\, 1 : T^{[\omega,k]} \,]\!] = (1 : ([\![\, 1 : T \,]\!]); \bar{a}^{\bigstar}; a^{\omega}_{\mathbf{A}})$
- $+$-responsive names are encoded similarly, using the following correspondance: on inputs, capabilities n, s, m and p correspond respectively to total multiplicities 0, 1, $\bigstar$ and ω, and on outputs, n, s, m and p correspond respectively to total multiplicities 0, $\bigstar$, $\bigstar$ and ω.

We have no way to prevent a name to be sent around (in object position), so their $\bot$ type can't be encoded. Encoding it like I is a good approximation, however. Also, their levels k are ignored by this encoding, because they are implicitly contained in the dependency network which is inferred by the type system. Those levels basically put an upper bound on the length of substitution chains ($\{^{\beta}/_{\alpha}\}\{^{\gamma}/_{\beta}\}\cdots$) that can be done in activeness dependencies before reaching the $\emptyset$-dependency. The above encoding is not completely accurate but corresponds to what their type system enforces.

3.5.2 Semantics

As far as terminology is concerned, their "responsiveness" property mostly corresponds to our "activeness" property, on processes in which responsiveness (in our terminology) holds on all names. It is not strictly equivalent because we work with a labelled transition system and define activeness and responsiveness

in terms of interactions with the environment, while they work in a reduction setting, and define responsiveness in terms of internal actions. The correspondence can be made by comparing our activeness on a port $p \in \{a, \bar{a}\}$ in a process P to their responsiveness on channel a in a process like $P \mid Q$ where Q is a process interacting on $\bar{p}$ (such as $\overline{a}\langle b\rangle$ or $a(x).Q'$, depending on p).

Note that their semantic definition is also weaker as it accepts as responsive channel a in "unbalanced" processes like $(a \mid \bar{a}) \mid a$ or $(a \mid \bar{a}) \mid \bar{a}$, where the rightmost a or $\bar{a}$ can be seen as the "testing" process Q, but may not succeed. Also they require more than fairness on the scheduler as they would consider s responsive in process $P_5 \mid s$ (where P_5 is given by equation (3)). However it seems that strenghtening their semantic definition to reject such cases would preserve soundness of their type system.

It should be noted also that they require *all* names to be "responsive" (or ω-receptive, which is essentially the same but with another multiplicity) — they don't consider processes where both "plain" and "responsive" names are involved.

3.5.3 Power

The base form of both their type systems, described in their sections 3 and 6 are strictly subsumed by ours.

Similarly to what was presented in this paper, their first type system uses a dependency network is used to check strong linear activeness or strong ω-activeness on input ports, and activeness for linear output ports. For a process like $\bar{b} \mid b.\bar{a}$, a dependency $a \to b$ indicates the order in which linear channels are consumed. it uses *levels* to check delegation, in a way that corresponds more or less to our *responsiveness* dependency chains, e.g. $!\, a(x).\bar{b}\langle x\rangle$ requires b's level to be smaller than a's.

Their first system rejects a number of processes accepted by our type system, such as "half-linear names" like t in $(\boldsymbol{\nu} t)\,(\bar{t} \mid t.P \mid t.Q)$, as well as processes such as $(\boldsymbol{\nu} a)\,(a(x).(\bar{x} \mid b(y).\bar{y}) | \overline{a}\langle t\rangle)$ because the input on b is not immediately available. It is however weakly bisimilar to $b(y).\bar{y}$, which is typable.

On the other hand the extension for handling recursive functions goes beyond what our type system is capable of, as already said.

The second type system allows guarded inputs, the "half-linear names" already mentioned and replicated outputs, but rejects some recursive functions such as the "factorial" one given previously. It is also strictly subsumed by ours because for instance they do not allow guarded free replicated inputs.

We would like to point out that this paper answers the question they rise at the end of Section 6.2, concerning the generalisation of dependency graphs when inputs may be nested. They give an example of process that would require such a generalisation: $b(x).\overline{a}\langle x\rangle \mid c(x).a(y).\overline{x}\langle y\rangle \mid \overline{c}\langle b\rangle$, where all names are assumed responsive (in their terminology, or "bi-linear active" in ours). That process should be ruled out because it reduces to $b(x).\overline{a}\langle x\rangle \mid a(y).\bar{b}\langle y\rangle$, where a and b are now clearly deadlocked. Using dependency graphs on responsiveness (in addition to activeness) rules out the first process, because it contains the cycles $b_{\mathbf{R}} \leq \bar{c}_{\mathbf{R}} < a_{\mathbf{R}} < b_{\mathbf{R}}$ and $c_{\mathbf{R}} < \bar{b}_{\mathbf{R}} < \bar{a}_{\mathbf{R}} < c_{\mathbf{R}}$.

In conclusion, generalising their analysis of recursion on well-founded domains on our type system would give a type system that is strictly more powerful

than both their systems, so that it is no longer necessary to have two separate systems with different typing strategies.

3.6 Acciai and Boreale — Spatial and Behavioral Types in the Pi-Calculus

This [AB08b] combines ideas from spatial logics and from [IK01]. Their systems relies on spatial model checking, but properties — both safety and liveness ones — are checked against types rather than against processes. Implementation and complexity issues, as well as the degree of completeness of the approach, are not treated, but naturally one expects a type checking system to be simple to implement and be of low complexity.

3.7 Kobayashi — TyPiCal

This [Kob08] is an implementation of a lock-freedom type system. Although it also performs termination and information flow analysis we are particularly interested in its lock-freedom analysis.

3.7.1 Terminology

We first introduce a few concepts used by TyPiCal when analysing processes.

Definition 3.7.1 (Deadlock) *An input or output prefix in a process P is* deadlocked *if it is top-level and P can't be reduced.*

An input or output prefix in a process P is deadlock-free *if no reduction of P leads to that prefix being deadlocked.*

For example, if $\nexists Q : P \rightarrow Q$ then all top-level actions in P are deadlocked. In $!\,a(x).P \mid Q$, all a-outputs are deadlock-free. In $a.\bar{b} \mid b.\bar{a}$, both a and b are deadlocked. In $P = ?.a \mid \bar{a}$, a is deadlock-free, but $\bar{a}$ isn't ($P \rightarrow \equiv \bot.a \mid \bar{a}$ in which $\bar{a}$ is deadlocked, although $P \rightarrow \sim a \mid \bar{a}$ in which $\bar{a}$ is deadlock-free).

Deadlock-freedom is not a very interesting property on its own, because for instance $P|\Omega$ is deadlock-free as it can always be reduced.

One way would be to require all processes to terminate, but a more general approach is introduce to the following (strictly stronger) property:

Definition 3.7.2 (Livelock-freedom) *An action of a process P on a port p is* livelock-free *if it reaching top-level implies it can be consumed.*

For example, a request to a server is livelock-free is and only if it is guaranteed to be eventually received. In $!\,a(x).\bar{x} \mid \overline{a}\langle b\rangle \mid b$, the input at b is livelock-free, and in $P = !\,a(x).\bar{b}\langle x\rangle \mid !\,b(x).\overline{a}\langle x\rangle \mid \overline{a}\langle s\rangle \mid s$, the s-input is deadlock-free but not livelock-free.

This property is related to activeness in that (although either definition need to be adapted as we work in a labelled setting and TyPiCal in a reduction setting) p is livelock-free if and only if the complement port $\bar{p}$ is active.

Channel usages are a generalisation of our multiplicities, and tell for a particular channel how many times the input and output ports are used, and in what order.

Definition 3.7.3 (Channel Usages) *The* usage *of a channel is an expression given by the following grammar:*

$$U \;::=\; \mathbf{0} \;\mid\; \rho \;\mid\; u.U \;\mid\; (U|U) \;\mid\; U\&U \;\mid\; \mu\rho.U$$
$$u \;::=\; ! \;\mid\; ?$$

Usage $!.U$ does an output and then U; Usage $?.U$ does an input and then U. $(U_1|U_2)$ uses according to U_1 and U_2 in parallel. $U_1\&U_2$ uses according to either U_1 or U_2 but not both. We write $\mathrm{chan}_U(\tilde{\sigma})$ for a channel of usage U and parameters $\tilde{\sigma}$. When the context is clear, we may write just the usage for a parameterless channel.

For example, $a.b \,|\, \bar{b}.\bar{c}$ uses a according to $?$, b according to $?|!$ and c according to $!$. In $\mathbf{!}\, a(x).\bar{x}\langle 1\rangle$, a has usage $*?|!$ (with $*? \stackrel{\text{def}}{=} \mu\rho.(?.\rho)$), and thus $\mathrm{chan}_{*?|!}(!), b :!$ as a channel type (the parameter usages give the behaviour of the channel's *input* side, and here the a-input *outputs* on x). As a last example, say $a \neq t$ has usage U_1 in P and U_2 in Q. It then has usage $U_1\&U_2$ in $(\boldsymbol{\nu} t)\,(\bar{t} \,|\, t.P \,|\, t.Q)$.

Obligation and Capability levels generalise the levels used in [AB08a]:

Definition 3.7.4 (Obligation and Capability Levels) *An* obligation level *for an (input or output) primitive is a number (or ∞) telling when it will be ready to fire (i.e. at top-level), while a* capability level *tells, if that primitive is at top-level, when will it actually be consumed.*

These levels are included into usages with the syntax $u \;::=\; !^{t_O}_{t_C} \;\mid\; ?^{t_O}_{t_C}$.

For example, consider the process $a.b \,|\, \bar{b}.\bar{c}$. The input a is at top-level and thus has obligation level 0: Assuming it gets consumed at time t, b will be ready to fire at time $t+1$. The output $\bar{b}$ is immediately ready, but will actually get consumed at time $t+1$. b has capability 0 because no matter when it is brought to top-level, $\bar{b}$ will be ready to communicate with it. To sum up, we get the following: $a : (?^0_t), b : (?^{t+1}_0 | !^0_{t+1}), c : (!^{t+2}_{t'})$.

In this example, the obligation level of a port is equal to the capability level of its complement. However this is not always the case in presence of non-linearity: In $\bar{a}.x \,|\, \bar{a}.y \,|\, a.z \,|\, \bar{x}$, a has usage $(!^0_\infty | !^0_\infty | ?^0_0)$ — both $\bar{a}$ have capability zero because neither is guaranteed to succeed. Being at top-level, all a and $\bar{a}$ have obligation zero.

As expected, activeness, responsiveness, livelock-freedom, and obligation/capability levels are tightely related:

- A term is active if and only if it has a finite obligation level and all complement actions have a finite capability level.
- A term is strongly active if and only if it has a zero obligation level and all complement actions have a zero capability level.
- A term is livelock-free if and only if it has a finite capability level.
- Input (resp., output) responsiveness corresponds to finiteness of all obligation (resp., capability) levels on parameter usages.

3.7.2 Power

There is no subsumption relation either way between our system and the one implemented by TyPiCal.

On the one hand, the usage information is strictly more expressive than multiplicities (which can mostly be encoded in terms of usages, with the slight difference that usages can't express the *uniformity* inherent to ω-multiplicity). This permits for instance TyPiCal to handle locks correctly, as well as processes like $a \mid a.\bar{s} \mid \bar{a} \mid \bar{a}$ (where $\bar{s}$ is active because a's input and outputs are balanced, unlike for example b in $b \mid b.\bar{s} \mid \bar{b} \mid \bar{b} \mid \bar{b}$). Multiplicities would dismiss locks as well as that port a as plain names.

On the other hand, labelled dependencies as described in Section 2.11 permit an accurate analysis of processes such as

$$(\boldsymbol{\nu} t)\left(\bar{t} \,\middle|\, t.(\mathbf{!}\, z|\mathbf{!}\, a(x).\bar{z}.\bar{x}) \,\middle|\, t.\mathbf{!}\, a(y).\bar{y}\right)$$

which randomly picks a "slow" or a "fast" a-input. TyPical incorrectly marks the $\bar{z}$ output as unreliable (not livelock-free). Labels make z's unreliability (or non-activeness, or infinite obligation level) irrelevant when checking a's responsiveness.

It should be noted that neither our system nor TyPiCal recognises a as input active in that process, which suggests a future research direction.

Finally, TyPiCal does not handle recursive channel types that would be required to analyse processes like $\overline{a}\langle a\rangle$ or $\mathbf{!}\, a(x).\overline{x}\langle a\rangle$ but we believe it would be a rather simple extension, as was the case for our system.

3.8 Kobayashi — Type Systems for Concurrent Programs

This paper [Kob02] covers most of the theorical basis (including channel usages, capability and obligation levels) for TyPiCal, in the form of a type system being described incrementally, similarly to the present paper. The analysis given in Section 3.7 therefore remains mostly valid, except that [Kob02] works on polyadic π. It also covers tail recursive functions (similarly to [AB08a]), and a number of interesting extensions such as *session types* and *termination* analysis.

Their types don't seem to describe a separation of input and output protocols in channel types.

Our strategy of using explicit dependency networks instead of obligation (and capability) levels has the advantage of describing a process as an open system, in that it describes how the process would react when composed with an arbitrary other process. For instance, if $P = a.b$, then seeing P as a closed system implies that b will never be available. Describing it with a dependency network makes explicit in the type that b becomes active if $\bar{a}$ is.

3.9 Igarashi and Kobayashi — A generic type system for the Pi-calculus

This [IK01] is a framework for type-checking various safety properties such as deadlock-freedom or race-freedom. It works with *abstract processes* — a simplified form of the target process — and soundness theorems establishing that if the abstraction is well-behaved then so is the actual process. It is particularly

useful for *safety* properties as subject reduction is proven once and for all, so that instances of the generic type system only need to show that if the abstract process is well-behaved, the target process is not *immediately* breaking the desired property. In contrast, our type system is focused on *liveness* properties like activeness or termination so that showing the validity of a dependency analysis done on the abstract process and the correspondance between activeness in the abstract process and the actual one would likely require the same amount of work as starting from scratch.

4 Conclusion

In this paper, we proposed a notation for channel and process types in the π-calculus, as well as semantics for activeness and responsiveness.

The strong point of this notation is a higher expressiveness — not only it permits encoding the types of all papers being considered (except for non trivial channel usages), it allows for a more detailed specification of the protocol being used on a channel and capabilities being transmitted.

Dependency networks in process types accurately specify the interface of the process with the environment, so that having typed P and Q independently, their types can be composed to obtain $P|Q$'s type (unlike most works covered here, which treat processes as a whole and in a reduction-based setting and thus can't directly predict the effects of such a composition without type checking the composition itself). Similarly, the reliability built into semantics (Section 2.8) means the properties are preserved by composition, which is not always the case with such reduction-based settings. Finally, labelled dependencies permit improving the accuracy of a type checking analysis by breaking dependency transitivity where it is not relevant.

A direction for future work (after verifying and publishing an actual type system based on this paper) is typing *choice types*, to be able, for example, to express a process sending a message to exactly one of two channels, so that boolean types can be encoded ($\mathbf{!}\, b_T(tf).\bar{t}$) being the server encoding "true" and $\overline{b}(\boldsymbol{\nu} tf).(t.T{+}f.F)$ being the if-then-else construct. This in turn is important to type non-trivial encodings. With some work, our system could also probably integrate subtyping [PS93], recursion on well-founded data domains [AB08a] and channel usages [Kob02, Kob08] for improved accuracy.

References

[AB08a] L. Acciai and M. Boreale. Responsiveness in process calculi. *Theoretical Computer Science*, 409(1):59–93, 2008.

[AB08b] L. Acciai and M. Boreale. Spatial and Behavioral Types in the Pi-Calculus. In *Proceedings of the 19th international conference on Concurrency Theory*, volume LNCS 5201, pages 372–386. Springer, 2008.

[ABL03] R. M. Amadio, G. Boudol and C. Lhoussaine. The receptive distributed π-calculus. *ACM Transactions on Programming Languages and Systems*, 25(5):549–577, 2003.

[CC04] D. Cacciagrano and F. Corradini. Fairness in the pi-calculus. Technical Report, Dipartimenti di Informatica, Università di L'Aquila, 2004.

[IK01] A. Igarashi and N. Kobayashi. A generic type system for the Pi-calculus. *ACM SIGPLAN Notices*, 36(3):128–141, 2001.

[Kob02] N. Kobayashi. Type systems for concurrent programs. In *Proceedings of UNU/IIST 10th Anniversary Colloquium*, volume 2757 of *LNCS*, pages 439–453. Springer, 2002.

[Kob08] N. Kobayashi. TyPiCal 1.6.2, 2008. http://www.kb.ecei.tohoku.ac.jp/ koba/typical/.

[KPT99] N. Kobayashi, B. C. Pierce and D. N. Turner. Linearity and the Pi-Calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.

[PS93] B. C. Pierce and D. Sangiorgi. Typing and Subtyping for Mobile Processes. In *Proceedings 8th IEEE Logics in Computer Science*, pages 376–385. IEEE Computer Society, 1993.

[PT00] B. C. Pierce and D. N. Turner. Pict: A Programming Language Based on the Pi-Calculus. In G. Plotkin, C. Stirling and M. Tofte, eds, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[San99] D. Sangiorgi. The Name Discipline of Uniform Receptiveness. *Theoretical Computer Science*, 221(1–2):457–493, 1999.