A π-calculus with limited resources, garbage-collection and guarantees

David Teller

Abstract. Techniques such as mobility and distribution are often use to overcome limitations of resources such as the amount of memory or the necessity for specialized devices. However, few attempts have been made at formalizing the notion of *limited resources* in process algebras for mobility and distribution. One of our previous works[11] introduced a variant of the π-calculus with explicit allocation, garbage-collection and finalization of resources and a type system to guarantee bounds on resource usage.

In this paper, we revisit this controlled π-calculus, providing better semantics for allocation and deallocation and a better observation of resource-related behaviours and garbage-collection. We also expand the type system to provide guarantees of respect of more complex protocols on resources. We demonstrate the interest of the controlled π-calculus by building a model of a complex resources manager for concurrent systems, with manual and automatic garbage-collection, error-handling and statically distributed and transmitted authorizations and resources.

1 Introduction

In modern computer technologies, aspects of mobility, concurrency and distribution are nearly omnipresent. Software deployed on a system has often been built on a different system, possibly using a distributed Compile Farm, and channeled to the end-user through safe or unsafe means, distributed Domain Name Servers, cache hierarchies or grid supercomputers arhieve great efficiency, while consumer electronics is expected to take advantage of Cell [7] technologies in the near future to provide mostly transparent distribution of tasks among dynamically assembled arrays of processors.

While these techniques are often used for the sake of convenience, as they correspond to the intuitive way of thinking about some problems, they are also often necessary to overcome limitations on resources. For instance, tasks such as building large software projects or computing large sets of data for numerical analysis typically require amounts of memory or CPU power not found on individual computers, while in many organizations, documents are commonly sent to print servers, as client PCs do not have their own printing resource.

Several attempts [6, 8] have been made at modelling the usage of resources for mobility. However, few of these attempts take into account the fact that many resources are both limited and used concurrently. In

this study, we attempt to go further, by taking into account notions of allocation, deallocation, reallocation of previously deallocated resources and garbage-collection.

This document presents a process algebra based on the π-calculus, the *controlled π calculus*, or cπ, built upon ideas previously expressed in the previous incarnation of the controlled π-calculus [12] and in BoCa [2].

The foremost concept is that of resource. Resources are abstract representations of notions such as memory space, file access rights or printers. Resources may be allocated to entities such as variables, file handlers or print requests. The language construction corresponding to entities is the channel name. Thus, allocation and deallocation of resources to entities are modelled by creation and destruction of names.

From this conception of resources, we enrich the π-calculus with notions of resource usage, allocation and deallocation. In the controlled π-calculus, if we consider the creation of name a by the transition

$$\texttt{new}\ a\ \texttt{in}\ j \longrightarrow (\nu a)j\ ,$$

$\texttt{new}\ a\ \texttt{in}\ j$ is a process which creates a new name a and then behaves as j while process $(\nu a)j$ is a process which behaves as j but can use a new channel name a. In turn, process $(\nu a)j$ uses more resources than j as some resources have been allocated to a. Hence, the transition cannot take place if there are not enough resources available for a. Conversely, if, for any reason, the following transition takes place

$$(\nu a)j' \longrightarrow j'\ ,$$

the resources held by a have been freed and may be reused by a different process, possibly triggering transitions which were blocked by the lack of resources.

The actual instant at which a resource is deallocated depends on a notion of garbage-collection. Although it is simple to remove a name when the corresponding channel has disappeared from any term, this is not the only circumstance in which a channel is never used again. For instance, in the following term,

$$(\nu a)a(x).j\ ,$$

the process has allocated resources for a and is expecting some input on a but, since no other process knows of a, the communication will never take place. That channel is therefore useless. A well-chosen garbage-collection scheme may decide to suppress (νa) and effectively remove the whole term,

$$
\begin{array}{lll}
\text{Processes} & P, Q & ::= (\nu a : r)P \mid P|Q \mid i \\
\text{Instructions} & i, j & ::= \mathbf{0} \mid \texttt{new}\ a : r\ \texttt{in}\ i \mid \texttt{spawn}\ i\ \texttt{in}\ j \mid a(b).i \\
 & & \quad \mid \overline{a}\langle b\rangle.i \mid\ !i \mid \texttt{ifnull}\ a\ \texttt{then}\ i\ \texttt{else}\ j \\
\text{Contexts} & C[\cdot] & ::= (\nu a : r)C[\cdot] \mid C[\cdot]|P \mid P|C[\cdot] \mid [\cdot]
\end{array}
$$

FIGURE 1. Syntax of cπ.

hence releasing resources. Complete enough garbage-collection schemes may also free resources held by deadlocks or livelocks. As there are many algorithms which may produce a garbage-collector and as, as we will see, complete garbage-collection is an undecidable problem, we use a parametric relation $\vDash_{GC}$ to determine when a channel may be removed.

This enriched π-calculus is both simpler and more generic than the original cπ as well as more adapted to reasoning and programming with resources than the standard π-calculus. Well-chosen garbage-collectors permit dynamic handling of resources and exception-like error mechanisms. In turn, this allows the writing of processes which enforce resource usage policies. We complete the language by a type system, more powerful than that of the original cπ, to prove statically the respect of some such policies.

In section 2, we introduce the revisited controlled π-calculus which we illustrate in section 3 with the example of print spooler, demonstrating possible usages of the primitives and garbage-collection. We then discuss in section 4 properties of the language and its relations with the π-calculus. Section 5 contains a type system for guarantees on resource usage. We conclude this paper by a review of related works and on-going developments.

2 The language

The syntax of cπ is presented on figure 1.

As we wish to distinguish when operations take place, we differenciate currently executed processes from (P, $Q\ldots$) from instructions (i, $j\ldots$) waiting to be executed. Resources (r, $s\ \ldots$) are members of the a set of resources $\mathcal{S}$, parameter of the language. Names (a, $b\ldots$) are elements of an infinite set $\mathcal{N}$ which also contains the set of "null" names ($\circledcirc_a$, $\circledcirc_b\ldots$), special names of deallocated channels on which it is thus impossible to communicate. Process $(\nu a : r)P$ holds resources r for its private name a and behaves as P, while process i executes the instructions i.

Instruction $\texttt{new}\ a : r\ \texttt{in}\ i$ allocates a resource r resource to name a and proceeds as i, while $\texttt{spawn}\ i\ \texttt{in}\ j$ produces two concurrent threads i and j. Instruction $\texttt{ifnull}\ a\ \texttt{then}\ i\ \texttt{else}\ j$ checks whether a is $\circledcirc_{\_}$. If so, it runs i, otherwise j. The other constructions are identical to

their π-calculus counterpart. As a syntactical shortcut, we will write $\overrightarrow{a}$ for $a_1, a_2 \ldots, a_n$ and $(\nu \overrightarrow{a} : \overrightarrow{r})$ for $(\nu a_1 : r_1) \ldots (\nu a_n : r_n)$. The sets fn/bn of free/bound names are defined as in the π-calculus – note that $fn((\nu a : r)i) = fn(\texttt{new}\ a : r\ \texttt{in}\ i) = fn(i) \cup \{a\}$ and that $\circledcirc\_$ is always free.

Note that the main difference between (ν) and $\texttt{new} \cdots\ \texttt{in}$ as well as between $\mid$ and $\texttt{spawn} \cdots\ \texttt{in}$ is that the first construct of each pair describes a running process, while the second describes an instanciation. In fact, an implementation of cπ might introduce *constructors* for $\texttt{new} \cdots\ \texttt{in}$ – representing different methods of generating an entity – and for $\texttt{spawn} \cdots\ \texttt{in}$ – representing different natures and locations of instanciated processes.

Definition 1 (Set of resources). *A set of resources is a set* $(\mathcal{S}, \oplus, \bot, \preceq, \top)$ *such that* $\oplus$ *is commutative wherever it is defined, with neutral element* $\bot$, $\preceq$ *is a preorder on* $\mathcal{S}$ *with minimal element* $\bot$ *and maximal element* $\top$ *and for all* x, y, z, *whenver* $x \preceq y$ *holds,* $x \oplus z \preceq y \oplus z$ *also holds.*

When $x \oplus x \oplus \cdots \oplus x$ is defined, we note this infinite sum $\infty \cdot x$.

Definition 2 (Resources held). *A process* P *is said to hold resources* $res(P)$*, where* res *is defined by*

$$\begin{aligned} res((\nu a : r)P) &= res(P) \oplus r \\ res(P|Q) &= res(P) \oplus res(Q) \\ res(i) &= 0 \end{aligned}$$

Note that the resources held by a process may change with time.

Semantics The structural equivalence is defined as the smallest equivalence $\equiv$, compatible with the axioms of figure 2 and with α-conversion of bound names. Note that $\circledcirc\_$ is cannot be bound and is thus not α-convertible. The resource-unaware semantics are defined as the smallest relation $\longrightarrow_{pre}$ compatible with the axioms of figure 3.

$$(\nu a : r)(\nu b : s)P \equiv (\nu b : s)(\nu a : r)P \qquad !t \equiv t|!t$$

$$\frac{a \notin fv(Q)}{((\nu a : r)P)|Q \equiv (\nu a : r)(P|Q)} \qquad \frac{P \equiv Q}{P|R \equiv Q|R} \qquad \frac{P \equiv Q}{(\nu a : r)P \equiv (\nu a : r)Q}$$

Figure 2. Structural equivalence of cπ.

The semantics are very similar to those of a π-calculus with structural equivalence. Note, however, that we do not have $(\nu a : r)\mathbf{0} \equiv \mathbf{0}$, as the left

$$\text{R-Par}\ \frac{P \longrightarrow_{pre} P'}{P|Q \longrightarrow_{pre} P'|Q} \qquad \text{R-Label}\ \frac{P \xrightarrow{\alpha}_{pre} P'}{P|Q \xrightarrow{\alpha}_{pre} P'|Q}$$

$$\text{R-Comm}\ \frac{P \xrightarrow{a(b)}_{pre} P' \qquad Q \xrightarrow{\overline{a}\langle b\rangle}_{pre} Q'}{P|Q \longrightarrow_{pre} P'|Q'}$$

$$\text{R-Entity}\ \frac{P \longrightarrow_{pre} P'}{(\nu c : r)P \longrightarrow_{pre} (\nu c : r)P'}$$

$$\text{R-Hide}\ \frac{P \xrightarrow{\alpha}_{pre} P'}{(\nu a : r)P \xrightarrow{\alpha}_{pre} (\nu x : r)P'}\ a \notin \alpha$$

$$\text{R-Equiv}\ \frac{P \equiv P' \qquad Q' \equiv Q \qquad P' \longrightarrow_{pre} Q'}{P \longrightarrow_{pre} Q}$$

$$\text{R-Spawn}\quad \texttt{spawn}\ i\ \texttt{in}\ j \longrightarrow_{pre} i|j \qquad \text{R-Alloc}\quad \texttt{new}\ a : r\ \texttt{in}\ i \longrightarrow_{pre} (\nu a : r)i$$

$$\text{R-Receive}\ \frac{a \neq \circledcirc_{\_}}{a(b)i \xrightarrow{a(c)}_{pre} i\{b \leftarrow c\}} \qquad \text{R-Send}\ \frac{a \neq \circledcirc_{\_}}{\overline{a}\langle c\rangle i \xrightarrow{\overline{a}\langle c\rangle}_{pre} i}$$

$$\text{R-Null}\ \texttt{ifnull}\ \circledcirc_{\_}\ \texttt{then}\ i\ \texttt{else}\ j \longrightarrow_{pre} i$$

$$\text{R-Defined}\ \frac{a \neq \circledcirc_{\_}}{\texttt{ifnull}\ a\ \texttt{then}\ i\ \texttt{else}\ j \longrightarrow_{pre} j}$$

$$\text{Gc-Deallocate}\ \frac{\overrightarrow{a : r} \vDash_{GC} P}{(\nu\, \overrightarrow{a} : \overrightarrow{r})P \longrightarrow_{pre} P\{\overrightarrow{a} \leftarrow \overrightarrow{\circledcirc_x}\}}\ \overrightarrow{cnull_x}\ \text{fresh}$$

$$\text{Gc-Receive}\ \circledcirc_{\_}(x)i \longrightarrow_{pre} \mathbf{0} \qquad \text{Gc-Send}\ \overline{\circledcirc_{\_}}\langle x\rangle t \longrightarrow_{pre} \mathbf{0}$$

$$\text{Gc-RReceive}\ !\circledcirc_{\_}(x)t \longrightarrow_{pre} \mathbf{0} \qquad \text{Gc-RSend}\ !\overline{\circledcirc_{\_}}\langle x\rangle t \longrightarrow_{pre} \mathbf{0}$$

FIGURE 3. Resource-unaware semantics of cπ.

side holds resources r, while the right side holds no resource. As opposed to the previous incarnation of cπ, we do not have $(\nu a : r)\mathbf{0} \longrightarrow_{pre} \mathbf{0}$ either, as this is actually a step of garbage-collection with most garbage-collection schemes.

Rule Spawn executes the spawning of a new thread, while rule Channel executes the creation of a new channel, hence consuming resources. Rules Receive and Send differ from their π-calculus counterparts insofar as no communication is allowed on a destroyed channel. Processes waiting for communication on $\circledcirc\_$ are rather garbage-collected by Gc-Receive, Gc-Send, Gc-RReceive and Gc-RSend. The destruction of a channel – and the corresponding release of resources held by the – is handled by Gc-Deallocate. This rule depends on a parametric relation $\vDash_{GC}$, which determine when a set of names can be removed. Note that, as opposed to the π-calculus, a name can be removed although it still appears syntactically in the term. This allows well-chosen garbage-collectors to clean processes such as $(\nu a : r)a(b).t$ and replace them by $\mathbf{0}$, as well as more complex scenarios. We provide more complete examples in section 3.

Definition 3 (Resource-aware semantics). *The resource-aware semantics of an instance of cπ on the set of resources $\mathcal{S}$ with a resource limit of $n \in \mathcal{S}$ and with a garbage-collector $\vDash_{GC}$ is defined by the relation $\longrightarrow_n^{GC}$ where*

$$\frac{P \longrightarrow_{pre} Q \qquad res(Q) \preceq n}{P \longrightarrow_n^{GC} Q}$$

Resource-aware semantics take into account the fact that a system has limited resources. When a resource is not available for a transition, that transition does not take place. As we will see in section 3, the garbage-collector can be used to provide error-handling mechanisms in case of resource exhaustion.

3 Example: A print spooler

Let us consider a simple bounded resources manager, such as a simple print spooler – note that most of the techniques we expose here would be equally useful in the case of a system memory manager or any resource broker. This spooler maintains a pool of available printers, receives requests from clients and return a handler to any non-busy printer. Whenever the caller has finished with the printer, this resource is returned to the pool. If some requests had been put on hold, one of them is then unblocked and fulfilled.

Let us build a a cπ implementation/model of this manager, by successive iterations. For the sake of simplicity, in the course of this example,

$$
\begin{array}{ll}
Push\ p & = \overline{printer}\langle p\rangle \\
Pop\ (x).i & = printer(x).i \\
Pool & = (\nu_{chan}\ p_1 : Printer)Push\ p_1 \mid \cdots \mid (\nu_{chan}\ p_n : Printer)Push\ p_n \\
 & \\
Client & = \texttt{new}\ request\ \texttt{in}\ .\overline{alloc}\langle request\rangle. \\
 & \quad request(printer, destructor).\text{Print}.\overline{destructor}\langle\rangle.\text{Proceed}
\end{array}
$$

FIGURE 4. Building blocks of the print spooler.

we extend the syntax of cπ to support n-ary communications.

Figure 4 presents the low-level components of our spooler as well as the expected behaviour of a client. The main components are $Push$, a macro used to add (or return) a printer to the pool and Pop, a macro used to take a printer from the pool, itself represented by $Pool$. The term $Client$ is an example of a possible client: it sends a request on the predefined channel $alloc$, waits for the pair $printer, destructor$, uses $printer$ as necessary, then deallocates by calling $destructor$ and proceeds without using $printer$ anymore.

3.1 Allocation and deallocation

Figure 5 presents one possible strategy for the print spooler. Allocating a printer implies giving the client full access, while deallocating a resource through the destructor returns that printer to the pool, without any check.

$$
\begin{array}{l}
BRMDriver_1 = !alloc(request).\texttt{new}\ destructor\ \texttt{in}\ .Pop\ (x).(\\
\qquad \overline{request}\langle x, destructor\rangle \\
\qquad\quad \mid destructor().Push\ x \\
\qquad)
\end{array}
$$

Garbage-Collection $\vDash_1$: $\forall x, \forall P, \{x : \_\} \vDash_1 P \iff x \notin fv(P)$

FIGURE 5. Model of a print spooler

This deallocation strategy is unsafe, as a malicious or ill-written client may never call $destructor$, may call it several times or may continue to use the printer afterwards – this deallocation interface is therefore similar to `free`() in C.

An orthogonal problem is the garbage-collection of intermediate channels such as l and $destructor$. For this purpose, relation $\vDash_1$ permits the

destruction of names whenever they are not referenced anymore, in a manner similar to that of the traditional π-calculus. Such a mechanism is commonly found in programming languages, implemented as a reference-counter in Python, Visual Basic or C++ frameworks such as Microsoft's Com or Mozilla's XPCom. Note that this aspect of garbage-collection is orthogonal to the management of printers themselves.

$$
\begin{aligned}
BRMDriver_2 = {} & !alloc(request).\texttt{new}\ destructor\ \texttt{in}\ . \\
& \texttt{new}\ handler\ \texttt{in}\ .Pop\ (x).(\\
& \quad \mid \overline{request}\langle handler, destructor\rangle.\overline{delete}\langle buf\rangle.!handler(y).\overline{x}\langle y\rangle \\
& \quad \mid destructor().\overline{delete}\langle destructor\rangle.\overline{delete}\langle handler\rangle.Push\ x \\
&)
\end{aligned}
$$

The Garbage-Collection relation is the smallest $\vDash_2$ verifying

$$\forall x, \forall P, Q, \{a : \_\} \vDash_2 \overline{delete}\langle a\rangle.i \mid Q$$

Figure 6. A print spooler without dangling pointers

The process $BRMDriver_2$ and the garbage-collection scheme $\vDash_2$, presented on figure 6, provide a more robust management of resources. Relation $\vDash_2$ mimmicks a generic manual deallocator: any name a can be destroyed by calling $\overline{delete}\langle a\rangle$. Since the destruction is handled by the garbage-collector, the semantics of cπ guarantee that name a effectively disappears.

The manager takes advantage of this deallocator to improve safety. Instead of giving full control to the client, it transmits a (dynamically created) handler, which can be revoked at any time by calling $\overline{delete}\langle handler\rangle$. For this example, revokation only takes place when it is explicitely requested through *destructor*. The printer can then safely be put back onto the pool, without any risk of being reused by the client and without any dangling pointers.

Although this strategy makes deallocation safer, it still does not work whenever a client fails to call the destructor. As in modern programming languages, such problems can be avoided using garbage-collection and finalisation, as shown on figure 7.

Relation $\vDash_3$ defines a garbage-collection scheme, which supports a mechanism similar to reference-counting, in which names can be removed whenever they only appear in receptions or finalisations, as well as manual deallocation of handlers using *delete*. The notion of finalisation, as encountered in many garbage-collected programming languages such as

$$
\begin{array}{l}
Finalize\ x.i \quad = \texttt{new}\ loop : Loop\ \texttt{in}\ (!loop().\texttt{ifnull}\ x\ \texttt{then}\ i\ \texttt{else}\ \overline{loop}\langle\rangle \mid \overline{loop}\langle\rangle) \\
BRMDriver_3 = !alloc(request).\texttt{new}\ destructor\ \texttt{in}\ . \\
\qquad \texttt{new}\ handler : Handler\ \texttt{in}\ .Pop\ (x).(\\
\qquad\quad \mid \overline{request}\langle handler, destructor\rangle.!handler(y).\overline{x}\langle y\rangle \\
\qquad\quad \mid destructor().\mathbf{0} \\
\qquad\quad \mid Finalize\ handler.Push\ x \\
\qquad)
\end{array}
$$

The Garbage-Collection relation is the smallest $\vDash_3$ verifying

$$
\begin{array}{l}
\forall x, \forall P, x \notin fv(P) \Rightarrow \{x\} \vDash_3 P \\
\forall x, \forall P, Q, \{x\} \vDash_3 P \Rightarrow \{x\} \vDash_3 P \mid x(y).Q \\
\forall x, \forall P, Q, \{x\} \vDash_3 P \Rightarrow \{x\} \vDash_3 P \mid !x(y).Q \\
\forall x, \forall P, Q, \{x\} \vDash_3 P \Rightarrow \{x\} \vDash_3 P \mid Finalize\ x.Q \\
\forall x, \forall P, Q, \{a : Handler\} \vDash_2 \overline{delete}\langle a\rangle.i \mid Q
\end{array}
$$

FIGURE 7. A garbage-collected print spooler

Java, C# or OCaml, and as defined here by $Finalize\ x.i$, triggers a function/method/process (here, i) in response to the deallocation of an entity (here, x). Note that, as in our previous works [11], and by opposition to these languages, finalisation is safe, insofar as *resurrection* of an entity [1] is impossible. Also note that, by opposition to the first version of cπ, finalisation is a macro rather than a primitive of the language.

Term $BRMDriver_3$ takes advantage of the automatic garbage-collection and finalisation: *destructor* and *handler* are automatically destroyed, while finalisation premits returning the printer to the pool after the dallocation of *handler*. This behaviour is more robust than that of either $BRMDriver_1$ or $BRMDriver_2$ and could be rendered even more robust by more complete garbage-collectors.

3.2 Error-handling

Let us consider the following scenario: a client has acquired a printer but has started misbehaving, possibly by sending a stream of incorrect instructions to that printer. Assuming that the spooler can detect such a situation, it should stop the printing transaction and return the printer to the pool. A number of other external reasons may require stopping the printing transaction, such as lack of memory or prioritization of a specific client.

These behaviours can be modelled easily, as shown on figure 8, by modifying the garbage-collector to send signals representing the error/exception. A signal ERR is sent to represent a non-deterministic client

$$
\begin{aligned}
BRMDriver_4 = {} & !alloc(request).Pop\ (x).\texttt{new}\ destructor\ \texttt{in} \\
& \texttt{new}\ handler\ \texttt{in}\ \texttt{new}\ sigmem : MEM\ \texttt{in}\ \texttt{new}\ prio : PRIO\ \texttt{in} \\
& \texttt{new}\ err : ERR\ \texttt{in}\ \texttt{new}\ flag : Flag\ \texttt{in}\ (\\
& \quad \mid \overline{request}\langle handler, destructor\rangle.!handler(y).\overline{x}\langle y\rangle \\
& \quad \mid destructor().\mathbf{0} \\
& \quad \mid Finalize\ handler.\texttt{ifnull}\ prio\ \texttt{then}\ prioritize(c).\cdots\ \texttt{else}\ Push\ x \\
& \quad \mid Finalize\ err.\overline{delete}\langle handler\rangle \\
& \quad \mid Finalize\ prio.\overline{delete}\langle handler\rangle \\
& \quad \mid Finalize\ mem.\overline{delete}\langle handler\rangle \\
&)
\end{aligned}
$$

The Garbage-Collection relation is the smallest $\vDash_4$ verifying

$$
\begin{aligned}
& \{x\} \vDash_3 P \Rightarrow \{x\} \vDash_4 P \\
& \{x : ERR\} \vDash_4 P \text{ non-deterministically} \\
& res(P) \succeq memory\_limit \Rightarrow \{signal : MEM\} \vDash_4 P \\
& \{signal : PRIO, flag : Flag\} \vDash_4 P \mid \overline{prioritize}\langle client\rangle \mid \overline{flag}\langle\rangle
\end{aligned}
$$

FIGURE 8. A garbage-collected print spooler with signal- and error-handling

error, a signal $PRIO$ to represent a reprioritization, requested on channel *prioritize* ($flag$ serves to guarantee that only one transaction will be cancelled), and a signal MEM is triggered whenever processes use too much memory. In all three cases, the spooler destroys the handler, hence terminating the authorization of the client. If the request was a prioritization, the prioritized client receives a new handler, without going through the queue. Otherwise, the printer is returned to the pool.

The process $BRMDriver_4$ defines the responses of the spooler to these signals. From the point of view of programming languages, $Finalize\ err$, $Finalize\ prio$ and $Finalize\ mem$ are exception-handlers, comparable to `try` $\cdots$ `catch` blocks, although in a concurrent setting.

4 Behaviours and properties

4.1 Properties of the language

PROPOSITION 1 (Cπ CAN CONTAIN π).
There is a "good" encoding of the π-calculus to an instance of cπ.

We produce a simple encoding of a monadic synchronous π-calculus with structural equivalence and guarded replication, without choice, with a set of names not containing $\circledcirc\_$, to an instance of cπ with the trivial set of resources and a garbage-collector of unused names. This encoding preserves termination, reduction, structure, distribution, structural equivalence and

barbs.

Proposition 2 (More resources give more freedom).
If $\mathcal{S}$ is a set of resources and if r and s are elements of $\mathcal{S}$ such that $r \prec s$ then, for any garbage-collection scheme GC, $\longrightarrow_r^{GC} \subsetneq \longrightarrow_r^{GC}$.

The inclusion derives directly from the definition of $\longrightarrow_r^{GC}$. The non-equality can be proved by examining process $(\nu a : s)(a(x) \mid \overline{a}\langle\circledcirc\rangle)$, as this process has no reduction in $\longrightarrow_r^{GC}$ and one step of reduction in $\longrightarrow_s^{GC}$.

As in the π-calculus, we may observe behaviours of terms in cπ using barbs and simulations.

4.2 Behaviours

Definition 4 (Barbs).
If P and P' are processes such that $P \xrightarrow{x(\text{-})}_{pre} P'$ (respectively $P \xrightarrow{\overline{x}\langle\text{-}\rangle}_{pre} P'$), we say that P has a barb $x()$ (respectively $\overline{x}\langle\rangle$). Whenever P has a barb α, we write $P \downarrow_\alpha$.

Definition 5 (Weak barbed simulation).
For a resource-aware instance of cπ on the set of resources $\mathcal{S}$ and with a limit n, a relation $\mathcal{R}$ is a weak barbed simulation if, whenever $(P, Q) \in \mathcal{R}$,

- *if $P \downarrow_\alpha$, then $Q \downarrow_\alpha$*
- *if $P \longrightarrow_n^{GC} P'$ then, for some Q', $Q \longrightarrow_n^{GC*} Q'$ and $(P', Q') \in \mathcal{R}$*

If $\mathcal{R}$ is a weak barbed simulation and $P\mathcal{R}Q$, we write that Q simulates P.

Definition 6 (Weak barbed preorder).
In an instance of cpi, a relation $\mathcal{R}$ is a weak barbed preorder if, whenever $(P, Q) \in \mathcal{R}$, for all process contexts C, $C[Q]$ simulates $C[P]$. If $\mathcal{R}$ is a weak barbed preorder and $P\mathcal{R}Q$, we write $P \ll Q$.

4.3 Properties of garbage-collection

Definition 7 (Sound).
A garbage-collection scheme GC is sound if and only if, for any $\overrightarrow{a}$ and P such that $\overrightarrow{a} \vDash_{GC} P$, we have $P \ll P\{\overrightarrow{a} \leftarrow \circledcirc_a\}$.

Soundness is often a desirable property as it guarantees that the garbage-collector will not remove behaviours. Note that soundness involves a pre-order rather than an equivalence, as an equivalence would be contradictory with the use of finalizers such as defined on figure 5.

Also note that one can imagine useful non-sound garbage-collectors, for instance to model channel failures.

Definition 8 (Complete).
A garbage-collection scheme GC is complete if and only if it contains all sound garbage-collection schemes.

Proposition 3 (Perfect garbage-collection).
Sound and complete garbage-collection is undecidable.

We prove this by examining process $P = (\nu a : r)(\overline{a}\langle\circledcirc\rangle \mid a().M_b)$ where M_b encodes a Turing machine and emits a message on channel b after termination. As a sound and complete garbage-collector must decide whether $P \ll Q$, it must also decide whether M_b terminates, hence solve the halting problem.

4.4 Properties of garbage-collectors

Proposition 4 (Print spoolers).
From the garbage-collectors presented in section 3, $\vDash_1$ is sound, while $\vDash_2$, $\vDash_3$ and $\vDash_4$ are unsound. None is complete.

Soundness By definition, if $\{a\} \vDash_1 P$, a is not free in P, therefore $P\{a \leftarrow \circledcirc_a\} = P$. We also have $(\nu a : r)P \equiv P \mid (\nu a : r)\mathbf{0}$. We can prove easily that $P \mid (\nu a : r)\mathbf{0} \ll P$.

Unsoundness Let us write

$$P = (\nu handler : Handler)\overline{delete}\langle handler\rangle \mid \overline{handler}\langle a\rangle \mid handler(x).\overline{x}\langle b\rangle$$

and

$$Q = \overline{delete}\langle\circledcirc\rangle \mid \overline{\circledcirc}\langle a\rangle \mid \circledcirc\,(x).\overline{x}\langle b\rangle\ .$$

We have $\{handler : Handler\} \vDash_2 P$ and $P \longrightarrow Q$ by garbage-collection. Since $P \longrightarrow^* \downarrow_{\overline{a}\langle\rangle}$ and $Q\neg \longrightarrow^* \downarrow_{\overline{a}\langle\rangle}$, we conclude that $P\neg \ll Q$, hence $\vDash_2$ is unsound. The proof is identical for $\vDash_3$ and $\vDash_4$.

Uncompleteness None of these schemes will garbage collect $(\nu a)\overline{a}\langle b\rangle$.

Proposition 5 (Actual garbage-collection).
Informally, the Garbage-Collection of Jvm, .Net's Cli or OCaml is unsound and incomplete.

Unsoundness All three platforms have unsafe weak references, which can be dereferenced even when they point to `null`. Therefore, assuming that `weak` is a weak reference, let us consider an extract such as

- Java/Jvm

  ```
  String s = weak.get().toString();
  out.println("Action");
  ```

- C#/Cli

```
string s = weak.get().target;
Console.WriteLine("Action");
```

- OCaml

```
match Weak.get weak 0 with
  Some x -> print\_endline "Action";;
```

If the garbage-collector has removed the object referenced by `ref`, a null-pointer or match-failure exception will prevent the observable output `"Action"` from being performed.

INCOMPLETENESS As garbage-collection relies purely on the analysis of stack and heap, in the following example, the value of `s` is never recovered:

```
boolean value = true;
final String s = "useless";
while(value) ;
System.out.println(s);
```

5 A type system for resource guarantees

5.1 The system

The semantics of cπ are parametrized on a notion of resources. The mechanism of parametric garbage-collection combined with the use of terms such as *Finalize* permit to write systems which take into account allocation of resources as well as deallocations. We now introduce a type system to provide guarantees on the usage of such resources.

$$\begin{array}{lll} T & ::= Bound(t,\lambda) & r \in \mathcal{S}, \lambda : \mathcal{N} \longrightarrow r \\ N & ::= Name(C,r) & e \in \mathcal{S} \\ C & ::= Chan(N,g,\lambda) & g \in \mathcal{S}, \lambda : \mathcal{N} \longrightarrow r \\ & \mid \ Ssh & \end{array}$$

Judgement $\Gamma \vdash P : Bound(t,\lambda)$ states that, under environment Γ, P can be evaluated as a process which may be executed fully using no other resource than t and may have reused resources of external entities as specified by λ. In particular, if $\lambda(a) = l_a$, after the deallocation of a, P may reuse at most l_a of the resources originally allocated to a. The judgement $\Gamma \vdash a : Name(C,r)$ states that, according to Γ, a is the name of an entity using resource r, with role C. If C is $Chan(N,g,\lambda)$, a is a communication channel, which can be used to communicate names of type N, to transfer resource g from the sender to the receiver, some of which may be deallocated and reused as per λ. Conversely, if C is Ssh, a is not a channel and cannot be used for communication.

Figure 9 presents the rules of this type system. For the sake of readability, we slighly alter the syntax to allow writing `new` $a : N$ `in` $\cdots$ and $(\nu a : N)$. When necessary, we will write 0_λ for the function defined on $\mathcal{N}$ whose value is uniformly $\bot$ and $a \mapsto r$ for the function defined on $\mathcal{N}$ whose value is r for a and $\bot$ for everything else.

Properties

Lemma 1 (Weakening).
If Γ is an environment and P a process such that $\Gamma \vdash P : Bound(t, \lambda)$, then, for any $t' \succeq t$ and any $\lambda' \succeq \lambda$, we have $\Gamma \vdash P : Bound(t', \lambda')$.

The proof of this lemma is trivial, as each rule of the type system allows growing t and λ.

Theorem 1 (Subject Reduction).
If P is a process, if $\Gamma \vdash P : Bound(r, \lambda)$ and $P \longrightarrow^* P'$ then there is a r' and a λ' such that $\Gamma \vdash Bound(r', \lambda')$ and $r \oplus \Sigma_{x \in \mathcal{N}} \lambda(x) \succeq r' \oplus \Sigma_{x \in \mathcal{N}} \lambda'(x)$.

To understand this, let us first consider the case where $\lambda = 0_\lambda$. This case corresponds to a system closed as far as resource deallocation is concerned, as it does not reuse resources held by free names. In this case, the property becomes $r' \preceq r$: the guaranteed bound on resources cannot increase.

The more general case where λ is not necessarily 0_λ also covers transitory states between the deallocation of a name and the reuse of resources previously held by that name.

The proof is detailed in the annex.

Theorem 2 (Resource control).
If $\mathcal{S}$ is a set of resources, if GC is a garbage-collection scheme, if P is a process, if $P \longrightarrow_\top^{GC*} P'$ and if $\Gamma \vdash P : Bound(r, 0_\lambda)$ then, for all $r' \succeq r$, we also have $P \longrightarrow_{r'}^{GC*} P'$.

The proof (detailed in the annex) is straightforward.

5.2 Applications

Proposition 6 (Finalization). *We can always derive*

$$\frac{\Gamma \vdash i : Bound(r \oplus r_n, \lambda)}{\Gamma \vdash Finalize\ x.i : Bound(r', \lambda')}\ r' \succeq r, \lambda' \succeq \lambda \oplus x \mapsto r_n$$

The proof (detailed in the annex) is straightforward. It is sufficient to declare *loop* to be of type $Name(Chan(\_, r', \lambda'), \bot)$. Note that, in this macro, name *loop* serves as a loop-invariant stating resources used by the loop and its continuation.

$$\frac{\Gamma(a) = N}{\Gamma \vdash x : N} \qquad \Gamma \vdash \circledcirc_{\_} : N \qquad \Gamma \vdash \mathbf{0} : T$$

$$\frac{\begin{array}{c}\Gamma \vdash P : \mathit{Bound}(t_P, \lambda) \\ t' \succeq r_a \oplus t_P \qquad \lambda(a) \preceq r_a \qquad \forall x \neq a, \lambda'(x) \succeq \lambda(x)\end{array}}{\Gamma \vdash (\nu a : \mathit{Name}(\_, r_a))P : \mathit{Bound}(t', \lambda')}$$

$$\frac{\begin{array}{c}\Gamma \vdash P : \mathit{Bound}(t_P, \lambda_P) \qquad \Gamma \vdash Q : \mathit{Bound}(t_Q, \lambda_Q) \\ t' \succeq t_P \oplus t_Q \qquad \lambda' \succeq \lambda_P \oplus \lambda_Q\end{array}}{\Gamma \vdash P|Q : \mathit{Bound}(t', \lambda')}$$

$$\text{T-IfNull}\ \frac{\begin{array}{c}\Gamma \vdash i : \mathit{Bound}(t \oplus r, \lambda_i) \qquad \Gamma \vdash j : \mathit{Bound}(t, \lambda_j) \\ t' \succeq t \qquad \lambda' \succeq \lambda_i \oplus (x \mapsto r) \qquad \lambda' \succeq \lambda_j\end{array}}{\Gamma \vdash \texttt{ifnull}\ x : r\ \texttt{then}\ i\ \texttt{else}\ j : \mathit{Bound}(t', \lambda')}$$

$$\frac{\begin{array}{c}\Gamma \vdash a : \mathit{Name}(\mathit{Chan}(N, r, \lambda_a), \_) \qquad \Gamma, b : N \vdash i : \mathit{Bound}(t \oplus r, \lambda \oplus \lambda_a) \\ t' \succeq t \qquad \lambda' \succeq \lambda\end{array}}{\Gamma \vdash a(b).i : \mathit{Bound}(t', \lambda')}$$

$$\frac{\begin{array}{c}\Gamma \vdash a : \mathit{Name}(\mathit{Chan}(N, r, \lambda_a), \_) \qquad \Gamma \vdash i : \mathit{Bound}(t, \lambda) \qquad \Gamma \vdash b : N \\ t' \succeq t \oplus r \qquad \lambda' \succeq \lambda \oplus \lambda_a\end{array}}{\Gamma \vdash \overline{a}\langle b \rangle.i : \mathit{Bound}(t', \lambda')}$$

$$\frac{\begin{array}{c}\Gamma \vdash i : \mathit{Bound}(t_i, \lambda_i) \qquad \Gamma \vdash j : \mathit{Bound}(t_j, \lambda_j) \\ t' \succeq t_i \oplus t_j \qquad \lambda' \succeq \lambda_i \oplus \lambda_j\end{array}}{\Gamma \vdash \texttt{spawn}\ i\ \texttt{in}\ j : \mathit{Bound}(t', \lambda')}$$

$$\frac{\begin{array}{c}\Gamma \vdash i : \mathit{Bound}(t_i)[\lambda] \\ \lambda(a) \preceq r_a \qquad t' \succeq r_a \oplus t_i \qquad \forall x \neq a, \lambda'(x) \succeq \lambda(x)\end{array}}{\Gamma \vdash \texttt{new}\ a : \mathit{Name}(\_, r_a)\ \texttt{in}\ i : \mathit{Bound}(t', \lambda')}$$

$$\frac{\Gamma \vdash i : \mathit{Bound}(t, \lambda) \qquad t' \succeq \infty \cdot t \qquad \lambda' \succeq \infty \cdot \lambda}{\Gamma \vdash !i : \mathit{Bound}(t', \lambda')}$$

FIGURE 9. Type system for resource guarantees

Proposition 7 (Print spooler). *Typing the print spooler permits us to determine the following properties:*

- *The spooler uses at most n printers.*
- *Each incoming call causes the allocation of at most one handler.*
- *There can be at most n handlers running at any time.*
- *The spooler allocates handlers only on demand.*
- *The spooler sends messages to the printer only when requested to do so by a client.*

The main idea is to use the set of resources $\mathbf{N}^4$ where $\Gamma \vdash P : \mathit{Bound}((p, h, k, m), \lambda)$ means that P uses resources to allocate at most p printers, h handlers, k handlers and m messages. For this example, we use both h and k to count handlers, respectively from the point of view of the client and from that of the spooler – creating a handler uses resource $(0, 1, 1, 0)$.

Channel *alloc* serves to transfer resource $(0, 1, 0, 0)$ from the client to the spooler, while channel *handler* serves to transfer resource $(0, 0, 0, 1)$ from the client to the spooler and each printing channel $p_1, \cdots, p_n$ serves to transfer resource $(0, 0, 0, 1)$ from the spooler to the printer. Channel *printer* transfers one resource $(0, 0, 1, 0)$ from the pool to the spooler, for allocation to a handler.

It is thus sufficient to check that

$$\Gamma \vdash \mathit{BRMDriver}_4 \mid \mathit{Pool} : \mathit{Bound}((n, 0, n, 0), 0_\lambda)$$

to prove the proposition. Conversely, a client will have type

$$\mathit{Bound}((0, h, 0, m), 0_\lambda)$$

if it requests at most h printers/handlers and sends at most m messages. Depending on the actual type of *request*, h can measure either the total number of handlers allocated during the execution of the client or the maximal number of handlers held at any time by the client, assuming that the client uses finalization to recover the resources held by the handler. By using a slightly more complicated set of resources, it is possible to measure both properties at once

The typing derications themselves are long but straightforward.

5.3 Extending the type system

This version of the type system permits transferring resources from an agent to another using a communication channel. This situation, however, fails to take into account the fact that a process may charge for some

$$C ::= Chan(N, g, \lambda_g, p, \lambda_p)\ g, p \in \mathcal{S}, \lambda_g, \lambda_p : \mathcal{N} \longrightarrow r$$

$$\frac{\begin{array}{c}\Gamma \vdash a : Name(Chan(N, g, \lambda_g, p, \lambda_p), \_) \\ \Gamma, b : N \vdash i : Bound(t_i \oplus g, \lambda_i \oplus \lambda_g) \\ t'_i \succeq t_i \oplus p \qquad \lambda'_i \succeq \lambda_i \oplus \lambda_p\end{array}}{\Gamma \vdash a(b).i : Bound(t'_i, \lambda'_i)}\ \text{T-Rcv-Exchange}$$

$$\text{T-Snd-Exchange}\ \frac{\begin{array}{c}\Gamma \vdash a : Name(Chan(N, g, \lambda_g, p, \lambda_p), \_) \\ \Gamma \vdash j : Bound(t_j \oplus p, \lambda_j \oplus \lambda_p) \qquad \Gamma \vdash b : N \\ t'_j \succeq t_j \oplus g \qquad \lambda'_j \succeq \lambda_j \oplus \lambda_g\end{array}}{\Gamma \vdash \overline{a}\langle b\rangle.j : Bound(t'_j, \lambda'_j)}$$

Figure 10. A type system modified to permit resource exchanges.

resources. In this case, the transfer of a resource from process P_1 to process P_2 may require the simultaneous transfer of resources from P_2 to P_1.

Figure 10 presents the modifications to the type system. In this extended version, the resource type of communication channels is symmetrical. Note that the properties of weakening, subject reduction and resource control still hold after this modification.

6 Conclusions

Summary We have presented the new controlled π-calculus, a π-calculus altered to take into account allocation of resources, resource-bounded execution and garbage-collection. We have shown how to use this calculus to represent a complex bounded resources manager – in this case, a print spooler – with both manual and automatic garbage-collection and several forms of error-handling, including reaction to the absence of some resources. Furthermore, we have developped a rich type system to guarantee properties of resource management in this controlled π-calculus, including a novel manner of statically typing exchange of resources. Applying this type system to the bounded resources manager, we have managed to extract properties of resource-boundedness and static transfer of authorizations.

This work builds on our previous experiences [14, 10, 12] with resource control for concurrent and distributed systems. In relation to these developments, it improves our definition of garbage-collection and of resource-related simulations, it removes the necessity of deallocators built-in the

language, it starts to deal with error-handling and it adds the notions of transfer of resources.

RELATED WORKS Other approaches of resource management have been proposed. The BoCa [2] calculus is a variant of Mobile Ambients with a notion of resources which can be dynamically transferred, acquired or released. Our notion of resources held during the execution of a process, in particular, is close to the corresponding notion of weight of a process in that language, although that notion is part of the well-formedness of a BoCa term and is central to the semantics of the calculus.

The Mobile Resource Guarantees [6] project builds on a linear type system to provide guarantees of safe memory deallocation and reuse as well as memory bounds in a single-threaded ML dialect. The Vault project [3] uses in a multithreaded yet safe subset of C and a complex type system to guarantee that resources are in a correct state whenever they are used. TyPiCal [8] has comparable aims with the π-calculus. None of these works, however, takes into account garbage-collection.

Several other, mostly dynamic, solutions have been offered, from Guardians for Mobile Ambients [4] to JML or Spec#'s design-by-contract. These works, however, fail to provide static guarantees, behavioral observation of resources or to take into account deallocation and reuse.

FUTURE DEVELOPMENTS As we mentioned, instructions such as `spawn` $\cdots$ `in` $\cdots$ and `new` $\cdots$ `in` $\cdots$ instanciate processes or resources and, in an implementation of cπ, would be accompanied by constructors. Although we have not dealt with constructors for processes, a number of processes such as the print spooler can be seen as constructors for resources, which brings a number of question – firstly, if it is possible to write a constructor in cπ, how such a constructor should be defined, invoked, and what properties it should have.

Closely related is the question of transformation and composition of resources. While some resources, such as hard drive space and perhaps some authorizations, can be composed into bigger resources, and while we can take this into account at the level of typing, at the level of the language, we have no way of express such behaviour. Similarely, while some resources can be transformed by operations – such as a *file* becoming an *opened file*, our definition of resources is insufficient to model this.

We have started working on all these problems. Preliminary results seem to indicate that the controlled π-calculus and its type system may be adapted to take into account constructors, composition and transformations and to provide static guarantees based on the state of resources.

We have also started to investigate whether the notion of static re-

source exchange could be generalized to more than two participants, perhaps using some form of n-ary communication as seen in the Join-Calculus [5] or in the Kell-Calculus [9].

Garbage-collection schemes raise another series of questions. As we have seen, our definitions of soundness and completeness of a garbage-collector are too restrictive for common garbage-collectors such as those found in Java, C# or OCaml. We thus hope to better criteria to classify such services.

More importantly, we have observed that nearly all the garbage-collection schemes we have been using in our examples, both in this document and during our research, could be classified as simple cases of pattern-matching. We wonder whether this observation can be generalized and if a "useful" set of garbage-collectors can be easily defined. In particular, we have attempted to define stack-based as well as regions-based techniques as instances of $\vDash$ and preliminary results lead us to believe in the feasibility of the task.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1998.

[2] F. Barbanera, M. Bugliesi, M. Dezani, and V. Sassone. A calculus of bounded capacities. In *Proceedings of Advances in Computing Science, 9th Asian Computing Science Conference, ASIAN'03*, volume 2896 of *Lecture Notes in Computer Science*. Springer, 2003.

[3] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *SIGPLAN Conference on Programming Language Design and Implementation*, 2001.

[4] G. Ferrari, E. Moggi, and R. Pugliese. Guardians for ambient-based monitoring. In V. Sassone, editor, *F-WAN: Foundations of Wide Area Network Computing*, number 66 in ENTCS. Elsevier Science, 2002.

[5] C. Fournet and G. Gonthier. The reflexive cham and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 1996.

[6] M. Hofmann. A type system for bounded space and functional in-place update–extended abstract. *Nordic Journal of Computing*, 7(4), Autumn 2000. An earlier version appeared in ESOP2000.

[7] H. P. Hofstee. Power efficient processor architecture and the cell processor. In *HPCA*, pages 258–262. IEEE Computer Society, 2005.

[8] N. Kobayashi. TyPiCal: Type-based static analyzer for the pi-calculus.

[9] J.-B. Stefani. A calculus of kells. In V. Sassone, editor, *Electronic Notes in Theoretical Computer Science*, volume 85. Elsevier, 2003.

[10] D. Teller. Formalisms for mobile resource control. In *Proceedings of FGC'03*, volume 85 of *ENCS*. Elsevier, 2003.

[11] D. Teller. Resource recovery in the π-calculus. In *Proceedings of the 3rd IFIP*

International Conference on Theoretical Computer Science, 2004. tbp.

[12] D. Teller. Resource recovery in the π-calculus. In *Proceedings of the 3rd IFIP International Conference on Theoretical Computer Science*, 2004. tbp.

[13] D. Teller. *Ressources limitées pour la mobilité: Utilisation, réutilisation, garanties*. PhD thesis, École Doctorale MathIF, 2004.

[14] D. Teller, P. Zimmer, and D. Hirschkoff. Using Ambients to Control Resources. In *Proceedings of the 13th International Conference on Concurrency Theory*, volume 2421 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

A Encoding π in cπ (theorem 1)

We use a monadic synchronous π-calculus with structural equivalence (rather than structural congruence) and guarded replication and we assume that the set of names does not contain $\circledcirc\_$. The set of resources is the singleton $\{\bot\}$ with trivial rules.

We define our encoding $[\![\ ]\!]_p$ by the following equations

$$\begin{array}{ll}
[\![\mathbf{0}]\!]_p & = \mathbf{0} \\
[\![P|Q]\!]_p & = [\![P]\!]_p | \, [\![Q]\!]_p \\
[\![(\nu a)P]\!]_p & = (\nu a : \bot) \, [\![P]\!]_p \\
[\![!P]\!]_p & =! \, [\![P]\!]_t \\
[\![\alpha.P]\!]_p & = \alpha. \, [\![P]\!]_t
\end{array}$$

$$\begin{array}{ll}
[\![\mathbf{0}]\!]_t & = \mathbf{0} \\
[\![P|Q]\!]_t & = \texttt{spawn } [\![P]\!]_t \texttt{ in } . \, [\![Q]\!]_t \\
[\![(\nu a)P]\!]_t & = (\nu a : \bot) \, [\![P]\!]_t \\
[\![!P]\!]_t & =! \, [\![P]\!]_t \\
[\![\alpha.P]\!]_t & = \alpha. \, [\![P]\!]_t
\end{array}$$

Lemma 2 (Substitutions). *For any P, x and y, $[\![P\{x \leftarrow y\}]\!]\_ = [\![P]\!]\_\{x \leftarrow y\}$.*

Trivial.

Lemma 3 (Threads to processes). *For any P, either $[\![P]\!]_t = [\![P]\!]_p$ or $[\![P]\!]_t \longrightarrow [\![P]\!]_p$.*

Trivial.

Lemma 4 (Structural equivalence). *For all P and Q, if $P \equiv Q$ then $[\![P]\!]_p \equiv [\![Q]\!]_p$.*

We prove this by induction on the structure of a proof of $P \equiv Q$. The only subtleties are

Replication If $P =!\alpha.R$ and $Q =!\alpha.R|R$, we have $[\![P]\!]_p =! \, [\![\alpha.R]\!]_t$ and $! \, [\![\alpha.R]\!]_t \equiv! \, [\![\alpha.R]\!]_t | \alpha. \, [\![R]\!]_t$. By definition of the encoding, $\alpha. \, [\![R]\!]_t = [\![\alpha.R]\!]_p$. Hence, $[\![P]\!]_p \equiv [\![Q]\!]_p$.

GARBAGE-COLLECTION If $P = (\nu a)\mathbf{0}$ and $Q = \mathbf{0}$ then $[\![P]\!]_p = [\![Q]\!]_p$.

LEMMA 5 (NULL VALUES). *For any P and p such that $[\![P]\!]_p \longrightarrow *p$, p does not contain any occurrence of $\circledcirc\_$.*

Trivial.

PROPOSITION 8 (SOUNDNESS). *For all processes P and Q of the π-calculus such that $P \longrightarrow Q$, we have $[\![P]\!]_p \longrightarrow [\![Q]\!]_p$.*

We prove this by induction on the structure of a proof of $P \longrightarrow Q$.

COMMUNICATION If $P = a(x).R \mid \overline{a}\langle y\rangle.S$ and $Q = R\{x \leftarrow y\} \mid S$, we have $[\![P]\!]_p = a(x).[\![R]\!]_t \mid \overline{a}\langle y\rangle.[\![S]\!]_t$ and $[\![Q]\!]_p = [\![R\{x \leftarrow y\}]\!]_t \mid [\![S]\!]_t$.

By one communication and the substitution lemma, we have $[\![P]\!]_p \longrightarrow [\![Q]\!]_q$.

STRUCTURAL EQUIVALENCE If $P \equiv P'$, $P' \longrightarrow Q'$ and $Q' \equiv Q$, by induction hypothesis, $[\![P']\!]_p \longrightarrow [\![Q']\!]_p$. By lemma 4, we conclude the case.

PROPOSITION 9 (COMPLETENESS). *For every process P of the π-calculus, if $[\![P]\!]_p \longrightarrow q$ then there exists Q in the π-calculus such that $P \longrightarrow Q$ and $q = [\![Q]\!]_p$.*

By induction on a proof of $[\![P]\!]_p \longrightarrow q$. By 5, we know that there is no occurrence of $\circledcirc\_$ hence no use of the GC-* transitions.

B Subject Reduction of the type system

B.1 Auxiliary lemma

LEMMA 6 (SUBSTITUTION PRESERVES TYPING). *If $\Gamma, x : N \vdash P : Bound(t, \lambda)$ and $\Gamma \vdash a : N$ then $\Gamma \vdash P\{x \leftarrow a\} : Bound(t, \lambda^{x \leftarrow a})$ where*

- $\forall y \notin \{x, a\}, \lambda^{x \leftarrow a}(y) = \lambda(y)$
- $\lambda^{x \leftarrow a}(x) = \bot$
- $\lambda^{x \leftarrow a}(a) = \lambda(a) \oplus \lambda(x)$

This is proved by structural induction.
The only trick case is that of $P = \texttt{ifnull}\ x : r_n\ \texttt{then}\ i\ \texttt{else}\ j$.
Let us write

$$\begin{cases} \Gamma, x : N \vdash i : Bound(t \oplus r, \lambda_i) \\ \Gamma, x : N \vdash j : Bound(t, \lambda_j) \\ \qquad\qquad t' \succeq t \\ \qquad\qquad \lambda' \succeq \lambda_i \oplus (x \mapsto r) \\ \qquad\qquad \lambda' \succeq \lambda_j \\ \Gamma, x : N \vdash \texttt{ifnull}\ x : r\ \texttt{then}\ i\ \texttt{else}\ j : Bound(t', \lambda') \end{cases}$$

By induction hypothesis, we also have

$$\begin{cases} \Gamma \vdash i\{x \leftarrow a\} : \mathit{Bound}(t \oplus r, \lambda_i^{x \leftarrow a}) \\ \Gamma \vdash j\{x \leftarrow a\} : \mathit{Bound}(t, \lambda_j^{x \leftarrow a}) \end{cases}$$

Let us prove that the relation between λ', λ_i and λ_j still holds after substitution. Let us write $\Lambda = \lambda_i \oplus (x \mapsto r)$.

For any z distinct of x and a, we have

$$\lambda'^{x \leftarrow a}(z) = \lambda'(z) \succeq \lambda_j(z) = \lambda_j^{x \leftarrow a}$$

and

$$\lambda'^{x \leftarrow a}(z) = \lambda'(z) \succeq \Lambda(z) = \Lambda^{x \leftarrow a}(z) \ .$$

We also have

$$\lambda'^{x \leftarrow a}(x) = \lambda_j^{x \leftarrow a}(x) = \Lambda^{x \leftarrow a}(x) = \bot \ .$$

Also,

$$\lambda'^{x \leftarrow a}(a) = \lambda'(x) \oplus \lambda'(a) \succeq \lambda_j(x) \oplus \lambda_j(a) = \lambda_j^{x \leftarrow a}(a)$$

and

$$\lambda'(x) \oplus \lambda'(a) \succeq \Lambda(x) \oplus \Lambda(a) = \lambda_i(x) \oplus \lambda_i(a) \oplus r_n = \Lambda^{x \leftarrow a}(a) \ .$$

Hence, the relations still hold. From T-IfNull, since $(\texttt{ifnull}\ x\ \texttt{then}\ i\ \texttt{else}\ j)\{x \leftarrow a\} = \texttt{ifnull}\ a\ \texttt{then}\ i\{x \leftarrow a\}\ \texttt{else}\ j\{x \leftarrow a\}$, we conclude the case.

Lemma 7 (Equivalence preserves typing). *If $P \equiv Q$ and $\Gamma \vdash P : T$ then $\Gamma \vdash Q : T$.*

Proof by a simple induction. The α-equivalence derives from the substitution lemma (lemma 6).

B.2 Main proof

Proved by induction on the structure of a proof of $P \longrightarrow Q$.

Let us define the weight of T by $\mathit{weight}(\mathit{Bound}(t, \lambda)) = t \oplus \Sigma_{x \in \mathcal{N}} \lambda(x)$.

Instanciation Case R-Spawn is trivial as the typing rule for $\texttt{spawn}\ i\ \texttt{in}\ j$ is identical to that of $i \mid j$.

Case R-Alloc is trivial as the typing rule for $\texttt{new}\ a : N\ \texttt{in}\ i$ is identical to that of $(\nu a : N)i$.

Tests Let us write $P = \texttt{ifnull}\ a : r_n\ \texttt{then}\ i\ \texttt{else}\ j$. If x is not $\circledcirc_{\_}$ then, as the type of j is stronger than the type of j, the case is proved.

If $x = \circledcirc_{\_}$, let us write

$$\begin{cases} \Gamma \vdash & i : \mathit{Bound}(t \oplus r, \lambda_i) \\ & t' \succeq t \\ & \lambda' \succeq \lambda_i \oplus (x \mapsto r) \\ \Gamma, x \vdash & P : \mathit{Bound}(t', \lambda') \end{cases}$$

Trivially, we have $weight(Bound(t', \lambda')) \succeq Bound(t \oplus r, \lambda_i)$. Which proves the case.

COMMUNICATION Let us write

$$\begin{cases} \Gamma, x : N \vdash & i : Bound(t \oplus r, \lambda \oplus \lambda_a) \\ \Gamma \vdash & a : Name(Chan(N, r, \lambda_a), \_) \\ \Gamma \vdash & j : Bound(t_j, \lambda_j) \\ \Gamma \vdash & b : N \end{cases}$$

Typage de P		
Typage de $a(x).i$		
$\Gamma, x : N \vdash i :$	$Bound(t \oplus r, \lambda \oplus \lambda_a)$	Par hypothse
$\Gamma \vdash a :$	$Name(Chan(N, r, \lambda_a), \_)$	Par hypothse
$\Rightarrow \Gamma \vdash a(x).i :$	$Bound(t_1, \lambda_1)$	Par T-RCV
Avec	$t_1 \succeq t$	
$\lambda_1 \succeq \lambda$		

Typage de $\overline{a}\langle b \rangle.j$		
$\Gamma \vdash j :$	$Bound(t_j, \lambda_j)$	Par hypothse
$\Gamma \vdash a :$	$Name(Chan(N, r, \lambda_a), \_)$	Par hypothse
$\Gamma \vdash b :$	N	Par hypothse
$\Rightarrow \Gamma \vdash \overline{a}\langle b \rangle.j :$	$Bound(t_2, \lambda_2)$	Par T-SND
Avec	$t_2 \succeq t_j \oplus r$	
$\lambda_2 \succeq \lambda_j \oplus \lambda_a$		

Typage de P		
$\Gamma \vdash a(x).i :$	$Bound(t_1, \lambda_1)$	Cf. plus haut
$\Gamma \vdash :$	$Bound(t_2, \lambda_2)$	Cf. plus haut
$\Rightarrow \Gamma \vdash P :$	$Bound(t_3, \lambda_3)$	Par T-PAR
Avec	$t_3 \succeq t_1 \oplus t_2$	
$\lambda_3 \succeq \lambda_1 \oplus \lambda_2$		
$\bigcirc$		

Typage de Q		
Typage de $i\{x \leftarrow b\}$		
$\Gamma, x : N \vdash i\{x \leftarrow b\} :$	$Bound(t \oplus r, \lambda \oplus \lambda_a)$	Par hypothse
$\Rightarrow \Gamma \vdash i :$	$Bound(t \oplus r, \lambda \oplus \lambda_a)$	Par *Substitution*

Typage de Q		
$\Gamma \vdash i\{x \leftarrow b\} :$	$Bound(t \oplus r, \lambda \oplus \lambda_a)$	Cf. plus haut
$\Gamma \vdash j :$	$Bound(t_j, \lambda_j)$	Par hypothse
Comme	$t_3 \oplus t_1 \oplus t_2$	

$$
\begin{array}{llll}
 & t_2 \succeq t_j \oplus r & & \\
 & t_1 \succeq t & & \\
 & \quad \text{Comme} & \lambda_3 \succeq \lambda_1 \oplus \lambda_2 & \\
 & \lambda_1 \succeq \lambda & & \\
 & \lambda_2 \succeq \lambda_j \oplus \lambda_a & & \\
\Rightarrow \Gamma \vdash Q : & & \mathit{Bound}(t_3, \lambda_3) & \text{Par T-Par}
\end{array}
$$

$\bigcirc$

The case is proved.

Structure Proof of the various structural cases are identical to the corresponding proofs in our previous works [13].

Garbage-collection Cases GC-Receive, GC-Send, GC-RReceive and GC-RSend are trivial as Q is $\mathbf{0}$, which can always be typed, with any type.

Case GC-Deallocate derives directly from the substitution lemma (lemma 6).

The induction is thus proved. Hence the subject-reduction property.

C Resource control

Lemma 8 (Resource total). *If* $\Gamma \vdash P : \mathit{Bound}(r, 0_\lambda)$ *then* $\mathit{res}(P) \preceq r$.

Trivial.

C.1 Main proof

From the Resource total lemma and subject-reduction, we conclude the resource control theorem.

D Typing finalization

We have

$$
\begin{cases}
\mathit{Loop} = \mathit{Name}(\mathit{Chan}(\_, r', \lambda'), \bot) \\
\Gamma \vdash \quad i : \mathit{Bound}(r \oplus r_n, \lambda) \\
\qquad\quad r' \succeq r \\
\qquad\quad \lambda' \succeq \lambda \oplus x \mapsto r_n
\end{cases}
$$

Typage de $\overline{loop}\langle\rangle$

$\Gamma \vdash \mathbf{0}$	$: Bound(\bot, 0_\lambda)$	Par *T-Nil*
$\Gamma \vdash loop$	$: Name(Chan(\_, r', \lambda'), \_)$	Par hypothse
$\Rightarrow \Gamma \vdash \overline{loop}\langle\rangle$	$: Bound(r', \lambda')$	Par T-Snd

Typage de `ifnull` x `then` i `else` $\overline{loop}\langle\rangle$

$\Gamma \vdash \overline{loop}\langle\rangle$	$: Bound(r', \lambda')$	Cf. plus haut
$\Gamma \vdash i$	$: Bound(r' \oplus r_n, \lambda)$	Par hypothse
$\Rightarrow \Gamma \vdash$ `ifnull` x `then` i `else` $\overline{loop}\langle\rangle$	$: Bound(r', \lambda')$	Par T-Test-Nil

Typage de $loop()$.`ifnull` x `then` i `else` $\overline{loop}\langle\rangle$

$\Gamma \vdash$ `ifnull` x `then` i `else` $\overline{loop}\langle\rangle$	$: Bound(r', \lambda')$	Cf. plus haut
$\Gamma \vdash loop$	$: Name(Chan(\_, r', \lambda'), \_)$	Par hypothse
$\Rightarrow \Gamma \vdash loop().\cdots$	$: Bound(\bot, 0_\lambda)$	Par T-Rcv

Typage de $!loop()$.`ifnull` x `then` i `else` $\overline{loop}\langle\rangle$

$\Gamma \vdash loop().\cdots$	$: Bound(\bot, 0_\lambda)$	Cf. plus haut
$\Rightarrow \Gamma \vdash !loop().\cdots$	$: Bound(\bot, 0_\lambda)$	Par T-Bang

Typage de $!loop()$.`ifnull` x `then` i `else` $\overline{loop}\langle\rangle \mid \overline{loop}\langle\rangle$

$\Gamma \vdash !loop().\cdots$	$: Bound(\bot, 0_\lambda)$	Cf. plus haut
$\Gamma \vdash \overline{loop}\langle\rangle$	$: Bound(r', \lambda')$	Cf. plus haut
$\Rightarrow \Gamma \vdash !loop()\cdots \mid \overline{loop}\langle\rangle$	$: Bound(r', \lambda')$	Par T-Par

Typage de $Finalize\ x.i$

$\Gamma \vdash !loop()\cdots \mid \overline{loop}\langle\rangle$	$: Bound(r', \lambda')$	Cf. plus haut
$\Rightarrow \Gamma \vdash (\nu loop : Loop)(\cdots)$	$: Bound(r', \lambda')$	

$\bigcirc$