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

David Teller

Abstract.

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

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

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 $\square_{GC}$ 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 byano1(amic)-388hmolRot isoli1eroved.

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

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

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

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

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

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

Push p

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.

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

The Garbage-Collection relation is the smallest $□_2$ verifying

$$x,\ \ P, Q, \{a : \_\}\ □_2\ \overline{delete}\ a\ .i \mid Q$$

Figure 6. A print spooler without dangling pointers

The process $BRMDriver_2$ and the garbage-collection scheme $□_2$, presented on figure 6, provide a more robust management of resources. Relation $□_2$ mimmicks a generic manual deallocator: any name a can be destroyed by calling $\overline{delete}\ a$. Since the destruction is handled by the garbage-collector, the semantics of cπ guarantee that name a e□ectively 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 $\overline{delete}\ handler$. 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 $□_3$ 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

$Finalize\ x.i\ = \texttt{new}\ loop : Loop\ \texttt{in}\ (!loop().\texttt{ifnull}\ x\ \texttt{then}\ i\ \texttt{else}\ \overline{loop}\quad |\ \overline{loop}\ )$

$BRMDriver_3 = !alloc(request).\texttt{new}\ destructor\ \texttt{in}\ .$

$\quad \texttt{new}\ handler : Handler\ \texttt{in}\ .Pop\ (x).($

$\qquad |\ \overline{request}\ handler, destructor\ .!handler(y).\overline{x}\ y$

$\qquad |\ destructor().\mathtt{0}$

$\qquad |\ Finalize\ handler.Push\ x$

$\quad)$

The Garbage-Collection relation is the smallest $\square_3$ verifying

$x,\ \ P, x\ /\ fv(P)\qquad \{x\}\ \square_3\ P$

$x,\ \ P, Q, \{x\}\ \square_3\ P\qquad \{x\}\ \square_3\ P\ |\ x(y).Q$

$x,\ \ P, Q, \{x\}\ \square_3\ P\qquad \{x\}\ \square_3\ P\ |\ !x(y).Q$

$x,\ \ P, Q, \{x\}\ \square_3\ P\qquad \{x\}\ \square_3\ P\ |\ Finalize\ x.Q$

$x,\ \ P, Q, \{a : Handler\}\ \square_2\ \overline{delete}\ a\ .i\ |\ Q$

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

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

The Garbage-Collection relation is the smallest $\square_4$ verifying

$\{x\}\ \square_3\ P \quad \{x\}\ \square_4\ P$
$\{x : ERR\}\ \square_4\ P$ non-deterministically
$res(P) \quad memory\_limit \quad \{signal : MEM\}\ \square_4\ P$
$\{signal : PRIO, flag : Flag\}\ \square_4\ P\ |\ \overline{prioritize}\ client\ \ |\ \overline{flag}$

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 $\texttt{try} \cdots \texttt{catch}$ 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 $\square_\_$, 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 S is a set of resources and if r and s are elements of S such that r s then, for any garbage-collection scheme GC, $\longrightarrow_r^{GC}$ ▫ $\longrightarrow_r^{GC}$.

The inclusion derives directly from the definition of $\longrightarrow_r^{GC}$. The non-equality can be proved by examining process $(\nu a : s)(a(x) \mid \overline{a}$ □), 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 $\xrightarrow{x(\_)}_{pre}$ *P (respectively P* $\xrightarrow{\overline{x}\ \_}_{pre}$ *P), we say that P has a barb x() (respectively* $\overline{x}$ *). 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 S and with a limit n, a relation R is a weak barbed simulation if, whenever (P, Q) R,

- *if* $P \downarrow_\alpha$, *then* $Q \downarrow_\alpha$
- *if P* $\longrightarrow_n^{GC}$ *P then, for some Q , Q* $\longrightarrow_n^{GC}$ *Q P and)*e3910.9ifreedomΓ9230-301(a)-293(r)1(e)-1(l)1(ati)]TJ/F

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 $P = (\nu a : r)(\overline{a}\ \square \ \mid a().M_b)$ 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 $P \quad Q$, 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, $\square_1$ is sound, while $\square_2$, $\square_3$ and $\square_4$ are unsound. None is complete.

Soundness By definition, if $\{a\}\ \square_1\ P$, a is not free in P, therefore $P\{a \leftarrow \square_a\} = P$. We also have $(\nu a : r)P \equiv P \mid (\nu a : r)0$. We can prove easily that $P \mid (\nu a : r)0 \quad P$.

Unsoundness Let us write

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

and

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

We have $\{handler : Handler\}\ \square_2\ P$ and $P \longrightarrow Q$ by garbage-collection. Since $P \longrightarrow \ \downarrow_{\overline{a}}$ and $Q\neg \longrightarrow \ \downarrow_{\overline{a}}$, we conclude that $P\neg \quad Q$, hence $\square_2$ is unsound. The proof is identical for $\square_3$ and $\square_4$.

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

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.

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

Judgement $\Gamma \quad P : Bound(t,\lambda)$ states that, under environment Γ, P

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 N whose value is uniformly and $a \rightarrow r$ for the function defined on N whose value is r for a and for everything else.

Properties

Lemma 1 (Weakening).
If Γ is an environment and P a process such that $\Gamma \quad P : Bound(t, \lambda)$, then, for any $t \quad t$ and any $\lambda \quad \lambda$, we have $\Gamma \quad P : Bound(t , \lambda)$.

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 $\Gamma \quad P : Bound(r, \lambda)$ and $P \longrightarrow P$ then there is a r and a λ such that $\Gamma \quad Bound(r , \lambda)$ and $r \quad \Sigma_{x \quad N} \lambda(x) \quad r \quad \Sigma_{x \quad N} \lambda (x)$.

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 $r \quad r$: 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 S is a set of resources, if GC is a garbage-collection scheme, if P is a process, if $P \longrightarrow^{GC} P$ and if $\Gamma \quad P : Bound(r, 0_\lambda)$ then, for all $r \quad r$, we also have $P \longrightarrow^{GC}_{r'} P$.

S
The proof (detailed in the annex) is straight-28(e)(sour)1(c)-7120Td[(:)]TJ/F119.963Tf5.70Td[

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 $\mathbb{N}^4$ where $\Gamma \quad P : Bound((p,h,k,m),\lambda)$ 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

$$\Gamma \quad BRMDriver_4 \mid Pool : Bound((n,0,n,0),0_\lambda)$$

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

$$Bound((0,h,0,m),0_\lambda)$$

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

$$C ::= Chan(N, g, \lambda_g, p, \lambda_p) \; g, p \quad S, \lambda_g, \lambda_p : N \; \longrightarrow \; r$$

$$\frac{\begin{array}{c}\Gamma \quad a : Name(Chan(N, g, \lambda_g, p, \lambda_p), \_) \\ \Gamma, b : N \quad i : Bound(t_i \quad g, \lambda_i \quad \lambda_g) \\ t_i \quad t_i \quad p \qquad \lambda_i \quad \lambda_i \quad \lambda_p\end{array}}{\Gamma \quad a(b).i : Bound(t_i, \lambda_i)} \; \text{T-Rcv-Exchange}$$

$$\text{T-Snd-Exchange} \quad \begin{array}{c}\Gamma \quad a : Name(Chan(N, g, \lambda_g, p, \lambda_p), \_) \\ \Gamma \quad j : Bound(t_j \quad p, \lambda_j \quad \lambda_p) \qquad \Gamma \quad b : N \\ t_j \quad t\end{array}$$

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 $c\pi$, 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 $c\pi$, 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 resource 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 □ 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*

Garbage-collection If $P = (\nu a)0$ and $Q = 0$ 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 $\Box_{\_}$.*

Trivial.

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

By induction hypothesis, we also have

□

$$\Gamma \quad i\{x \leftarrow a\} : Bound(t \quad r, \lambda_i^{x \leftarrow a})$$
$$\Gamma \quad j\{x \leftarrow a\} : Bound(t, \lambda_j^{x \leftarrow a})$$

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

For any z distinct of x and a, we have

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

and

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

We also have

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

Also,

$$\lambda^{\ x \leftarrow a}(a) = \lambda\ (x) \quad \lambda\ (a) \ \ )\ .$$

z) .

Trivially, we have $weight(Bound(t\ , \lambda\ )) \quad Bound(t \quad r, \lambda_i)$. Which proves the case.

Communication Let us write

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

		Typage de P	
		Typage de $a(x).i$	
$\Gamma, x : N$	i :	$Bound(t \quad r, \lambda \quad \lambda_a)$	Par hypothse
Γ	a :	$Name(Chan(N, r, \lambda_a), \_)$	Par hypothse
Γ	$a(x).i$:	$Bound(t_1, \lambda_1)$	Par T-Rcv
	Avec	$t_1 \quad t$	
	$\lambda_1 \quad \lambda$		

		Typage de $\overline{a}\ b\ .j$	
Γ	j :	$Bound(t_j, \lambda_j)$	Par hypothse
Γ	a :	$Name(Chan(N, r, \lambda_a), \_)$	Par hypothse
Γ	b :	N	Par hypothse
Γ	$\overline{a}\ b\ .j$:	$Bound(t_2, \lambda_2)$	Par T-Snd
	Avec	$t_2 \quad t_j \quad r$	
	$\lambda_2 \quad \lambda_j \quad \lambda_a$		

		Typage de P	
Γ	$a(x).i$:	$Bound(t_1, \lambda_1)$	Cf. plus haut
Γ	:	$Bound(t_2, \lambda_2)$	Cf. plus haut
Γ	P :	$Bound(t_3, \lambda_3)$	Par T-Par
	Avec	$t_3 \quad t_1 \quad t_2$	
	$\lambda_3 \quad \lambda_1 \quad \lambda_2$		

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

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

$t_2 \quad t_j \quad r$

$t_1 \quad t$

Comme $\lambda_3 \quad \lambda_1 \quad \lambda_2$

$\lambda_1 \quad \lambda$

$\lambda_2 \quad \lambda_j \quad \lambda_a$

$\Gamma \quad Q : \qquad Bound(t_3, \lambda_3)$ Par T-Par

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 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* $\Gamma \quad P : Bound(r, 0_\lambda)$ *then* $res(P) \quad r$.

Trivial.

C.1 Main proof

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

D Typing finalization

We have

$$Loop = Name(Chan(\_, r, \lambda), \quad)$$
$$\Gamma \qquad i : Bound(r \quad r_n, \lambda)$$
$$r \quad r$$
$$\lambda \quad \lambda \quad x \to r_n$$

Typage de $\overline{loop}$

Γ	0	$: Bound(\ ,0_\lambda)$	Par *T-Nil*
Γ	$loop$	$: Name(Chan(\_, r', \lambda'), \_)$	Par hypothse
Γ	$\overline{loop}$	$: Bound(r', \lambda')$	Par T-Snd

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

Γ	$\overline{loop}$	$: Bound(r', \lambda')$	Cf. plus haut
Γ	i	$: Bound(r' \quad r_n, \lambda)$	Par hypothse
Γ	`ifnull` x `then` i `else` $\overline{loop}$	$: Bound(r', \lambda')$	Par T-Test-Nil

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

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

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

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

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

Γ	$!loop(). \cdots$	$: Bound(\ ,0_\lambda)$	Cf. plus haut
Γ	$\overline{loop}$	$: Bound(r', \lambda')$	Cf. plus haut
Γ	$!loop() \cdots \mid \overline{loop}$	$: Bound(r', \lambda')$	Par T-Par

Typage de $Finalize\ x.i$

Γ	$!loop() \cdots \mid \overline{loop}$	$: Bound(r', \lambda')$	Cf. plus haut
Γ	$(\nu loop : Loop)(\cdots)$	$: Bound(r', \lambda')$	