

# Rapport

## Un noyau générique pour interprète CCS

Philippe Wang<sup>1</sup>

3 février 2007

*Revision : 1.6*

---

<sup>1</sup>[philippe.wang@etu.upmc.fr](mailto:philippe.wang@etu.upmc.fr)  
Université Pierre et Marie Curie, Paris, France  
Master de Recherche en Informatique,  
Spécialité Science et Technologie du Logiciel - Algorithmique et Programmation

### **Résumé**

Ce rapport présente une étude de l'implantation d'un noyau générique pour interprète CCS (*Calculus for Communicating Systems*) écrit en Objective Caml. Sont mis en évidence les problèmes de calculs de transitions par évaluation immédiate, les avantages du polymorphisme et son impact sur la généricité.

## Table des matières

<b>1 Introduction</b>	<b>4</b>
1.1 CCS	4
1.2 Syntaxe	4
<b>2 Syntaxe</b>	<b>4</b>
<b>3 Implantation d'un noyau générique</b>	<b>5</b>
3.1 Calcul de toutes les transitions possibles	5
3.2 Extensions	7
3.2.1 Nommage de processus	7
3.2.2 Autres extensions	7
<b>4 Conclusion</b>	<b>8</b>
<b>5 Annexe</b>	<b>8</b>
<b>6 Bibliographie</b>	<b>11</b>

# 1 Introduction

Ce document rapporte l'étude d'une implantation d'un noyau générique d'interprète CCS (Calculus of Communicating Systems).

CCS est un « langage concurrent », ou plus spécifiquement une « algèbre de processus », où les processus sont des éléments de « première classe » du langage. Le comportement d'un processus est représenté par toutes les transitions possibles du processus.

Ce document présente un noyau générique d'interprète de CCS écrit en Objective Caml. La généricité est basée sur le polymorphisme d'Objective Caml.

## 1.1 CCS

Pour avoir plus d'informations sur le CCS, se reporter par exemple au livre de Milner [3] ou à un cours [1] [2].

## 1.2 Syntaxe

La syntaxe abstraite et la syntaxe concrète ne font qu'une : il s'agit d'écrire dans la syntaxe d'Objective Caml. L'avantage de ce choix est multiple : d'une part il permet de ne pas avoir à écrire d'analyseurs lexical et syntaxique qui casseraient la généricité, et d'autre part il permet de profiter du compilateur Objective Caml pour détecter les erreurs de syntaxe et de types.

En effet, si on écrit un *frontend* spécifique pour l'interprète, on perd directement la possibilité de pouvoir passer n'importe quelle valeur Objective Caml dans les canaux de communication du langage CCS, à moins d'écrire un *frontend* plus vaste encore que celui d'Objective Caml.

L'inconvénient est que la syntaxe n'est pas aussi compacte que si un petit *frontend* avait été utilisé, comme il l'est fait pour de nombreux des interprètes CCS.

(*Pourquoi donc réinventer la roue ? Certes on le fait déjà un peu !*)

# 2 Syntaxe

Pour bien comprendre la syntaxe de la déclaration de types, se reporter à la documentation du langage Objective Caml ([4] [5]).

```
1 type 'a process =
2   | Nil
3   | Output      of 'a action * 'a process
4   | Input       of channel * ('a -> 'a process)
5   | Par         of 'a process * 'a process
6   | Sum         of 'a process * 'a process
7   | Restriction of channel list * 'a process
8 and 'a action =
9   | Tau
10  | Data        of channel * 'a
11 and 'a transition =
12  | InT         of channel * ('a -> 'a process)
13  | OutT        of 'a action * 'a process
```

```
14 and name = string
15 and channel = string
```

### Listing 1 – Structures de données pour la représentation des processus

Un processus est soit `0` (`Nil`), soit une transmission de valeur (`Output`), soit une réception de valeur (`Input`), soit deux (sous-)processus en parallèle (`Par`), soit une somme de processus (`Sum`), soit une restriction (`Restriction`)<sup>2</sup>.

Une action est soit constituée d'un canal de transmission et d'une donnée : la donnée est envoyée dans le canal, soit une action silencieuse (`Tau`).

Une transition est soit une entrée (`InT`), soit une sortie (`OutT`).

On peut noter l'usage de la construction de fermetures anonymes : c'est un plus offert par les langages fonctionnels. On en profite pour représenter un processus en attente d'une donnée, puisqu'une fermeture est en attente de son (ou ses) argument(s).

Il est important de noter que l'utilisation du polymorphisme offert par Objective Caml permet de définir cette interface générique. Cependant il est à noter que le système devient monomorphe dès qu'on choisit un type de valeurs à transmettre dans un système. C'est-à-dire que si je choisis de faire un système de communication transmettant des nombres entiers de type `int`, je ne pourrai pas transmettre des chaînes de caractères dans ce système. Cependant, il est au moins aussi important de noter que toutes les fonctions travaillant sur ces structures de données sont génériques et polymorphes. Autrement dit, le système reste polymorphe jusqu'au dernier moment : celui où l'on commence à transmettre de « vraies valeurs ».

## 3 Implantation d'un noyau générique

### 3.1 Calcul de toutes les transitions possibles

La fonction `step`<sup>3</sup> calcule toutes les transitions possibles d'un processus et les rend sous forme d'une liste.

Attention : le nombre de transitions est exponentiel par rapport au nombre de processus non séquentiels. Cela peut dépasser la capacité de la pile d'Objective Caml, et rendre l'exception `Stack_overflow`... L'évaluation paresseuse peut paraître plus maligne dans ce cas, cependant dans « la vraie vie » on ne fait pas non plus ce genre de calculs...

```
1 (* computes and returns the list of all the possible transitions *)
2 let rec step = function
3   | Nil -> []
4   | Output (act, p) -> [OutT (act, p)]
5   | Input (chan, f) -> [InT (chan, f)]
6   | Sum (p1, p2) -> step p1 @ step p2
```

<sup>2</sup>on peut remarquer que les restrictions étant permutables, on choisit pour simplifier de mettre les restrictions successives dans une liste.

<sup>3</sup>Les amateurs du langage Haskell pourront dire que c'est un peu « verbeux » et ils n'auraient pas vraiment tort car c'est effectivement plus compact... Cependant les amateurs des langages non-fonctionnels ne pourront pas prétendre la même chose!

```

7 | Restriction (r, p) ->
8   let ps = step p
9   in
10    List.fold_left
11      (fun l -> function
12        | OutT (Data (chan, v), p2) ->
13          if not (List.mem chan r) then
14            OutT (Data(chan, v), (Restriction (r, p2))) :: l
15          else l
16        | InT (chan, f) ->
17          if not (List.mem chan r) then
18            InT (chan, fun x -> (Restriction (r, f x))) :: l
19          else l
20        | OutT (Tau, p2) ->
21          OutT (Tau, (Restriction (r, p2))) :: l
22      )
23    []
24    ps
25 | Par (p1, p2) ->
26   let plt, p2t = step p1, step p2 in
27   let pls =
28     (List.fold_left
29       (fun l -> function
30         | OutT(act, p) -> OutT (act, (Par (p, p2))) :: l
31         | InT(chan, f) -> InT (chan, fun x -> (Par (f x, p2))) :: l
32         | _ -> l)
33       []
34       plt)
35   and p2s =
36     (List.fold_left
37       (fun l -> function
38         | OutT(act, p) -> OutT (act, (Par (p, p1))) :: l
39         | InT(chan, f) -> InT (chan, fun x -> (Par (f x, p1))) :: l
40         | _ -> l)
41       []
42       p2t)
43   in
44   let tpltp2 : 'a transition list list =
45     List.map
46       (function
47         | InT (chan, f) ->
48           List.fold_left
49             (fun l -> function
50               | OutT (Data (chanB, v), p) ->
51                 if chanB = chan then
52                   OutT (Tau, Par (f v, p)) :: l
53                 else l
54               | _ -> l)
55             []
56             p2t
57         | OutT (Data (chan, v), p) ->
58           List.fold_left
59             (fun l -> function
60               | InT (chanB, f) ->
61                 if chanB = chan then
62                   OutT (Tau, Par (p, f v)) :: l
63                 else l
64               | _ -> l)
65             []
66             p2t
67         | _ -> [])
68       plt
69   in
70   pls @ p2s @ (List.flatten tpltp2)
71 (* val step : 'a process -> 'a transition list *)

```

Listing 2 – Calcul de toutes les transitions possibles

## 3.2 Extensions

### 3.2.1 Nommage de processus

Il est possible d'étendre le système pour pouvoir donner un nom à des définitions de processus.

```
1 type 'a def =
2 | Let      of name      * 'a process
3 | Rec      of name      * 'a process
4
5 type 'a top = Process of 'a process | Definition of 'a def
```

Listing 3 – Extension : définitions de processus

On utilise alors le type  $\forall \alpha. \alpha \text{ top}$  qui se place au dessus des types  $\forall \alpha. \alpha \text{ process}$  et  $\forall \alpha. \alpha \text{ def}$ .

Pour pouvoir utiliser les définitions de processus, il faut ajouter un environnement.

```
1 type 'a process =
2 | Nil
3 | Output of 'a action * 'a process
4 | Input  of channel  * ('a -> 'a process)
5 | Par    of 'a process * 'a process
6 | Sum    of 'a process * 'a process
7 | Restriction of channel list * 'a process
8 | Process of name (* <= add *)
9
10 type 'a environment =
11 | LET of name * 'a process * 'a environment
12 | REC of name * 'a process * 'a environment
13 | NIL
14
15 let string_of_name n = n
16
17 let rec look_up name = function
18 | LET (n, p, e) -> if n = name then p else look_up name e
19 | REC (n, p, e) -> if n = name then p else look_up name e
20 | NIL ->
21   failwith ("Error: undefined process: [" ^ string_of_name name ^ "]")
22
23 let add_def environment = function
24 | Let (n, p) -> LET (n, p, environment)
25 | Rec (n, p) -> REC (n, p, environment)
```

Listing 4 – Extension : nommage de processus

Cela implique donc l'ajout d'un environnement, mais aussi sa gestion, notamment au niveau de la fonction `step`. On se rend alors compte que la définition récursive pose un problème certain pour cette fonction `step`. Il faudrait faire intervenir l'évaluation paresseuse pour ne pas boucler indéfiniment...

### 3.2.2 Autres extensions

Il est possible d'imaginer beaucoup d'extensions pour le langage CCS car en lui-même il ne fournit pas assez de souplesse de programmation et ses possibilités sont relativement limitées. Par exemple, introduire des *tics* pour donner un temps limité aux instructions afin d'éviter les blocages (potentiellement) infinis.

Mais avant on peut aussi penser à implanter la conditionnelle...

## 4 Conclusion

Ce rapport présente l'implantation d'un noyau générique pour interprète CCS avec un langage fonctionnel à évaluation immédiate, et avec passages par valeurs. La généricité permet d'adapter le noyau pour n'importe quelles types de données échangées entre les processus, en particulier les chaînes de caractères qui sont capables de véhiculer n'importe quelle donnée sérialisée : mais dans ce cas on perd la sûreté de typage offerte par Objective Caml.

Pour calculer une exécution d'un processus, il y a deux possibilités : soit on calcule toutes les transitions et on en sélectionne aléatoirement, ce qui se fait bien dans un langage paresseux car les processus non sélectionnés ne seront jamais calculés, soit on écrit une fonction similaire à `step` qui ne calcule pas toutes les transitions possibles mais qui en sélectionne une aléatoirement à chaque fois qu'un choix se présente.

## 5 Annexe

```
(*****  
(* CCS Interpreter (2006) *)  
(* by Philippe Wang *)  
(* *)  
(* $+Revision: 1.10 +$ *)  
(*****  
(* $+Id: ccs_interpreter.ml,v 1.10 2006/11/30 13:19:29 philippej Exp +$*)  
  
(* data types *)  
type 'a process =  
  | Nil  
  | Output of 'a action * 'a process  
  | Input of channel * ('a -> 'a process)  
  | Par of 'a process * 'a process  
  | Sum of 'a process * 'a process  
  | Restriction of channel list * 'a process  
and 'a def =  
  | Let of name * 'a process  
  | Rec of name * 'a process  
and 'a action =  
  | Tau  
  | Data of channel * 'a  
and 'a transition =  
  | InT of channel * ('a -> 'a process)  
  | OutT of 'a action * 'a process  
and name = string  
and var = string
```

```

and channel = string

(* computes and returns the list of all the possible transitions *)
let rec step = function
| Nil -> []
| Output (act, p) -> [OutT (act, p)]
| Input (chan, f) -> [InT (chan, f)]
| Sum (p1, p2) -> step p1 @ step p2
| Restriction (r, p) ->
    let ps = step p
    in
    List.fold_left
      (fun l -> function
        | OutT (Data (chan, v), p2) ->
            if not (List.mem chan r) then
              OutT (Data(chan, v), (Restriction (r, p2))) :: l
            else l
        | InT (chan, f) ->
            if not (List.mem chan r) then
              InT (chan, fun x -> (Restriction (r, f x))) :: l
            else l
        | OutT (Tau, p2) ->
            OutT (Tau, (Restriction (r, p2))) :: l
      )
      []
      ps
| Par (p1, p2) ->
    let plt, p2t = step p1, step p2 in
    let pls =
      (List.fold_left
        (fun l -> function
          | OutT(act, p) -> OutT (act, (Par (p, p2))) :: l
          | InT(chan, f) -> InT (chan, fun x -> (Par (f x, p2))) :: l
          | _ -> l)
        []
        plt)
    and p2s =
      (List.fold_left
        (fun l -> function
          | OutT(act, p) -> OutT (act, (Par (p, p1))) :: l
          | InT(chan, f) -> InT (chan, fun x -> (Par (f x, p1))) :: l
          | _ -> l)
        []
        p2t)
    in
    let tpltp2 : 'a transition list list =
      List.map
        (function
          | InT (chan, f) ->

```

```

List.fold_left
  (fun l -> function
    | OutT (Data (chanB, v), p) ->
      if chanB = chan then
        OutT (Tau, Par (f v, p)) :: l
      else l
    | _ -> l)
  []
  p2t
| OutT (Data (chan, v), p) ->
  List.fold_left
    (fun l -> function
      | InT (chanB, f) ->
        if chanB = chan then
          OutT (Tau, Par (p, f v)) :: l
        else l
      | _ -> l)
    []
    p2t
  | _ -> [])
  p1t
in
  p1s @ p2s @ (List.flatten tp1tp2)

```

## 6 Bibliographie

### Références

- [1] [Frédéric Peschanski](#) : *Sémantique de la concurrence et de la mobilité*, cours de master, Université Pierre et Marie Curie, 2006/2007.
- [2] Catuscia Palamidessi, Pierre-Louis Curien, Francesco Zappa Nardelli : *Concurrence*, cours de master, Master Parisien de Recherche en Informatique, 2006/2007.  
<http://mpri.master.univ-paris7.fr/C-2-3.html>
- [3] Milner, R. : *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, (1989).
- [4] [Xavier Leroy](#) (with [Damien Doligez](#), Jacques Garrigue, Didier Rémy and Jérôme Vouillon) : *The Objective Caml system 3.09. Documentation and user's manual*. Inria, France (1985-2006).  
<http://caml.inria.fr/pub/docs/manual-ocaml/>
- [5] [Emmanuel Chailloux](#), Bruno Pagano, [Pascal Manoury](#) : *Développement d'applications avec Objective Caml*, O'REILLY, 2000.