PC3R
Cours 10 - Bilan, Ouverture : Pi en Go

Romain Demangeon

PC3R MU1IN507 - STL S2

15/04/2021

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Evaluation

- Projet (20%) : 16 Mai
- TMEs 1-5 (20%) : 16 Mai (resoumission)
- ER1 (20%) : notes pendant les vacances.
- ER2 (40%) : semaine d'examen.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

PC3R: Contenu du Cours

- Nature du cours:
  - Pas un cours de programmation classique (on prend un paradigme et on explore les possibilités, les méthodes de développement, ...)
  - Pas un cours de technologie (on apprend à se servir d'un langage / d'une bibliothèque / d'une application)
  - un cours de survol de différents langages, technologies, méthodes liées à la programmation concurrente.
  - Pas de soucis d'exhaustivité.
  - des langages/technologies *legacy* (FT, Esterel) ou plus confidentiels (Promela, Lustre) et des langages/technologies populaires (JS, Go, POSIX).
- Objectifs:
  - offrir une vision de différentes manières de programmer des systèmes concurrents, dans différents domaines.
  - exhiber des similarités et des différences dans le traitement des protocoles/programmes distribués.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

PC3R: Que retenir ?

1. "la concurrence c'est difficile":
   - ne pas se surestimer face aux questions de concurrences qui peuvent arriver en situation professionnelle.
2. reconnaître des situations identifiables:
   - coopération vs. préemption,
   - protocole vérifiable en SPIN,
   - approche ressource vs. approche service,
   - système synchrone, ...
3. enrichir sa culture pour avoir de l'inspiration
   - richesse et difficulté de JS,
   - traitement monadique de la concurrence de `Lwt`,
   - utilisation de canaux plutôt que de mémoires partagées,
   - décomposition des interactions en requêtes / réponses,
   - transformation d'un programme en manipulation de flux, ...

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Après PC3R

- M2 STL:
  - Théorie de la concurrence: PPC-1 (S3)
  - Modèle d'acteurs: PPC-2 (S3)
  - Programmation synchrone: PPC-2 (S3)
  - Vérification des programmes (séquentiel) avec des types: TAS-1 (S3)
  - Vérification des programmes (séquentiel) avec de l'analyse statique: TAS-2 (S3)
  - Algorithmique et Web: DAAR (S3)
- PPC:
  - Continuation de PC3R.
  - Partie 1: plus théorique (modèles mathématiques, preuves)
  - Partie 2: de la pratique aussi (programmation synchrone, drones)

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

π-calcul

- π: modèle mathématique pour la programmation concurrence de haut-niveau (design de protocoles ou programmes).
- Intérêt:
  - modéliser des protocoles/programme formellement pour les étudier:
    - comparaison de protocoles/programmes,
    - validation de propriétés (terminaison, vivacité, correction, ...)
  - étudier scientifiquement la programmation concurrente:
    - expressivité (encodages),
    - équivalence (égaliser des programmes différents avec le même comportement observable)
    - décidabilité (des problèmes d'accessibilité, de validation, ...)
- Algèbre de processus: langage mathématique représentant les processus sous forme de termes, avec une syntaxe et une sémantique étiquetée formelles
  - CCS, π, *join*, ...
  - années 1980 (*Robin Milner*) puis 1990.
  - continue à être exploré aujourd'hui.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

[PPC] Pourquoi apprendre le π-calcul

π-calcul: calcul formel concurrent par passage de messages.

- On ne programme pas en π-calcul.
- Le π-calcul est un modèle des comportements des programmes concurrents, distribués, répartis.
  - il ne modélise que la partie observable: envoi et réception de messages.
- On s'en sert pour étudier des propriétés de la concurrence (bloquages, terminaison, fuite d'information)
  - Prouver des théorèmes sur des modèles permet:
    - de développer des nouveaux langages sûrs,
    - de découvrir des techniques de programmation,
    - de construire des vérifications de programmes réels.
  - Il y a parfois une chaîne entre la pratique et la théorie.

    π-calcul typé $\longrightarrow$ types de sessions $\longrightarrow$ *Scribble* $\longrightarrow$ Moniteurs (`Python`)
- Pourquoi l'apprendre en M2 STL:
  - pour lire des papiers sur la concurrence,
  - pour développer des automatismes de programmation,
  - pour comprendre la difficulté de la programmation distribuée,
  - pour la culture générale informatique.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

[PPC] "Lettre-grecque"-calculs

- λ-calcul (1930): programmation fonctionnelle
  $M, N ::= \lambda x.M \mid M\ N \mid x$
- μ-calcul (1983): formules logiques "infinies":
  $\phi ::= \top \mid \bot \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid X \mid \langle\rangle\phi \mid []\phi \mid \mu X.\phi \mid \nu X.\phi$
- π-calcul (1992): passage de messages et mobilité:
  $P, Q ::= 0 \mid a(x).P \mid \overline{a}\langle v\rangle.P \mid !a(x).P \mid (P \mid Q) \mid (\nu a)\ P \mid (P + Q)$
- σ-calcul (1995): programmation objet:
  $a, b ::= [l_j = \sigma(x_j)b_j] \mid a.l_j \mid a.l_j := \sigma(x)b$
- ρ-calcul (1998): formalisme pour la réécriture:
  $t ::= x \mid f(t, \ldots, t) \mid \{t, \ldots, t\} \mid u_{[E]} \to t \mid [t]t$
- d'autres variantes: $\lambda_{\sigma\Uparrow}$-calcul (substitutions explicites), $\lambda_{\mu\mu'}$-calcul (pile d'exécutions), $\mathtt{HO}_\pi$ (processus dans les messages)

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

π: Fondamentaux

- Canaux:
  - noms $a, b, c \ldots$ des canaux de communication,
  - `chan int a` de *Go*
  - utilisation de types de base (`int`, `string`) (pas nécessaire en π pur)
  - ordre supérieur: `chan chan int`
- Processus:
  - programmes concurrents,
  - composé de sous-processus,
  - envoi/réception sur des canaux,
  - équivalent au programmes *Go*.
- Substitution:
  - $P[y/x]$ est le processus P dans lequel toutes les occurences de x sont remplacées par y.
  - on peut imaginer la même chose pour les programmes *Go*.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Syntaxe

- processus nul 0
  - processus inactif: aucun comportement observable.
  - représentation d'un programme terminé ou bloqué,
  - en *Go*: programme vide.
- émission sur un canal: $\overline{c}\langle v \rangle.P$
  - sujet c, objet v, continuation P
  - "on envoie la valeur v sur le canal c, quand la synchronisation a eu lieu, on continue avec le processus P"
  - en *Go*:

```
c <- v
[[P]]
```

- réception depuis un canal: $c(x).P$
  - sujet c, objet x, continuation P
  - "on reçoit la valeur x sur le canal c, quand la synchronisation a eu lie, on continue avec le processus P, qui peut utiliser x"
  - en *Go*:

```
x := <- c
[[P]]
```

le `:=` est important (déclaration + affectation)

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Syntaxe et Sémantique

- composition parallèle: $P \mid Q$
  - mise en parallèle de deux processus P et Q.
  - commutatif et associatif
  - en *Go*:

```
go func(){[[P]]}()
go func(){[[Q]]}()
```

- restriction: $(\nu c)\ P$
  - création d'un canal c local à P uniquement.
  - *à la Church* on peut écrire le type $(\nu c : T)\ P$
  - en *Go*:

```
c := make(chan T)
[[P]]
```

- Sémantique opérationnelle à petit pas donnée par une Système de Transition Etiqueté (LTS) ou une réduction.
  - c'est la limite de l'analogie avec *Go* (sémantique d'évaluation)
  - $a(x).P \mid \overline{a}\langle v \rangle.Q \longrightarrow P[v/x] \mid Q$
  - $+$ des règles administratives.
  - en *Go*, c'est l'exécution du programme.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

Exemple

- $a(x).(\overline{x}\langle\rangle.0 \mid b().0) \mid \overline{a}\langle b\rangle.0 \mid \overline{a}\langle c\rangle.0$
  - on oublie les occurences de 0 en fin de processus et les objets vides: $a(x).(\overline{x} \mid b) \mid \overline{a}\langle b\rangle \mid \overline{a}\langle c\rangle$
  - en *Go*:

```
a := make(chan int)
b := make(chan int)
c := make(chan int)
go func(){
  x := <- a
  go func(){x <- 0}()
  go func(){<- b}()
}()
go func(){a <- b}()
go func(){a <- c}()
```

On n'a pas de type `unit` évident, donc on utilise `int` (plutôt que `struct{}`).

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

Exemple (II)

- non-déterminisme inhérent de la concurrence:
- $a(x).(\overline{x} \mid b) \mid \overline{a}\langle b \rangle \mid \overline{a}\langle c \rangle \longrightarrow \overline{b} \mid b \mid \overline{a}\langle c \rangle \longrightarrow \overline{a}\langle c \rangle$
- $a(x).(\overline{x} \mid b) \mid \overline{a}\langle b \rangle \mid \overline{a}\langle c \rangle \longrightarrow \overline{c} \mid b \mid \overline{a}\langle b \rangle \not\longrightarrow$
- capturé imparfaitement par le programme *Go* (qui utilise un ordonnanceur semi-coopératif pour les goroutines)
  - on peut simuler avec des `time.Sleep` sur des valeurs aléatoires.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

Exemple (III)

- $(\nu c)\ (\overline{a}\langle c\rangle \mid \overline{c}) \mid a(x).x$
  - en *Go*:

```
a := make(chan int)
go func(){
  c := make(chan int)
  a <- c
  c <- 0
}()
go func(){
  x := <- a
  <- x
}()
```

  - mobilité: extrusion de noms locaux
    - initialement c est créé dans le premier processus/la première goroutine et n'existe qu'à cet endroit.
    - c est extrudé par la communication sur a, et la deuxième goroutine le "connaît" et peut l'utiliser dans une réception.
    - en π: $(\nu c)\ (\overline{a}\langle c\rangle \mid \overline{c}) \mid a(x).x \longrightarrow (\nu c)\ (\overline{c} \mid c) \longrightarrow 0$

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

Réplication

- le calcul actuel est terminant
  - à chaque réduction on consomme deux préfixes.
  - une récursion / réplication est nécessaire pour être Turing-complet.
- dans le π-calcul "classique": réplication
  - il existe d'autres manières (définitions récursives, récursion explicite)
  - $!P$ est "une infinité de copies de P en parallèle"
  - équivalence entre $!P$ et $!P \mid P$ (on déplie/replie la réplication)
  - en *Go*:

```
for {
  go func(){[[P]]}()
  }
```

  - problème: divergence dans la création de goroutines.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Input répliqué

- en π-calcul: input répliqué
  - réplication gardé $!a(x).P$
  - équivalent en expressivité à la réplication
  - on ne déplie que quand on s'en sert:
    - $!a(x).P \mid \overline{a}\langle v \rangle.Q \longrightarrow !a(x).P \mid P[v/x] \mid Q$
  - en *Go*:

```
for {
 x := <- a
 go func(y){[[P]]}(x)
 }
```

  - comportement de serveur: attente de la requête, création d'un "thread client" qui la traite.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

Choix gardé

- existence d'un opérateur de choix $+$ permettant de choisir de manière non déterministe entre deux comportements
  - $P + Q$ peut se comporter comme P ou Q
- en pratique on considère le choix gardé: P et Q doivent commencer par des préfixes d'émission ou de réception.
- par exemple $a(x).P + a(y).Q$
  -
```
select {
  case x := <- a :
  [[P]]
  case y := <- a :
  [[Q]]
  }
```

- ou encore $a(x).P + \overline{b}\langle v \rangle.Q$ (choix mixte)
  -
```
select {
  case x := <- a :
  [[P]]
  case b <- v :
  [[Q]]
  }
```

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

Polyadicité

Syntaxe des préfixes polyadiques

Le π-calcul polyadique utilise les préfixes $a(x_1, x_2, \ldots, x_n).P$ et $\overline{a}\langle v_1, v_2, \ldots, v_n\rangle.Q$ avec $n \in \mathbb{N}$.

- Idée: transporter plusieurs messages en même temps.
- Sémantique:
  $a(x_1, \ldots, x_n).P \mid \overline{a}\langle v_1, \ldots, v_n\rangle \longrightarrow P[v_1, \ldots, v_n/x_1, \ldots, x_n] \mid Q$
- Exemple: $(\overline{a}\langle b, c\rangle + \overline{a}\langle b, b\rangle) \mid a(x, y).(\overline{x} \mid y)$
- en *Go*, limite de l'analogie (utilisation de `struct`):

```
type message struct{
  gauche chan int
  droite chan int}
b := make(chan int)
c := make(chan int)
a := make(chan message)
go func(){select {
    case a <- message{gauche : b, droite : c}: break
    case a <- message{gauche : b, droite : b}: break
  }}()
go func(){
  m := <- a
  go func(){m.gauche <- 0}()
  go func(){<- m.droite}()
  }()
```

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Serveur

- en π:
  - $!a(x, r).\overline{r}\langle f(x)\rangle$: applique une fonction f au corps de la requête x et envoie le résultat sur le canal de retour r.
  - $(\nu c)\ \overline{a}\langle 3, c\rangle.c(y)$: client qui appelle f.
- en *Go*:

```
type message struct{
  corps int
  retour chan int}
a := make(chan message)
go func(){
  for{
    xr := <- a
    go func(){
      xr.retour <- f(xr.corps)
    }()
   }}()
go func(){
  c := make(chan int)
  a <- message{corps : 3, retour : c}
  y := <- c
  }()
```

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

Serveur récursif

- en π:
  - on utilise les gardes $[x = e]$ de π (classique dans la littérature).
  - $!f(n, r).[n = 0]\overline{r}\langle 1\rangle + [n \neq 0](\nu c)\ (\overline{f}\langle n - 1, c\rangle \mid c(y).\overline{r}\langle y * n\rangle)$:
- en *Go*:

```
type message struct{
  corps int
  retour chan int}
f := make(chan message)
go func(){for{
  nr := <- f
  go func(){
    switch n == 0 {
      case true: nr.retour <- 1
      case false:
        c := make(chan int)
        f <- message{corps : (nr.corps -1), retour : c}
        y := <- c
        nr.retour <- (y * nr.corps)
    }
  }()
}}()
```

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

Comparaison de processus

- $P = a.\overline{b} + \overline{b}.a$
- ```
  func P(a chan int, b chan int) { select{
     case <- a : b <- 0
     case b <- 0 : <- a
    }}
  ```
- $Q = a \mid \overline{b}$
- ```
  func Q(a chan int, b chan int) {
    go func(){<- a}()
    go func(){b <- 0}()   }
  ```
- P et Q sont des processus différents, mais ont-ils le même comportement ?

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

Comparaison de processus

- $P = a.\overline{b} + \overline{b}.a$
- ```
  func P(a chan int, b chan int) { select{
     case <- a : b <- 0
     case b <- 0 : <- a
    }}
  ```
- $Q = a \mid \overline{b}$
- ```
  func Q(a chan int, b chan int) {
    go func(){<- a}()
    go func(){b <- 0}()    }
  ```
- P et Q sont des processus différents, mais ont-ils le même comportement ?
- réponse standard: oui
  - preuve: équivalences comportementales, bisimulations.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Comparaison de processus

- $P = a.\overline{b} + \overline{b}.a$
- ```
  func P(a chan int, b chan int) { select{
     case <- a : b <- 0
     case b <- 0 : <- a
    }}
  ```
- $Q = a \mid \overline{b}$
- ```
  func Q(a chan int, b chan int) {
    go func(){<- a}()
    go func(){b <- 0}()    }
  ```
- P et Q sont des processus différents, mais ont-ils le même comportement ?
- réponse standard: oui
  - preuve: équivalences comportementales, bisimulations.
- pourtant en *Go*:

```
c := make(chan int)
P(c,c)
```

vs.

```
c := make(chan int)
Q(c,c)
```

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Comparaison de processus

- $P = p.(bt.\overline{the} + bc.\overline{cafe})$
- ```
  func P() {
    p <- ;
    select{
      case bt <- : the <- 0
      case bc <- : cafe <- 0
    }}
  ```
- $Q = p.bt.\overline{the} + p.bc.\overline{cafe}$
- ```
  func Q() {
    select{
      case p <- : bt <- ; the <- 0
      case p <- : bc <- ; cafe <- 0
    }}
  ```
- P et Q sont des processus différents, mais ont-ils le même comportement ?

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Comparaison de processus

- $P = p.(bt.\overline{the} + bc.\overline{cafe})$
- ```
  func P() {
    p <- ;
    select{
      case bt <- : the <- 0
      case bc <- : cafe <- 0
    }}
  ```
- $Q = p.bt.\overline{the} + p.bc.\overline{cafe}$
- ```
  func Q() {
    select{
      case p <- : bt <- ; the <- 0
      case p <- : bc <- ; cafe <- 0
    }}
  ```
- P et Q sont des processus différents, mais ont-ils le même comportement ?
- réponse standard: non
  - preuve: équivalences comportementales, bisimulation.
  - P et Q ont les mêmes traces
  - P offre un choix, Q choisit.
  - P et Q ne sont pas bisimilaires.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Conclusion

- π: un cadre mathématique pour l'étude des systèmes concurrents.
  - passage de messages,
  - sémantique de réduction,
  - équivalence comportementales
  - programme de PPC-1.
  - relativement facilement encodable en *Go*
    - "même esprit".
- PC3R:
  - "la concurrence c'est difficile".
- TD10:
  - exercices de *Lustre* (cf. Cours 09).

Bonne fin d'année !

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257