π-calculus and Go

Giorgio Marinelli

University of Camerino

January 21, 2019

Introduction

What we are going to do:

- We illustrate and compare the π-calculus with the Go programming language.
- We show how to encode a π-calculus expression using the Go primitives.
- We show, as an example, how to encode the *Church numerals* in the π-calculus and how they are translated into Go.

. . . and what we are not going to do

We do not show an implementation of the π-calculus in the Go language, or vice versa.

The π-calculus

The π-calculus was first presented in "A Calculus of Mobile Processes" [1] by Milner, Parrow and Walker in 1989.

It is an extension of the process algebra CCS.

It adds the possibility to represent mobile processes, their interconnections evolve during computation.

The CCS process algebra

CCS syntax[1]:

$$P ::= \ 0 \ \mid \ \alpha.P \ \mid \ P + Q \ \mid \ P|Q \ \mid \ P \setminus a$$

Where: α might be an input channel a, an output channel $\bar{a}$, or a silent action τ; P and Q are processes.

- 0 : the nil process;
- $\alpha.P$: do something and proceed as P;
- $P + Q$: act either as P or as Q;
- $P|Q$: run P and Q in parallel;
- $P \setminus a$: treat the channel a as private for P.

[1] In this CCS syntax, infinite behaviour, relabelling and definitions are not specified.

CCS limitations

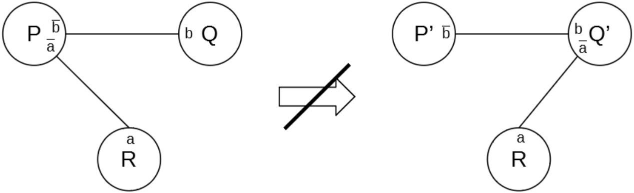

Figure 1: Channels as values

In CCS process algebra this "transition" is not possible. The process P should pass the $\bar{b}$ channel to Q.

The π-calculus syntax

In the π-calculus we have two primitive entities:
names $x, y, \ldots \in \mathcal{X}$ and processes $P, Q, \ldots \in \mathcal{P}$
Syntax for processes [2]:

$$P ::= \quad \sum_{i \in I} \pi_i.P_i \quad | \quad P|Q \quad | \quad !P \quad | \quad (\nu x)P$$

Where: π_i might be an input prefix $x(y)$ or an output prefix $\bar{x}y$.

- $\sum_{i \in I} \pi_i.P_i$ 0 : act as one of the processes in the sum.
  E.g.: with $i = 0$, we have the 0 (or nil) process; with $i = 1$, we have $x(y).P$ or $\bar{x}y.P$; with $i = 2$, we have $P_1 + P_2$;
- $P|Q$: run P and Q in parallel;
- $!P$: replication of P, i.e. $P|P|P| \ldots$;
- $(\nu x)P$: "new x in P", make the name x private for P.

The Go programming language

The project was started by Robert Griesemer, Rob Pike and Ken Thompson in 2007.

In 2009 Go became a public open source project.

Go has a C-like syntax and concurrency inspired by languages like Newsqueak and Limbo (both of them inspired by Tony Hoare's CSP language).

The Go syntax

Here is a Go program that will print on the screen the string "hello" followed by a number.

```
$ go run print.go
hello 6
```

```go
package main

import "fmt"

func main() {
  var sum int = 0
  var max int = 4

  for i := 0; i < max; i++ {
    sum += i
  }

  fmt.Println("hello", sum)
}
```

Listing 1: print.go

Concurrency in Go

In Go there are *goroutines*, a sort of lightweight threads, and *channels*, to create communications between *goroutines*.

```
$ go run channel.go
Run a goroutine
goroutine: 3
main: 5
```

```
package main
import "fmt"

func print_from_channel (channel chan int) {
  var y int = <- channel
  fmt.Println("goroutine:", y)
  channel <- (y + 2)
}

func main () {
  c := make (chan int)
  fmt.Println("Run a goroutine")
  go print_from_channel (c)
  c <- 3
  fmt.Println("main:", <- c)
}
```

Listing 2: channel.go

From π to Go

Let's see if for every basic process expression of the π-calculus there exist a corresponding expression or constructor in Go.

$$P ::= \sum_{i \in I} \pi_i.P_i \ \mid \ P|Q \ \mid \ !P \ \mid \ (\nu x)P$$

We can consider that any *process* and *name* in the π-calculus corresponds to a function (or a sequence of statements) and a channel in Go, respectively.

```
package main

type Name chan Name

// ...
```

From π to Go : Summation

The first π-calculus expression is $\sum_{i \in I} \pi_i . P_i$.
We can consider the four basic expressions for the sum:

- 0
- $x(y).P$
- $\bar{x}y.P$
- $P + Q$

From π to Go : Nil process

The 0 (or *nil*) process might be represented by a function that return anything.

```
func nil () {
  return
}
```

From π to Go : input and output

For the input and the output actions we have to use the primitive concurrency operators:

$x(y).P$:

```go
func input_act (x Name, y *Name) {
  *y = <-x
}
```

```go
x := make (Name)
// ...
var y Name
input_act(x, &y)
P
```

From π to Go : input and output

For the input and the output actions we have to use the primitive concurrency operators:

$\bar{x}y.P$:

```
func output_act (x Name, y Name) {
  x <- y
}
```

```
x := make (Name)
y := make (Name)
// ...
output_act(x, y)
P
```

From π to Go : $P + Q$

The $P + Q$ process behaves as P or Q and the choice is external to this process. We might have, for example:

```
func Copy(x, z, y, w Name) {
  select {
  case <- x : Succ (x, z, y, w)      // P process
  case <- z : output_act (w, empty)  // Q process
  }
}
```

From π to Go : $!P$

The $!P$ process could be taught as the repetitive call of the process P. A possibility is to use an infinite loop. For $Q|!P$ we could write:

```go
package main
import "fmt" ; import "math/rand" ; import "time"

func main () {
  x := make (chan int)
  // Q process
  go func (c chan int) { c <- 1 } (x)
  for {
     // P process
    go func (c chan int) { y := <- c ; fmt.Println(y, " ") ; c <- (y * 2) } (x)
    time.Sleep(time.Duration(100 + rand.Intn(100)) * time.Millisecond)
  }
}
```

Listing 3: powtwo.go

```
$ go run powtwo.go
1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768 ...
```

From π to Go : $(\nu x)P$

The $(\nu x)P$ process create a new name x and make it private for P. We can use the scope a function in Go to create a private channel. E.g.:

```
func new() {
    c := make (Name)
    P
}
```

Church numerals in the π-calculus

Church numerals are a way to implement natural numbers in λ-calculus:

Natural	Church numeral
0	$\lambda f.\lambda x.x$
1	$\lambda f.\lambda x.f\ x$
2	$\lambda f.\lambda x.f(f\ x)$
3	$\lambda f.\lambda x.f(f(f\ x))$
$\vdots$	$\vdots$
n	$\lambda f.\lambda x.f^{\circ n}\ x$

Every Church numeral encode the natural number as the number that the function f is applied to its argument x.

Church numerals in the π-calculus

In the π-calculus the computational strategy is different from the ones used in the λ-calculus. A possible way to implement a natural number n is to encode it with n output prefixes (and an extra output prefix for the zero):

Natural	π numeral
0	$\bar{z}$[2]
1	$\bar{x}.\bar{z}$
2	$\bar{x}.\bar{x}.\bar{z}$
3	$\bar{x}.\bar{x}.\bar{x}.\bar{z}$
$\vdots$	$\vdots$
n	$(\bar{x}.)^n\ \bar{z}$

[1] $\bar{z}$ and $\bar{x}$ are output prefixes where the received name is not associated to any other name like in $\bar{x}y.P$

π numerals in Go

As presented here [2], these are the processes for implementing addition between π numerals:

$$Add(x_1 z_1, x_2 z_2, yw) \stackrel{\text{def}}{=} x_1.\bar{y}.Add(x_1 z_1, x_2 z_2, yw) + z_1.Copy(x_2 z_2)$$

$$Copy(xz, yw) \stackrel{\text{def}}{=} x.Succ(xz, yw) + z.\bar{w}$$

$$Succ(xz, yw) \stackrel{\text{def}}{=} \bar{y}.Copy(xz, yw)$$

π numerals in Go : *Copy*

For the *Copy* process we have:

$$Copy(xz, yw)x.Succ(xz, yw) + z.\bar{w}$$

```
func Copy(x, z, y, w Name) {
  select {
  case <- x : Succ (x, z, y, w)
  case <- z : output_act (w, empty)
  }
}
```

π numerals in Go : *Succ*

For the *Succ* process we have:

$$Succ(xz, yw) \stackrel{\text{def}}{=} \bar{y}.Copy(xz, yw)$$

```go
func Succ(x, z, y, w Name) {
  output_act (y, empty)
  Copy (x, z, y, w)
}
```

π numerals in Go : *Add*

For the *Add* process we have:

$$Add(x_1 z_1, x_2 z_2, yw) \stackrel{\text{def}}{=} x_1.\bar{y}.Add(x_1 z_1, x_2 z_2, yw) + z_1.Copy(x_2 z_2)$$

```
func Add(x1, z1, x2, z2, y, w Name) {
  select {
  case <- x1 : output_act (y, empty) ; Add (x1, z1, x2, z2, y, w)
  case <- z1 : Copy (x2, z2, y, w)
  }
}
```

DEMO

Bibliography

[1] Milner, R., Parrow, J.G. and Walker, D.J., *A Calculus of Mobile Processes, Parts I and II*, Report ECS-LFCS-89-85 and -86, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1989.

[2] Milner, R., *The Polyadic π-Calculus: a Tutorial*, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1991.

[3] Marinelli, G., *pinumerals: Church numerals in the π-calculus*, University of Camerino, 2019. `https://github.com/marinelli/pinumerals`