Hoare's Communicating Sequential Processes (CSP) '85

$c?x \to P$	Read (synchronous) $c(x).P$
$c!x \to P$	Write (synchronous) $\bar{c}x.P$
$P;Q$	Sequence [redacted] ?
$P \mid\mid Q$	Parallel $P \mid Q$
$P \;[]\; Q$	Sum $P+Q$

where P, Q are guarded.

(either $d?y \to R$ or $d!y \to R$)

JCSP (Welch et al.)

CHANNELS

- Interfaces
  - ChannelInput
  - ChannelOutput
  - Channel
- Implementation
  - One2OneChannel

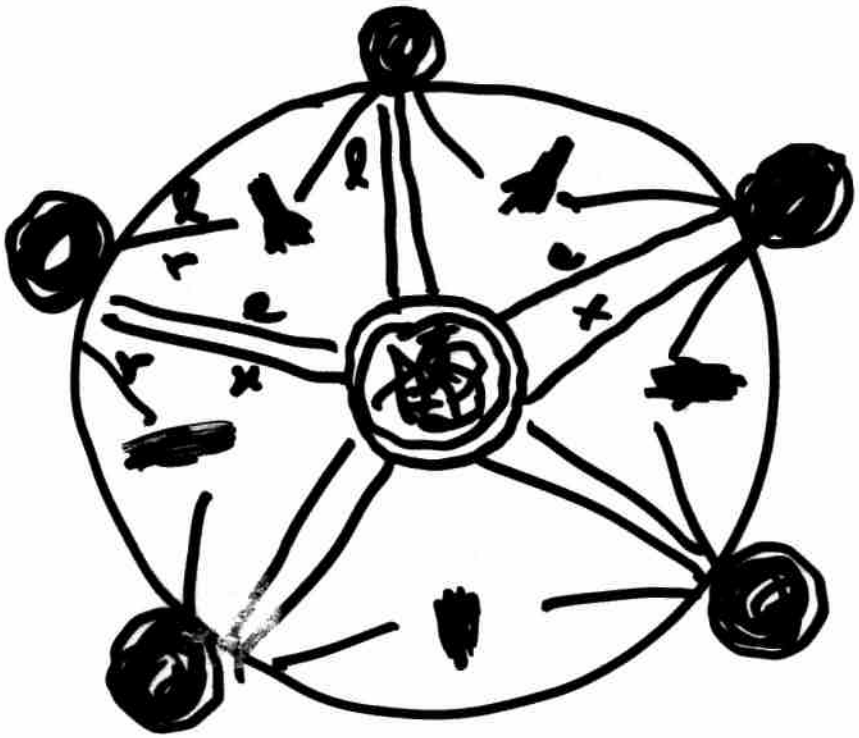
PROCESSES

- Interface
  - CSProcess — run()
- Implementations
  - Parallel
  - Sequence
  - Alternative — select, fairSelect

OTHERS

- Timer, Generate, Skip

Dining Philosophers



Classes

Fork	selects a philosopher to be picked up and put down.
Butler	makes sure at most N-1 philosophers are seated at a time. (uses pairSelect)
Philosopher	loop thinking, entering table, picking forks up, eating, putting forks down and leaving table.

π-Calculus	Pict
0	$()$
$\bar{c}x.0$	$c!x$
$c(x).P$	$c?x = P$
$P \mid Q$	$(P \mid q)$
$(\nu c)P$	$(new\ c\ P)$
$!c(x).P$	$c?*x = P$

Pict

Processes

```
run ()

run (() | () | ())

run ( print! "hello"
    | print! "world" )
```

Channels

```
run ( x?[] = print! "ok" | x![] )

run ( x?z = print! "got z" | x!y )

run ( x!y
    | x?z = z!u
    | y?w = print! "u replaces w" )
```

Process Definitions

def $p[a_1:T_1 \quad a_n:T_n] =$

<def_body>

$p![b_1 \ldots b_n]$

<def_body>$\{^{b_1 \ldots b_n}/_{a_1 \ldots a_n}\}$

Records

new x: ^[a=Bool b=Bool c=[]]

run x![a=false b=true c=[]]

run x?[a=p b=q c=r] =
    if q then print! "T"
    else print! "F"

run x?[a=_ b=q c=_] = ...

[redacted]

run x?r = if 5.a then ...

run x?5@[a=p b=q c=r] =

if 5.b then ...

new y: ^[Bool b=Bool []]

run y![true b=false []]

run y?[p b=q r] = if q then print! "T"
else print! "F"

run y?[_ b=q _] = ...

⇒ Tuples are unlabeled records.

Booleans in the π Calculus

$$True(b) = b(t,f).\bar{t}$$

$$False(b) = b(t,f).\bar{f}$$

$$If(b,t,e) = \bar{b}^{(\nu s)(\nu f)}\langle s,f\rangle.(s.\bar{t} \mid f.\bar{e})$$

Boolean Example

```
new b: ^[^[] ^[]]
new t: ^[]
new f: ^[]

run {- False -}
    b?[t f] = f![]

run {- Test -}
    ( b![t f]
    | t?[] = print! "True"
    | f?[] = print! "False")
```

False

PICT Core Syntax

Program	=	run Proc	Program
Proc	=	Val ! Val	Async. Output
		Val ? Abs	Sync. Input
		( )	Null proc
		(Proc | Proc)	Parallel
		(Dec Proc)	Declaration
		if Val then Proc else Proc	Conditional
Abs	=	Pat = Proc	Process Abstr
Pat	=	Id RType	Var
		[Label Pat ... Label Pat]	Record
		_ RType	Wildcar
		Id RType @ Pat	Layered

PICT CORE SYNTAX CONTINUED

RType = <empty> Omitted type
: Type Explicit type

Type reconstruction (inference) fills in annotations for type checking.

Val = Const Constant
Path Path
[Label Val ... Label Val] Record

Path = Id Variable
Path . Id Record field

Const = String | Char | Int | true | false

Type = ^ Type | Bool | String | Int | Char
| [Label Type ... Label Type]

PICT Core Syntax Continued (III)

Dec	=	new Id : Type	Channel creation
		def Id_1 Abs_1 and ... and Id_n Abs_n	Recursive definition
		type Id = Type	Type Abbreviation
Label	=	<empty>	Anonymous field.
		Id =	Labeled field.

PICT Operational Semantics

Same as π-Calculus (asynchronous version)

e.g. Scope Extrusion:

$$\frac{BV(d) \cap FV(e_2) = \emptyset}{((d\ e_1) \mid e_2) \equiv (d\ (e_1 \mid e_2))}$$

$$((\text{new } y{:}\hat{}[\,]\ x!y) \mid x?z = z![\,])$$

$$\equiv (\text{new } y{:}\hat{}[\,]\ (x!y \mid x?z = z![\,]))$$

e.g. Communication Rule:

$$\frac{\{p \mapsto v\} \text{ defined}}{(x!v \mid x?p = e) \to \{p \mapsto v\}e}$$

$$\{p \mapsto v\} = \{v/p$$

(Note asynchronous output)

Values and Patterns

a channel is a value

if $V_1 .. V_n$ are values then

$[V_1\ V_2\ \ V_n]$ is a value.

[] is the empty tuple value.

```
run ( x?[z] = print! "got z" | x![y])
run ( x?[z1 z2] = print! "ok" ( x![y1 y2
```

pattern

z1 ← y1
z2 ← y2

```
run (x?z = print! "ok" ( x![y1 y2])
```

z ← [y1 y2]

Wildcard pattern

```
run (x?_ = print!"ok" | x![y1 y2])
```

Layered pattern

```
new x:^[Sig Sig]
run (x?z@[z1 z2] = print!"ok" | x![y1 y2])
```

$z \leftarrow [y_1, y_2]$

$z_1 \leftarrow y_1$

$z_2 \leftarrow y_2$

Types

if T is a type

^T is a channel carrying elements of that type

Sig = ^[]

run w?[a] = a?[] = ()

a : Sig ^[]

w : ^[^[]]

run w?a = a?[] = ()

w : ^Sig : ^^[]

type X = T

type Sig = ^[]

Channel Creation

new x : T

Boolean Example Re-visited

```
type Boolean = ^[^[] ^[]]
def tt[b:Boolean] = b?[t _] = t![]
and ff[b:Boolean] = b?[_ f] = f![]
def test[b:Boolean] =
    (new t:^[] new f:^[]
      (b![t f]
      | t?[] = print! "It's true"
      | f?[] = print! "It's false"))

new b:Boolean
run ( ff![b]
    | test![b])
```

Type Refinement

```
type Boolean       = ^[!C] !C]]
type ClientBoolean = ![!C] !C]]
type ServerBoolean = ?[!C] !C]]
```

Subsumption

$$\text{Boolean} < \text{ClientBoolean}$$
$$\text{Boolean} < \text{ServerBoolean}$$

Boolean is a subtype of $\left[\begin{smallmatrix}\text{Client}\\ \text{Server}\end{smallmatrix}\right]$Boolean.

In general, $^T < ?T$

$^T < !T$

Top is the super-type of every other type in PICT: "don't care"-type.

Boolean Example Revisited (III)

```
type Boolean = ^[^[] ^[]]
type ClientBoolean = ![^[] ^[]]
type ServerBoolean = ?[^[] ^[]]

def tt [b:ServerBoolean] = b?[t _] = t![]
and ff [b:ServerBoolean] = b?[_ f] = f![]

def test [b:ClientBoolean] =
    (new t:^[] new f:^[]
       ( b![t f]
       | t?[] = print! "True"
       | f?[] = print! "false" ))

new b:Boolean
run ( ff![b]
    | test![b] )
```

Subtypes

!T	output channel type
?T	input channel type
/T	responsive output channel type

Responsive Output Channels

Channels created by def clauses:

(1) are ALWAYS available to receive values

(2) all communications are received by the SAME receiver (the body of def)

A channel created by def, has the type /T ("responsive channel").

e.g.:

```
new x: ^[/Bool]
def d b: Bool = if b then print! "True"
                     else print! "false"

run x![d]
run x?[a] = a!false
```

prints False.

Responsive Output Channels Continued

/T < !T holds.

e.g.:

```
new y : ^[!Bool]

run y![d]
run y?[a] = a!false
```

prints False.

Many PICT standard libraries use responsive channels.

e.g.: pr /[String /[]]

a responsive channel expecting a String and a responsive channel to signal completion.

Responsive Channel Example

```
def d [] = print! "done"
run pr![" pr..." d]
```

pr...done

Coercing ordinary into responsive (channel)

```
new c: ^[]
run pr![" pr..." (rchan c)]
run c?[] = print! "done"
```

pr...done

Another Example

```
def r z:Int = printi!z
run +![2 3 r]
```

5

Exercise: (1) Type of +?
(2) Use w/ ordinary channel