Hoare's Communicating Sequential Processes (CSP) '85

c?x → P

c!x → P

P; Q

P || Q

P [.] Q

where P, Q are guarded.

either d?y → R or d!y → R

Read (synchronous) c(x).P

Write (synchronous) c.P

Sequence _P

Parallel P || Q

Sum P + Q
JCSP (Welch et al.)

Channels

Interfaces

Channel Input

Channel Output

Channel

Implementation

One2OneChannel

Processes

Interface

CSProcess

run()

Implementations

Parallel

Sequence

Alternative

Others

select

pair

select

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 `fairSelect`)
Philosopher loop thinking, entering table, picking forks up, eating, putting forks down and leaving table.
**Ti-Calculus**

\[ \emptyset \]

\[ \forall x. \emptyset \]

\[ c(x).P \]

\[ P \mid Q \]

\[ (\forall c) P \]

\[ \neg c(x).P \]

**Pict**

\[ () \]

\[ c! x \]

\[ c? x = P \]

\[ (p \mid q) \]

\[ (\text{new } c, p) \]

\[ c? \ast x = P \]
Pict

Processes

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

Channels

run ( x?E = print! "ok" | x!E )
run ( x?z = print! "got z" | x!y )
run ( x!y |
          x?z = z!u |
          y?w = print! "u replaces w" )
Process Definitions

```python
def p[a\,: \, T_1 \, 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

```plaintext
new x : \forall[a=B\, b=B\, c=C]\nrun x !\{a=false \, b=true \, c=C\}
```

```plaintext
run x ?\{a=p \, b=9 \, c=r\} =
\{if q then print! "T" else print! "F"
```

```plaintext
run x ?\{a=- \, b=9 \, c=-\} = ...
```
\[\text{run } x?r = \text{ if } \neg a \text{ then } \ldots\]
\[\text{run } x?5E[a=p\ b=q\ c=r] = \]
\[\text{if } \frac{s\ b}{q} \text{ then } \ldots\]

new y: \^[\text{Bool \ b=Bool \ [3]}\]
\[\text{run } y!\text{[true \ b=false \ [3]}\]
\[\text{run } y?\text{[p \ b=q \ r]} = \text{ if } q \text{ then print! "T" else print! "F"}
\]
\[\text{run } y?\text{[- \ b=q -]} = \ldots\]

\(\Rightarrow\) Tuples are unlabeled records.
Booleans in the Π Calculus

\[ \text{True}(b) = b(t, f). \bar{e} \]

\[ \text{False}(b) = b(t, f). \bar{\bar{e}} \]

\[ \text{If}(b, t, e) = \oslash b <s, t>. (s. \bar{e} \mid t. \bar{\bar{e}}) \]
**BOOLEAN EXAMPLE**

```plaintext
new b: ^[^[C]^[J]]
new t: ^[J]
new f: ^[J]

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

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

False
```
\[
\text{PICT Core Syntax}
\]

\[
\text{Program} = \text{run Proc}
\]

\[
\text{Proc} = \text{Val ! Val} \quad \text{Async. Output}
\]

\[
\text{Val ? Abs} \quad \text{Sync. Input}
\]

\[
(\quad) \quad \text{Null proc}
\]

\[
(\text{Proc} \mid \text{Proc}) \quad \text{Parallel}
\]

\[
(\text{Dec} \text{ Proc}) \quad \text{Declaration}
\]

\[
\text{if Val then Proc else Proc} \quad \text{Conditional}
\]

\[
\text{Abs} = \text{Pat} = \text{Proc} \quad \text{Process Abstraction}
\]

\[
\text{Pat} = \text{Id RType} \quad \text{Var}
\]

\[
[\text{Label Pat} \ldots \text{Label Pat}] \quad \text{Record}
\]

\[
- \text{RType} \quad \text{Wildcard}
\]

\[
\text{Id RType @ Pat} \quad \text{Layered}
\]
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
    def Id, Abs, and... Recurs... definiti
    and Idn Absn
    type Id = Type

Label = <empty>
    Id =

Pict Operational Semantics

Same as Ti-Calculus (asynchronous version,

E.g. Scope Extrusion:

\[ BV(d) \cap FV(e) = \emptyset \]

\[ (((d e_1) | e_2) \equiv (d (e_1, 1 e_2)) \]
\((\text{new } y: \forall [\_] x!y) \mid x?z = z! [\_]\) \\
\equiv \\
(\text{new } y: \forall [\_] (x!y \mid x?z = z! [\_])) \\

\text{e.g. Communication Rule:} \\

\[
\frac{\{p \rightarrow v\} \text{ defined}}{(x!v \mid x?p = e) \rightarrow \{p \rightarrow v\}e}
\]

\[
\{p \rightarrow v\} = \{v/p\}
\]

(Note asynchronous output)
Values and Patterns

a channel is a value

if \( v_1 \ldots v_n \) are values then

\([v_1 \; v_2 \; \ldots \; v_n]\) is a value.

[] is the empty tuple value.

\[
\text{run } ( \; x?[[z]] = \text{print!"got e"} \; | \; x![[y]] )
\]

\[
\text{run } ( \; x?[[z1 \; z2]] = \text{print!"ok"} \; ( \; x![[y1 \; y2]] \; \text{pattern} )
\]

\[
\begin{align*}
& z_1 \leftarrow y_1 \\
& z_2 \leftarrow y_2
\end{align*}
\]

\[
\text{run } ( \; x?z = \text{print!"ok"} \; ( \; x![[y1 \; y2]] )
\]

\[
\begin{align*}
& z \leftarrow [y1 \; y2]
\end{align*}
\]
Wildcard pattern

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

Layered pattern

```
new x:^[Sig Sig]
run (x?z @ [u22] = print! "ok" | x! [u1, u2:
  z <- [y1, y2]
  zs <- y1,
  z2 <- y2
```

Types

```
if T is a type

^T is a channel carrying elements of that type
```

```
Sig = #[]
```
run \ \texttt{w?} [a] = a?[[]] = 0

\texttt{a} : \texttt{Sig} \quad \texttt{w} : \texttt{^[[[]]]}

run \ \texttt{w?} a = a?[[]] = 0

\texttt{w} : \texttt{^Sig} : \texttt{^^[]} \\
\texttt{type X} = \texttt{T}

\texttt{type Sig} = \texttt{^[[]]}

Channel Creation
new x : T
**Boolean Example Re-visited**

```plaintext
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?[^[] = print! "It's true"
    | f?[^[] = print! "It's false")

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

\[
\text{type Boolean} = \text{^[!C] ![C]} \]

\[
\text{type ClientBoolean} = ! \text{^[!C] ![C]} \]

\[
\text{type ServerBoolean} = ? \text{^[!C] ![C]} \]

Subsumption

\[
\text{Boolean} < \text{ClientBoolean} \\
\text{Boolean} < \text{ServerBoolean} \\
\]

Boolean is a subtype of \([\text{Client}]_\text{Server} \text{Boolean}.\)

In general, \(\text{^T < ?T} \)

\(\text{^T < !T} \)

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!C]

and ff [b:ServerBoolean] = b?[ - f] = f!C]

def test [b:ClientBoolean] =

(new t:^[ ] new f:^[ ]

(b! [t f]

| t?[ ] = println! "True"

| f?[ ] = println! "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 1-> ("responsive channel").

e.g.:

new x: n[/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.:

\[
\begin{align*}
\text{new } y : & \ ^{[!} \text{Bool}] \\
\text{run } y ! [d] \\
\text{run } y ? [a] = a ! \text{false}
\end{align*}
\]

prints False.

Many PLC standard libraries use responsive channels.

e.g.:

\[
\text{pr } /[\text{String } /C] ]
\]

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 Chan

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

pr...done

Another Example

def r z:int = print! z
run +! [2 3 r]

5

Exercise: (1) Type op +?
(2) Use w/ ordinary chan
   (21)