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)
ex.P
Sequence ?
Parallel P ? Q
Sum P + Q
JCSP (Welch et al.)

Channels
  Interfaces
    Channel Input
    Channel Output
    Channel
  Implementation
    One2OneChannel

Processes
  Interface
    CSPProcess
    run()

Implementations
  Parallel
  Sequence
  Alternative
    select
    pairSelect

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.
Processes

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

Channels

run (x?[z] = 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:T1 a1:T1] =
    <def-body>

p([b, ... bn])
    <def-body> { b1,...,bn / a1,...,an }

Records
new x : ^{a=Bool b=Bool c=C3}
run x! [a=false b=true c=C3]

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

run x?[a=-- b=¢ c=¢] = ...
\[
\begin{align*}
\text{run } x?r &= \text{ if } r \neq a \text{ then } ... \\
\text{run } x?5 &\in [a=p \ b=q \ c=r] = \\
&\text{ if } 5 \neq b \text{ then } ...
\end{align*}
\]

new y: \( ^\uparrow [\text{Bool \ } b=\text{Bool} \ [J] \]

run y! [true \ b=false \ [J] 

run y? [p \ b=q \ r] = \text{ if } 5 \text{ then print! "T" else print! "F" }

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

\Rightarrow \text{Tuples are unlabeled records.}
Boolean Example

new b: \[ ^C \] ^[ ]
new t: \[ ^C \]
new f: \[ ^C \]

run { -False- }
    b?[^±f] = f![]

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

False
PICT Core Syntax

Program = run Proc
Proc = Val ! Val
       Val ? Abs
       ()
       (Proc | Proc)
       (Dec Proc)
       if Val then Proc else Proc
Abs = Pat = Proc
Pat = 1d RTtype
     [Label Pat ... Label Pat]
     - RTtype
     1d RTtype @ Pat

Async. Output
Sync. Input
Null proc
Parallel
Declaration
Conditional
Process Abs
Var
Record
Wildcard
Layered
Pict Core Syntax Continued

\[ \text{RType} \quad = \quad \langle \text{empty} \rangle \quad \text{Omitted type} \]

\[ \quad : \quad \text{Type} \quad \text{Explicit type} \]

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

\[ \text{Val} \quad = \quad \text{Const} \quad \text{Constant} \]

\[ \quad \text{Path} \quad \text{Path} \]

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

\[ \text{Path} \quad = \quad \text{Id} \quad \text{Variable} \]

\[ \quad \text{Path} \cdot \text{Id} \quad \text{Record field} \]

\[ \text{Const} \quad = \quad \text{String} \mid \text{Char} \mid \text{Int} \mid \text{true} \mid \text{false} \]

\[ \text{Type} \quad = \quad ^\wedge \text{Type} \mid \text{Bool} \mid \text{String} \mid \text{Int} \mid \text{Char} \]

\[ \quad [\text{Label Type} \ldots \text{Label Type}] \]
PICT Core Syntax Continued (III)

\[
\text{Dec} = \text{new } \text{Id} : \text{Type} \quad \text{Charact}
\]

\[
\text{def } \text{Id} : \text{Abs}, \text{ and... Reused}
\]

\[
\text{and } \text{Idn} \text{ Absn} \quad \text{definition}
\]

\[
\text{Type} \quad \text{Abbreviation}
\]

\[
\text{type } \text{Id} = \text{Type}
\]

Label = <empty>

\[
\text{Anonymous yield,}
\]

\[
\text{Labeled yield}
\]

PICT Operational Semantics

Same as TT-Calculus (asynchronous version)

e.g. Scope Extrusion:

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

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

\[((\text{new } y: ^*[]) (x!y \mid x?z = z !^*[[]]))\]

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.

run ( $x?E2 = \text{print!"got e" | x!E2} )

run ( $x?[E1 E2] = \text{print!"ok" | x!E2} )$

pattern

$z1 \leftarrow y1$
$z2 \leftarrow y2$

run ( $x?z = \text{print!"ok" | x!E1 E2} )$

$z \leftarrow [y1 y2]$
Wildcard pattern

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

Layered pattern

```run (x? z @ [u z2] = print! "ok" | x! [u1 y2]`
`  z @ [y1 y2]`
`  z1 @ y1`
`  z2 @ y2```

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

```java
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

\[
Boolean < ClientBoolean
\]

\[
Boolean < ServerBoolean
\]

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

In general,

\[
\begin{align*}
\wedge T &< ?T \\
\wedge T &< \! T
\end{align*}
\]

Top is the super-type of every other type in PICT: "don't care"-type.
Boolean Example Revisited (III)

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

new b: Boolean
run ( 
 | test! [b]
 | 1
)
SUTYPES

!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.:

```plaintext
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 PLCT standard libraries use responsive channels.

e.g.:  

pr ![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