*** Theory of totally ordered sets.

fth TOSET is protecting BOOL .
  sort Elt .
  op _>_ : Elt Elt -> Bool .

vars X Y Z : Elt .

eq X > X = false .           *** irreflexivity
ceq X > Z = true
  if X > Y = true \ Y > Z = true [nonexec] .
  *** transitivity
ceq X = Y if X > Y = false \ Y > X = false
  [nonexec] . *** trychotomy

op eq : Elt Elt -> Bool [comm] . *** equality

eq eq(X,X) = true .
ceq eq(X,Y) = false if X > Y = true .
endfth

*** CINNI is Mark-Oliver Stehr's calculus of explicit
*** substitution for any binding operator; here is the
*** pi-Calculus instance of CINNI explicit substitution:

mod PI-CINNI{N :: TOSET} is  protecting NAT .
  sorts Chan{N} NeProc{N} Proc{N} . *** Channel and Process
  subsorts NeProc{N} < Proc{N} .

op _{=} : N$Elt Nat ->  Chan{N} [ctor] .

op 0 : -> Proc{N} [ctor] .       *** null process

op _l_ : Proc{N} Proc{N} -> Proc{N}
  [ctor assoc comm id: 0] .
op _l_ : NeProc{N} NeProc{N} -> NeProc{N}
op _+_ : Proc{N} Proc{N} -> Proc{N}  
  [ctor frozen assoc comm id: 0] .  
 *** parallel process composition

op _+_ : NeProc{N} NeProc{N} -> NeProc{N}  
  [ctor frozen assoc comm id: 0] .  
 *** non-deterministic choice

  *** replication

  *** new-abstraction [name restriction]

op in_[__]._ : Chan{N} N$Elt Proc{N} -> NeProc{N}  
  [ctor frozen] .  
 *** synchronous message reception with name in-abstraction

op out_<_>._ : Chan{N} Chan{N} Proc{N} -> NeProc{N}  
  [ctor frozen] .  
 *** synchronous message send along a channel

op if : Bool Chan{N} Chan{N} -> Chan{N} .  
  *** if-then-else

sort Sub{N} .  *** substitutions sort

op [__:=_] : N$Elt Chan{N} -> Sub{N}  [ctor] .  
  *** substitution

op [^_] : N$Elt -> Sub{N}  [ctor] .  *** shift

op [^__] : N$Elt Sub{N} -> Sub{N}  [ctor] .  *** lift
op __ : Sub{N} Chan{N} -> Chan{N} [frozen].
     *** substitution applied to channel

op __ : Sub{N} Proc{N} -> Proc{N} [frozen].
     *** substitution applied to process

vars X Y Z : N$Elt .
vars CX CY CZ : Chan{N} .
vars N M P R : Proc{N} .
vars P' R' : NeProc{N} .
var S : Sub{N} .
vars n m : Nat .

*** equations for if-then-else

eq if(true,CX,CY) = CX .
eq if(false,CX,CY) = CY .

*** substitution application equations for non-binders

eq [1] : S 0 = 0 .
eq [2] : S (P' | R') = (S P') | (S R') .
eq [3] : S (P' + R') = (S P') + (S R') .
          out (S CX) < S CZ >. (S M ) .

*** substitution application equations for
*** in- and new-binders

          in (S CX) [ Y ]. ([^ Y S] M ) .

*** substitution application equations for channels

eq [8] : [X := CZ] (Y{0}) = if (eq(X,Y),CZ,Y{0}) .
eq [9] : [X := CZ] (Y{s(n)}) =
          if(eq(X,Y),Y{n},Y{s(n)}) .
eq [10] : \[ ^X \] (Y\{n\}) = \text{if}(\text{eq}(X,Y), Y\{s(n)\}, Y\{n\}) \).

eq [11] : \[ ^X \mathcal{S} \] (Y\{0\}) = 
\text{if}(\text{eq}(X,Y), Y\{0\}, \[ ^X \] (S (Y\{0\}))) \).

eq [12] : \[ ^X \mathcal{S} \] (Y\{s(n)\}) = 
\text{if}(\text{eq}(X,Y), \[ ^Y \] (S (Y\{n\})), \[ ^X \] (S (Y\{s(n)\}))) \).

*** equality of processes is given by the ACU axioms for
*** _+_ and _|_, plus the following equations for
*** new-restricted processes

eq [13] : \text{new} \ [X]\{0\} = 0 .

eq [14] : \text{new} \ [X]\{P'\} \ | \ R' = \text{new} \ [X]\{P' \ | \ \[ ^X \] R'\} .

*** \[ ^X \] avoids name capture

c\eq [15] : \text{new} \ [X]\{\text{new} \ [Y]\{P\}\} = \text{new} \ [Y]\{\text{new} \ [X]\{P\}\}
\text{if } X > Y .

eq [16] : ! 0 = 0 . *** not given by Milner, but natural

*** Stehr proves that above equations [1]-[16] (he does
*** not treat [3,5,16], but they should cause no problem)
*** are confluent. They can be proved terminating.

*** The key semantic rule is the following synchronous
*** communication rule between processes:

rl [synch-comm] :
(N + \text{out} CX < CZ >. P) \ | \ (M + \text{in} CX \ [ Y ]. R)
\Rightarrow P \ | \ [Y := CZ] R .

*** Since the process equivalence \! P' = P' \ | \ ! P' would:
*** (i) be non-terminating as a left-to-right equation,
*** or (ii) be terminating but PREVENT desired
*** computations if oriented as: P' \ | \ ! P' = ! P' we
*** instead add it as the rewrite rule:

rl [repl] : ! P' \Rightarrow P' \ | \ ! P' .
This, together with the fact that we do not deal with
alpha-conversion means that process equivalence is less
abstract than in Milner's formalization, but: (i) using
CINNI, alpha-conversion issues mostly evaporate; and
(ii) this has the advantage of yielding an effective
notion of process equivalence by confluent and
terminating equations executable in Maude.

Of course, even as a rule, \texttt{repl} is obviously
non-terminating. But: (i) this is no problem if we
use BOUNDED rewrite or search commands; and (ii) if
so desired, we can use Maude's strategy language to
execute \texttt{repl} according to a practical strategy that
will prevent "runaway" useless applications of \texttt{repl}.

The following module defines a total order on Qids as the standard lexicographic order on
their associated strings.

\begin{verbatim}
endm

\texttt{fmod QID> is protecting QID .}
\texttt{op _>_: Qid Qid -> Bool .}
\texttt{vars I J : Qid .}
\texttt{eq I > J = string(I) > string(J) .}
endfm

\texttt{view Qid from TOSET to QID> is}
\texttt{sort Elt to Qid .}
\texttt{vars X Y : Elt .}
\texttt{op eq(X,Y) to term (X == Y) .}
endv
\end{verbatim}
mod PI-CINNI-QID is
protecting PI-CINNI{Qid}.
endm

rewrite (out 'x{0} < 'y{0} >. 0) |
(in 'x{0} ['u].(out 'u{0} < 'v{0} >. 0))) .

***(
result NeProc{Qid}: out 'y{0} < 'v{0} >. 0)
)

red new['z]{{in 'z{0}['w]. out 'w{0} < 'y{0} >. 0) |
out 'x{0} < 'z{0} >. 0} |
(in 'x{0}['u]. out 'u{0} < 'v{0} >. 0) |
out 'x{0} < 'y{0} >. 0}.

***(
result NeProc{Qid}:
new['z]{{in 'x{0}['u]. out 'u{0} < 'v{0} >. 0) |
(in 'z{0}['w]. out 'w{0} < 'y{0} >. 0) |
(out 'x{0} < 'y{0} >. 0) |
out 'x{0} < 'z{0} >. 0}
)

search new ['z]{{out 'x{0} < 'y{0} >. 0) |
(in 'z{0} ['w].(out 'w{0} < 'y{0} >. 0)) |
(in 'x{0} ['u].(out 'u{0} < 'v{0} >. 0)) |
out 'x{0} < 'z{0} >. 0}
=>+ P:Proc{Qid}.

***(
Solution 1 (state 1)

P:Proc{Qid} -->
new['z]{(in 'z{0}['w]. out 'w{0} < 'y{0} >. 0) |
    (out 'x{0} < 'z{0} >. 0) |
    out 'y{0} < 'v{0} >. 0}

Solution 2 (state 2)

P:Proc{Qid} -->
new['z]{(in 'z{0}['w]. out 'w{0} < 'y{0} >. 0) |
    (out 'x{0} < 'y{0} >. 0) |
    out 'z{0} < 'v{0} >. 0}

Solution 3 (state 3)

P:Proc{Qid} -->
new['z]{(out 'v{0} < 'y{0} >. 0) | out 'x{0} < 'y{0} >. 0}

No more solutions.
)

show search graph .

***(
state 0, NeProc{Qid}:
new['z]{(in 'x{0}['u]. out 'u{0} < 'v{0} >. 0) |
    (in 'z{0}['w]. out 'w{0} < 'y{0} >. 0) |
    (out 'x{0} < 'y{0} >. 0) |
    out 'x{0} < 'z{0} >. 0}
arc 0 ===> state 1 (rl [syn-comm] .)

arc 1 ===> state 2 (rl [syn-comm] .)

state 1, NeProc{Qid}:

new['z]{(in 'z{0}['w]. out 'w{0} < 'y{0} >. 0) |
(out 'x{0} < 'z{0} >. 0) |
out 'y{0} < 'v{0} >. 0)}

state 2, NeProc{Qid}:

new['z]{(in 'z{0}['w]. out 'w{0} < 'y{0} >. 0) |
(out 'x{0} < 'y{0} >. 0) |
out 'z{0} < 'v{0} >. 0)}

arc 0 ===> state 3 (rl [syn-comm] .)

state 3, NeProc{Qid}:

new['z]{(out 'v{0} < 'y{0} >. 0) |
out 'x{0} < 'y{0} >. 0)}

)

*** let us consider a similar example, where now one of the
*** processes has non-deterministic choice:

search
new ['z]{((out 'x{0} < 'y{0} >. 0) +
(in 'z{0} ['w].(out 'w{0} < 'y{0} >. 0)))) |
(in 'x{0} ['u].(out 'u{0} < 'v{0} >. 0)) |
out 'x{0} < 'z{0} >. 0}

  =>+ P:Proc{Qid} .
***
Solution 1 (state 1)

P:Proc{Qid} -->

new['z]{(out 'x{0} < 'z{0} >. 0) |
        out 'y{0} < 'v{0} >. 0)}

Solution 2 (state 2)

P:Proc{Qid} -->

new['z]{((in 'z{0}[w]. out 'w{0} < 'y{0} >. 0) +
        out 'x{0} < 'y{0} >. 0) |
        out 'z{0} < 'v{0} >. 0)}

Solution 3 (state 3)

P:Proc{Qid} -->

new['z]{out 'v{0} < 'y{0} >. 0}

No more solutions.
)