Soundess of Derived Rules

Kristina Sojakova Mihai Codescu

[1]

1 Core Rules

1.1 Expressions

The typing predicate is

```
op typeOf
  : Signature TypeContext IPDLExpression
  -> IPDLType .
```

and the configuration has the form

```
op expConfig
  : Signature TypeContext IPDLType IPDLExpression
  -> ExpConfig [ctor] .
```

which means that the expression equality judgement

$$\Gamma \vDash e_1 = e_2 : T$$

translates to

```
expConfig(Sigma, Gamma, T, e1)
=>
expConfig(Sigma, Gamma, T, e2)
```

Equality rules are below. REFL, TRANS hold by the properties of rewriting in Maude. AXIOM is not needed as a rule, because we already write an axiom as

```
expConfig(Sigma, Gamma, T, e1)
=>
expConfig(Sigma, Gamma, T, e2)
```

[1]This project is funded through NGI Zero Core (`https://nlnet.nl/core`), a fund established by NLnet (`https://nlnet.nl`) with financial support from the European Commission's Next Generation Internet (`https://ngi.eu`) program. Learn more at the NLnet project page (`https://nlnet.nl/project/IPDL-II`).

SYM :

```
crl [exp-sym] :
expConfig(Sigma, Gamma, T, e2)
=>
expConfig(Sigma, Gamma, T, e1)
if
expConfig(Sigma, Gamma, T, e1)
=>
expConfig(Sigma, Gamma, T, e2)
/\ typeOf(Sigma, Gamma, e1) == T .
```

SUBST :

```
crl [exp-subst] :
expConfig(Sigma, Gamma1, T, e1')
=>
expConfig(Sigma, Gamma1, T, applySubst(e2, theta))
if
expConfig(Sigma, Gamma2, T, e1)
=>
expConfig(Sigma, Gamma2, T, e2)
/\
typeOf(Sigma, Gamma2, e1) == T
/\
e1' == applySubst(e1, theta^)
```

APP-CONG :

```
crl[app-cong] :
expConfig(Sigma (f : T1 -> T2), Gamma, T2, ap f e1)
=>
expConfig(Sigma (f : T1 -> T2), Gamma, T2, ap f e2)
if
expConfig(Sigma (f : T1 -> T2), Gamma, T1, e1)
=>
expConfig(Sigma (f : T1 -> T2), Gamma, T1, e2) .
```

PAIR-CONG :

```
crl [pair-cong] :
expConfig(Sigma, Gamma, T1 * T2, pair(M1, M2))
=>
expConfig(Sigma, Gamma, T1 * T2, pair(M3, M4))
if
expConfig(Sigma, Gamma, T1, M1)
=>
```

```
expConfig(Sigma, Gamma, T1, M3)
/\
expConfig(Sigma, Gamma, T2, M2)
=>
expConfig(Sigma, Gamma, T2, M4)
.
```

FST-CONG :

```
crl [fst-cong] :
expConfig(Sigma, Gamma, T1, fst(T1, T2, M1))
=>
expConfig(Sigma, Gamma, T1, fst(T1, T2, M2))
if
expConfig(Sigma, Gamma, T1 * T2, M1)
=>
expConfig(Sigma, Gamma, T1 * T2, M2)
.
```

SND-CONG :

```
crl [snd-cong] :
expConfig(Sigma, Gamma, T2, snd(T1, T2, M1))
=>
expConfig(Sigma, Gamma, T2, snd(T1, T2, M2))
if
expConfig(Sigma, Gamma, T1 * T2, M1)
=>
expConfig(Sigma, Gamma, T1 * T2, M2)
.
```

UNIT-EXT :

```
crl [unit-ext] :
expConfig(Sigma, Gamma, unit, M1)
=>
expConfig(Sigma, Gamma, unit, ())
if typeOf(Sigma, Gamma, M1) == unit .
```

FST-PAIR :

```
crl [fst-pair] :
expConfig(Sigma, Gamma, T1 * T2, fst pair(M1, M2))
=>
expConfig(Sigma, Gamma, T1, M1)
if typeOf(Sigma, Gamma, M1) == T1
/\ typeOf(Sigma, Gamma, M2) == T2
 .
```

SND-PAIR :

```
crl [snd-pair] :
expConfig(Sigma, Gamma, T1 * T2, snd pair(M1, M2))
=>
expConfig(Sigma, Gamma, T2, M2)
 if typeOf(Sigma, Gamma, M1) == T1
/\ typeOf(Sigma, Gamma, M2) == T2
 .
```

PAIR-EXT :

```
crl [pair-ext] :
expConfig(Sigma, Gamma, T1 * T2, pair(fst M, snd M))
=>
expConfig(Sigma, Gamma, T1 * T2, M)
if typeOf(Sigma, Gamma, M) == T1 * T2
 .
```

Comments:

- if we do not annotate the projections with their types, we would have to write

  ```
  crl [fst-cong] :
  expConfig(Sigma, Gamma, T1, fst(M1))
  =>
  expConfig(Sigma, Gamma, T1, fst(M2))
  if
  expConfig(Sigma, Gamma, T1 * T2, M1)
  =>
  expConfig(Sigma, Gamma, T1 * T2, M2)
  [nonexec]
  .
  ```

  and specify a type for T2 when applying the rule, which is inconvenient.

1.2 Reactions

The typing predicate is

```
op typeOf
: Signature ChannelContext TypeContext
  Set{CNameBound} Set{BoolTerm} Reaction
-> IPDLType
```

where the first set argument is the set of inputs and the second set argument is the set of assumptions on indices, for families of protocols, and hypotheses of the type $N + 1$ is honest.

The configuration has the form

```
op rConfig
  : Signature ChannelContext TypeContext
    Reaction Set{CNameBound} Set{BoolTerm}
    IPDLType
 -> ReactionConfig [ctor] .
```

which means that the reaction equality judgement

$$\Delta; \Gamma \vDash R_1 = R_2 : I \to T$$

translates to

```
rConfig(Sigma, Delta, Gamma, R1, I, A, T)
=>
rConfig(Sigma, Delta, Gamma, R2, I, A, T)
```

where A is the set of assumptions (new argument).

Equality rules are below. Again, REFL, TRANS, SUBST hold by the properties of rewriting in Maude and AXIOM is not needed as a rule. To avoid name clashes, these rules have their names in lowercaps.

SYM :

```
crl [sym] :
rConfig(Sigma, Delta, Gamma, R2, I, A, T)
=>
rConfig(Sigma, Delta, Gamma, R1, I, A, T)
if rConfig(Sigma, Delta, Gamma, R1, I, A, T)
   =>
  rConfig(Sigma, Delta, Gamma, R2, I, A, T)
[nonexec] .
```

INPUT-UNUSED :

```
crl [input-unused] :
rConfig(Sigma, Delta, Gamma, R1, (I, chn c), A, T)
=>
rConfig(Sigma, Delta, Gamma, R2, (I, chn c), A, T)
if
rConfig(Sigma, Delta, Gamma, R1, I, A, T)
=>
rConfig(Sigma, Delta, Gamma, R1, I, A, T) .
```

EMBED :

```
crl [embed] :
```

```
rConfig(Sigma, Delta1, Gamma, R1', I', A, T)
=>
rConfig(Sigma, Delta1, Gamma,
        embedReaction(R2, phi),
        I', A, T)
if
rConfig(Sigma, Delta2, Gamma, R1, I, A, T)
=>
rConfig(Sigma, Delta2, Gamma, R2, I, A, T)
/\
I' == embedIO(I, phi)
/\
R1' == embedReaction(R1, phi)
[nonexec]
.
```

where `phi : Delta1 -> Delta2` is an embedding, `embedIO(I, phi)` stands for $\phi^*(I)$ and `embedReaction(R, phi)` stands for $\phi^*(R)$.

CONG-RET :

```
crl [cong-ret] :
rConfig(Sigma, Delta, Gamma, return M1, I, A, T)
=>
rConfig(Sigma, Delta, Gamma, return M2, I, A, T)
if
expConfig(Sigma, Gamma, T, M1)
=>
expConfig(Sigma, Gamma, T, M2)
.
```

CONG-SAMP :

```
crl [cong-samp] :
rConfig(Sigma (d : T1 ->> T2), Delta, Gamma,
        samp (d < M1 >), I, A, T)
=>
rConfig(Sigma (d : T1 ->> T2), Delta, Gamma,
        samp (d < M2 >), I, A, T)
if
expConfig(Sigma (d : T1 ->> T2), Gamma, T1, M1)
=>
expConfig(Sigma (d : T1 ->> T2), Gamma, T1, M2)
.
```

CONG-IF :

```
crl [cong-if] :
rConfig(Sigma, Delta, Gamma,
        if M1 then R1 else R2, I, A, T)
=>
rConfig(Sigma, Delta, Gamma,
        if M2 then R3 else R4, I, A, T)
if
rConfig(Sigma, Delta, Gamma, R1, I, A, T)
=>
rConfig(Sigma, Delta, Gamma, R3, I, A, T)
/\
rConfig(Sigma, Delta, Gamma, R2, I, A, T)
=>
rConfig(Sigma, Delta, Gamma, R4, I, A, T)
/\
expConfig(Sigma, Gamma, bool, M1)
=>
expConfig(Sigma, Gamma, bool, M2) .
```

CONG-BIND :

```
crl [cong-bind] :
    rConfig(Sigma, Delta, Gamma,
            x : T1 <- R1 ; R2,
            I , A, T2)
    =>
    rConfig(Sigma, Delta, Gamma,
            x : T1 <- R3 ; R4,
            I, A, T2)
    if
    rConfig(Sigma, Delta, Gamma, R1, I, A, T1)
    =>
    rConfig(Sigma, Delta, Gamma, R3, I, A, T1)
    /\
    rConfig(Sigma, Delta, Gamma (x : T1),
            R2, I, A, T2)
    =>
    rConfig(Sigma, Delta, Gamma (x : T1),
            R4, I, A, T2) .
    .
```

SAMP-PURE :

```
crl [samp-pure] :
    rConfig(Sigma, Delta, Gamma,
            x : T1 <- samp D ; R,
            I, A, T2)
    =>
    rConfig(Sigma, Delta, Gamma,
            R,
            I, A, T2)
if typeOf(Sigma, Gamma, D) == T1
/\ typeOf(Sigma, Delta, Gamma, I, A, R) == T2
.
```

READ-DET :

```
crl [read-det] :
    rConfig(Sigma, Delta, Gamma,
                   x : T1 <- read i ;
                   y : T1 <- read i ;
                   R , I, A, T2)
    =>
    rConfig(Sigma, Delta, Gamma,
                   x : T1 <- read i ;
                   (R [y / x]), I, A, T2)
if  isElemB(i, I, A)
/\ elem (chn i) T1 Delta A
/\  typeOf(Sigma, Delta, Gamma (x : T1) (y : T1),
          I, A, R) == T2
.
```

where `isElemB(i, I, A)` checks that `i` is in `I` and `elem (chn i) T1 Delta A` checks that `(i : T1)` is in `Delta`. Both methods take into account that `i` may appear in `I` and `Delta` as a part of a family.

IF-LEFT :

```
crl [if-left] :
    rConfig(Sigma, Delta, Gamma,
            if True then R1 else R2, I, A, T)
    =>
    rConfig(Sigma, Delta, Gamma, R1, I, A, T)
if
    typeOf(Sigma, Delta, Gamma, I, A, R1) == T
    /\
    typeOf(Sigma, Delta, Gamma, I, A, R2) == T
.
```

IF-RIGHT :

```
crl [if-right] :
    rConfig(Sigma, Delta, Gamma,
            if False then R1 else R2, I, A, T)
    =>
    rConfig(Sigma, Delta, Gamma, R2, I, A, T)
if
    typeOf(Sigma, Delta, Gamma, I, A, R1) == T
    /\
    typeOf(Sigma, Delta, Gamma, I, A, R2) == T
.
```

IF-EXT : We would write the rule as

```
crl [if-ext] :
    rConfig(Sigma, Delta, Gamma, R [b / M], I, A, T)
    =>
    rConfig(Sigma, Delta, Gamma,
      if M then (R [b / True]) else (R [b / False]),
      I, A, T)
    if
    typeOf(Sigma, Gamma, M) == bool
    /\
    typeOf(Sigma, Delta, Gamma (b : bool),
           I, A, R) == T .
```

but Maude cannot handle these kind of rules. What we can write is a version where `M` is a variable:

```
crl [if-intro-ext] :
    rConfig(Sigma, Delta, Gamma (q : bool),
            R, I, A, T)
    =>
    rConfig(Sigma, Delta, Gamma (q : bool),
            if q then (R[q / True])
                 else (R[q / False]), I, A, T)
  if typeOf(Sigma, Delta, Gamma (q : bool),
            I, A, R) == T
.
```

When we apply the rule we might need to mention which `q` should be used, as there could be more than one in `Gamma`. We would not be able to write the rule in the reverse direction, but we can use this rule under a `sym`.

RET-BIND :

```
crl [ret-bind] :
    rConfig(Sigma, Delta, Gamma,
            x : T1 <- return M ; R , I , A, T2)
    =>
    rConfig(Sigma, Delta, Gamma,
            R [x / M], I, A, T2)
    if
    typeOf(Sigma, Gamma, M) == T1
    /\
    typeOf(Sigma, Delta, Gamma (x : T1),
           I, A, R) == T2 .
```

BIND-RET :

```
crl [bind-ret] :
    rConfig(Sigma, Delta, Gamma,
            x : T <- R ; return x, I , A, T)
    =>
    rConfig(Sigma, Delta, Gamma, R, I, A, T)
    if
    typeOf(Sigma, Delta, Gamma, I, A, R) == T .
```

BIND-BIND :

```
crl [bind-bind] :
    rConfig(Sigma, Delta, Gamma,
           x2 : T2 <- (x1 : T1 <- R1 ;
                        R2) ;
                       R3, I, A, T3)
    =>
    rConfig(Sigma, Delta, Gamma,
           x1 : T1 <- R1 ;
                       (x2 : T2 <- R2 ;
                        R3), I, A, T3)
if
    typeOf(Sigma, Delta, Gamma, I, A, R1) == T1
    /\
    typeOf(Sigma, Delta, Gamma (x1 : T1),
           I, A, R2) == T2
    /\
    typeOf(Sigma, Delta, Gamma (x2 : T2),
           I, A, R3) == T3
.
```

EXCH :

```
crl [exchange] :
    rConfig(Sigma, Delta, Gamma,
            x1 : T1 <- R1 ;
            x2 : T2 <- R2 ;
            R, I, A, T3)
    =>
    rConfig(Sigma, Delta, Gamma,
            x2 : T2 <- R2 ;
            x1 : T1 <- R1 ;
            R, I, A, T3)
if
    typeOf(Sigma, Delta, Gamma, I, A, R1) == T1
    /\
    typeOf(Sigma, Delta, Gamma, I, A, R2) == T2
    /\
    typeOf(Sigma, Delta, Gamma (x1 : T1) (x2 : T2),
           I, A, R) == T3
.
```

1.3 Protocols

The typing predicate is

```
op typeOf
 : Signature ChannelContext
   Set{CNameBound} Set{BoolTerm} Protocol
 -> Bool
```

where the first set argument is the set of inputs and the second set argument is the set of assumptions on indices.

The configuration has the form

```
op pConfig
 : Signature ChannelContext Protocol
   Set{CNameBound} Set{CNameBound} Set{BoolTerm} ->
   ProtocolConfig [ctor] .
```

which means that the protocol equality judgement

$$\Delta \vDash P_1 = P_2 : I \to O$$

translates to

```
pConfig(Sigma, Delta, P1, I, O, A)
=>
pConfig(Sigma, Delta, P2, I, O, A)
```

where A is the set of assumptions (new argument).

Equality rules are below. Again, REFL, TRANS, SUBST hold by the properties of rewriting in Maude and AXIOM is not needed as a rule. Moreover COMP-ASSOC and COMP-COMM are not needed, as we have defined the parallel composition as a commutative and associative operator. This also means that having both a -LEFT and a -RIGHT version for ABSORB, FOLD-IF and CONG-COMP is not needed, and we should keep just one.

SYM :

```
crl [SYM] :
    pConfig(Sigma, Delta2, P2, I, O2, A)
    =>
    pConfig(Sigma, Delta1, P1, I, O1, A)
    if
    pConfig(Sigma, Delta1, P1, I, O1, A)
    =>
    pConfig(Sigma, Delta2, P2, I, O2, A)
    /\ Delta1 equiv Delta2
    /\ O1 equiv O2
    [nonexec] .
```

where the `equiv` relations hold if the arguments are equal modulo splitting. Splitting of families of protocols means that e.g. the family `F[< X < Y < N + 2 ]` is equivalent with `F[< X < Y < N + 1 ]` and `F[< X < Y = N + 1 ]`.

INPUT-UNUSED :

```
crl [INPUT-UNUSED] :
 pConfig(Sigma, Delta, P1, (I, chn c), O, A)
 =>
 pConfig(Sigma, Delta, P2, (I, chn c), O, A)
 if
 pConfig(Sigma, Delta, P1, I, O, A)
 =>
 pConfig(Sigma, Delta, P2, I, O, A) .
```

CONG-REACT :

```
crl [CONG-REACT] :
    pConfig(Sigma, Delta, cn ::= R, I, bn, A)
    =>
    pConfig(Sigma, Delta, cn ::= R', I, bn, A)
    if
    rConfig(Sigma, Delta, emptyTypeContext, R,
            (I, chn cn), A,
```

```
            typeInCtx(chn cn, A, Delta))
    =>
    rConfig(Sigma, Delta, emptyTypeContext, R',
            (I, chn cn), A, T)
    /\ T == typeInCtx(chn cn, A, Delta)
    /\ not (isElemB(chn cn, I, A))
    .
```

where `typeInCtx(chn cn, A, Delta)` gives us the the type of `cn` in `Delta`, possibly by looking at the family that `cn` is a member of, and we also test that `chn cn` is not an input channel or member of an input family.

CONG-COMP-LEFT :

```
crl [CONG-COMP-LEFT] :
    pConfig(Sigma, Delta1, P1 || Q, I, O, A)
    =>
    pConfig(Sigma, Delta2, P2 || Q, I,
            union(getOutputs(P2), getOutputs(Q)), A)
    if
    pConfig(Sigma, Delta1, P1,
            union(I, getOutputs(Q)),
            getOutputs(P1), A)
    =>
    pConfig(Sigma, Delta2, P2, I1, O2, A)
    /\ O2 == getOutputs(P2)
    /\ I1 == union(I, getOutputs(Q))
    /\ typeOf(Sigma, Delta2,
              union(I, getOutputs(P2)),
              A, Q)
    /\ Delta1 equiv Delta2
    /\ O equiv
       (union(getOutputs(P2), getOutputs(Q)))
.
```

where we must allow splitting and `getOutputs(P)` gives us the outputs of the protocol `P`. Note that Maude won't let us write `getOutputs(P2)` after the `=>` sign. If we were to do that, it would look for an exact syntactic match and it would fail.

CONG-COMP-RIGHT :

```
crl [CONG-COMP-RIGHT] :
    pConfig(Sigma, Delta1, Q || P1,
            I, O, A)
    =>
```

```
        pConfig(Sigma, Delta2, Q || P2,
                I,
                union(getOutputs(P2), getOutputs(Q)),
                A)
        if
        pConfig(Sigma, Delta1, P1,
                union(I, getOutputs(Q)),
                getOutputs(P1), A)
        =>
        pConfig(Sigma, Delta2, P2, I1, O2, A)
        /\ typeOf(Sigma, Delta1,
                 union(I, getOutputs(P1)), A, Q)
        /\ I1 == union(I, getOutputs(Q))
        /\ O2 == getOutputs(P1)
        /\ O == union(getOutputs(P1), getOutputs(Q)).
```

CONG-NEW :

```
    crl [CONG-NEW] :
        pConfig(Sigma, Delta1,
                new cn : T in P1, I, O1, A)
        =>
        pConfig(Sigma,
                removeEntry ((chn cn) :: T) Delta2,
                new cn : T in P2, I, getOutputs(P2), A)
        if
        pConfig(Sigma, ((chn cn) :: T) Delta1,
                P1, I, insert(chn cn, O1), A)
        =>
        pConfig(Sigma, Delta2, P2, I, O2, A)
        /\ O2 == insert(chn cn, getOutputs(P2))
        /\ Delta2 equiv (((chn cn) :: T) Delta1)
        /\ insert(chn cn, O1) equiv O2
        .
```

where `removeEntry` deletes a channel from a channel context.

NEW-EXCH :

```
    crl [NEW-EXCH] :
        pConfig(Sigma, Delta,
                 new cn1 : T1 in
                    new cn2 : T2 in P, I, O, A)
        =>
```

```
    pConfig(Sigma, Delta,
              new cn2 : T2 in
               new cn1 : T1 in P, I, O, A)
    if
    typeOf(Sigma, Delta (chn cn1 :: T1)
                        (chn cn2 :: T2),
           I, A, P) /\
    getOutputs(P) == insert(chn cn1,
                            insert(chn cn2, O)) .
```

COMP-NEW :

```
crl [COMP-NEW] :
    pConfig(Sigma, Delta,
           P || (new cn : T in Q), I, O, A)
    =>
    pConfig(Sigma, Delta,
           new cn : T in (P || Q), I, O, A)
    if
    typeOf(Sigma, Delta (chn cn :: T),
           union(I, getOutputs(P)), A, Q)
    /\
    typeOf(Sigma, Delta,
           union(I ,(getOutputs(Q) \ (chn cn))),
           A, P)
.
```

ABSORB-LEFT :

```
 crl [ABSORB-LEFT] :
    pConfig(Sigma, Delta, P1 || P2, I, O, A) =>
    pConfig(Sigma, Delta, P1, I, O, A)
    if
    typeOf(Sigma, Delta, I, A, P1)
    /\
    typeOf(Sigma, Delta, union(I, O), A, P2)
    /\
    getOutputs(P1) == O
    /\
    getOutputs(P2) == empty
    .
```

ABSORB-RIGHT :

```
crl [ABSORB-RIGHT] :
    pConfig(Sigma, Delta, P1 || P2, I, O, A) =>
    pConfig(Sigma, Delta, P2, I, O, A)
    if
    typeOf(Sigma, Delta, I, A, P2)
    /\
    typeOf(Sigma, Delta, union(I, O), A, P1)
    /\
    getOutputs(P2) == O
    /\
    getOutputs(P1) == empty
.
```

DIVERGE :

```
crl [DIVERGE] :
    pConfig(Sigma, Delta,
            cn ::= x : T <- read cn ; R,
            I, chn cn, A)
    =>
    pConfig(Sigma, Delta,
            cn ::= read cn, I, chn cn, A)
    if
    typeOf(Sigma, Delta, emptyTypeContext,
           insert(chn cn, I), A, R)
    ==
    typeInCtx(chn cn, A, Delta)
    /\ occurs (chn cn) Delta A
     .
```

FOLD-IF-RIGHT :

```
crl [FOLD-IF-RIGHT] :
    pConfig(Sigma, Delta,
            new cn1 : T in
             ((cn2 ::= b : bool <- R ;
                        if b then S1
                             else read cn1)
               ||
               (cn1 ::= S2)
              )
           ,I, O, A)
    =>
    pConfig(Sigma, Delta,
            cn2 ::= b : bool <- R ;
```

```
                    if b then S1 else S2
            , I, O, A)
    if
    typeOf(Sigma, Delta, emptyTypeContext,
           I, A, R)
    ==
    bool
    /\
    typeOf(Sigma, Delta, emptyTypeContext,
           insert(chn cn2, I), A, S1)
    ==
    T
    /\
    typeOf(Sigma, Delta, emptyTypeContext,
           insert(chn cn2, I), A, S2) == T
    /\
    O == chn cn2
    /\
    elem (chn cn2) T Delta A .
```

FOLD-IF-LEFT :

```
crl [FOLD-IF-LEFT] :
    pConfig(Sigma, Delta,
            new cn2 : T in
               ((cn1 ::= b : bool <- R ;
                          if b then read cn2
                               else S2)
                 ||
                 (cn2 ::= S1))
           ,I, O, A)
    =>
    pConfig(Sigma, Delta,
            cn1 ::= b : bool <- R ;
                    if b then S1 else S2
            , I, O, A)
    if
    typeOf(Sigma, Delta, emptyTypeContext,
           I, A, R)
    ==
    bool
    /\
    typeOf(Sigma, Delta, emptyTypeContext,
           insert(chn cn1, I), A, S1) == T
    /\
```

```
        typeOf(Sigma, Delta, emptyTypeContext,
               insert(chn cn1, I), A, S2) == T
        /\
        O == chn cn1
        /\
        elem (chn cn1) T Delta A .
```

FOLD-BIND :

```
crl [FOLD-BIND] :
    pConfig(Sigma, Delta,
            new c : T in
               ((o ::= x : T <- read c ; S)
                 ||
                (c ::= R)),
            I, O, A)
  =>
    pConfig(Sigma, Delta,
            o ::= x : T <- R ; S,
            I, O, A)
  if
     typeOf(Sigma, Delta, x : T,
            (I, chn c),
            A, S)
     ==
     typeInCtx(chn o, A, Delta)
  /\ typeOf(Sigma, Delta, emptyTypeContext,
            I, A, R)
     ==
     T
  /\ O == chn o .
```

SUBSUME :

```
crl [SUBSUME] :
    pConfig(Sigma, Delta,
            (cn1 ::= x0 : T0 <- read i ; R1) ||
            (cn2 ::= x0 : T0 <- read i ;
                     x1 : T1 <- read cn1 ;
                     R2)
          , I, O, A)
  =>
    pConfig(Sigma, Delta,
            (cn1 ::= x0 : T0 <- read i ; R1) ||
            (cn2 ::= x1 : T1 <- read cn1 ; R2)
```

```
        , I, O, A)
  if typeOf(Sigma, Delta, x1 : T1,
            insert(chn cn1, insert(chn cn2, I)),
            A, R2) ==
     typeInCtx(chn cn2, A, Delta)
  /\ O == insert(chn cn1, insert(chn cn2, empty))
  /\ elem (chn cn1) T1 Delta A .
```

This rule is actually derivable.

DROP :

```
crl [DROP] :
  pConfig(Sigma, Delta,
          (cn1 ::= R1) ||
          (cn2 ::= x1 : T1 <- read cn1 ; R2)
         ,I, O, A)
  =>
  pConfig(Sigma, Delta,
          (cn1 ::= R1) || (cn2 ::= R2)
         ,I, O, A)
  if rConfig(Sigma, Delta, emptyTypeContext,
             x1 : T1 <- R1 ; R2
            , insert(chn cn1, insert(chn cn2, I)),
            A, typeInCtx(chn cn2, A, Delta))
     =>
     rConfig(Sigma, Delta, emptyTypeContext,
             R2
            ,I', A, T2) /\
     T2 == typeInCtx(chn cn2, A, Delta) /\
     I' == insert(chn cn1, insert(chn cn2, I)) /\
     O == insert(chn cn1, insert(chn cn2, empty)) /\
     typeOf(Sigma, Delta, emptyTypeContext,
            insert(chn cn1, insert(chn cn2, I)),
            A, R2)
     ==
     typeInCtx(chn cn2, A, Delta) /\
     elem (chn cn1) T1 Delta A
     [nonexec] .
```

SUBST :

```
crl [SUBST] :
  pConfig(Sigma, Delta,
          (cn1 ::= R1)
```

```
          ||
          (cn2 ::= x1 : T1 <- read cn1 ;
                   R2),
          I, O, A)
  =>
  pConfig(Sigma, Delta,
          (cn1 ::= R1)
          ||
          (cn2 ::= x1 : T1 <- R1 ; R2),
          I, O, A)
  if
  rConfig(Sigma, Delta, emptyTypeContext,
          x1 : T1 <- R1 ;
          x2 : T1 <- R1 ;
          return pair(x1, x2),
          insert(chn cn1, insert(chn cn2, I)), A,
          T1 * T1 )
  =>
  rConfig(Sigma, Delta, emptyTypeContext,
          x1 : T1 <- R1 ;  return pair(x1, x1),
          I', A, T1 * T1 ) /\
  O == insert(chn cn1, chn cn2) /\
  I' == insert(chn cn1, insert(chn cn2, I)) /\
  elem (chn cn1) T1 Delta A
  [nonexec] .
```

2 Derived Rules

2.1 Plain Protocols

Here we only have derived rules at the reaction level. The rule names should be changed.

SAME-REACTION-IF

```
 crl [same-reaction-if] :
  rConfig(Sigma, Delta, Gamma,
          if M then R else R,
          I, A, T)
  =>
  rConfig(Sigma, Delta, Gamma,
          R, I, A, T)
if typeOf(Sigma, Delta, Gamma,
          I, A, R) == T
/\ typeOf(Sigma, Gamma, M) == bool
.
```

Proof: Assume x : bool is a variable that doesn't occur in R. Then

```
R
= (by def. of _[_/_])
R [x / M]
=> (by if-ext)
 if M then R[x/True] else R[x/False]
= (by def. of _[_/_] )
 if M then R else R
```

CONG-BRANCH-REFL :

```
  crl [cong-branch-refl] :
     rConfig(Sigma, Delta, Gamma,
             if M then R1 else R2, I, A, T)
    =>
     rConfig(Sigma, Delta, Gamma,
             if M then R3 else R4, I, A, T)
     if
     typeOf(Sigma, Gamma, M) == bool
     /\
     rConfig(Sigma, Delta, Gamma, R1, I, A, T) =>
     rConfig(Sigma, Delta, Gamma, R3, I, A, T)
     /\
     rConfig(Sigma, Delta, Gamma, R2, I, A, T) =>
     rConfig(Sigma, Delta, Gamma, R4, I, A, T) .
```

This holds immediately by `CONG-IF` and taking the rewrite for `M` as the one that leaves it as it is. I added this rule when I did not have expression equality and I think we could still leave it for convenience.

IF-OVER-BIND :

```
 crl [if-over-bind] :
    rConfig(Sigma, Delta, Gamma,
             x : T1 <- if M then R1 else R2 ;
             R , I, A, T)
   =>
    rConfig(Sigma, Delta, Gamma,
             if M then (x : T1 <- R1 ; R)
                  else (x : T1 <- R2 ; R) ,
            I, A, T)
 if typeOf(Sigma, Delta, Gamma,
           I, A, R1) == T1  /\
    typeOf(Sigma, Delta, Gamma,
           I, A, R2) == T1  /\
    typeOf(Sigma, Delta, Gamma (x : T1),
```

```
            I, A, R) == T /\
      typeOf(Sigma, Gamma, M) == bool
    .
```

Proof:

```
x : T1 <- if M then R1 else R2 ;
R
=> (by if-ext)
if M
 then x : T1 <- if True then R1
                        else R2 ;
      R
 else x : T1 <- if False then R1
                        else R2 ;
      R
=> (by cong-branch-refl{
        cong-bind{if-left, idle},
        cong-bind{if-right, idle}
       })
if M then x : T1 <- R1 ; R
     else x : T1 <- R2 ; R
```

BIND-OVER-IF :

```
crl [bind-over-if] :
    rConfig(Sigma, Delta, Gamma,
      if M then (x : T1 <- R1 ; R)
           else (x : T1 <- R1 ; S),
      I, A, T)
   =>
    rConfig(Sigma, Delta, Gamma,
       x : T1 <- R1 ;
       if M then R else S, I, A, T)
if
    typeOf(Sigma, Delta, Gamma,
          I, A, R1) == T1 /\
    typeOf(Sigma, Delta, Gamma (x : T1),
           I, A, R) == T /\
    typeOf(Sigma, Delta, Gamma (x : T1),
           I, A, S) == T /\
    typeOf(Sigma, Gamma, M) == bool
 .
```

Proof:

```
x : T1 <- R1 ;
if M then R else S
=> (by if-ext)
if M
 then x : T1 <- R1 ;
      if True then R else S
 else x : T1 <- R1 ;
      if False then R else S
=> (by cong-branch-refl{
         cong-bind{idle , if-left},
         cong-bind{idle , if-right}
       })
if M then x : T1 <- R1 ; R
     else x : T1 <- R1 ; S
```

IF-OVER-BIND-SAME :

```
crl [if-over-bind-same] :
   rConfig(Sigma, Delta, Gamma,
    x : T1 <- if M then R1 else R2 ;
    if M then R3 else R4,
    I, A, T)
   =>
   rConfig(Sigma, Delta, Gamma,
   if M then (x : T1 <- R1 ; R3)
        else (x : T1 <- R2 ; R4) ,
   I, A, T)
if typeOf(Sigma, Delta, Gamma,
          I, A, R1)
   == T1  /\
   typeOf(Sigma, Delta, Gamma,
          I, A, R2)
   == T1  /\
   typeOf(Sigma, Delta, Gamma (x : T1),
          I, A, R3)
   == T /\
   typeOf(Sigma, Delta, Gamma (x : T1),
          I, A, R4)
   == T /\
   typeOf(Sigma, Gamma, M) == bool
.
```

Proof:

```
x : T1 <- if M then R1 else R2 ;
if M then R3 else R4
```

```
 => (by if-ext)
 if M then
   x : T1 <- if True then R1 else R2 ;
   if True then R3 else R4
 else
   x : T1 <- if False then R1 else R2 ;
     if False then R3 else R4
=> (by cong-branch-refl{
         cong-bind{if-left , if-left},
         cong-bind{if-right , if-right}
       })
 if M then
     x : T1 <- R1 ; R3
        else
     x : T1 <- R2 ; R4
```

IF-OVER-BIND-SAME-2 :

```
crl [if-over-bind-same-2] :
     rConfig(Sigma, Delta, Gamma,
      x : T1 <-
        if M1
         then if M2 then R1 else R2
         else if M2 then R3 else R4 ;
      if M1
          then if M2 then S1 else S2
          else if M2 then S3 else S4,
      I, A, T)
   =>
    rConfig(Sigma, Delta, Gamma,
      if M1
         then if M2 then (x : T1 <- R1 ; S1)
                     else (x : T1 <- R2 ; S2)
         else if M2 then (x : T1 <- R3 ; S3)
                     else (x : T1 <- R4 ; S4),
      I, A, T)
  if typeOf(Sigma, Gamma, M1) == bool
  /\ typeOf(Sigma, Gamma, M2) == bool
  /\ typeOf(Sigma, Delta, Gamma,
             I, A, R1) == T1
  /\ typeOf(Sigma, Delta, Gamma,
             I, A, R2) == T1
  /\ typeOf(Sigma, Delta, Gamma,
             I, A, R3) == T1
  /\ typeOf(Sigma, Delta, Gamma,
             I, A, R4) == T1
```

```
  /\ typeOf(Sigma, Delta, Gamma (x : T1),
              I, A, S1) == T
  /\ typeOf(Sigma, Delta, Gamma (x : T1),
              I, A, S2) == T
  /\ typeOf(Sigma, Delta, Gamma (x : T1),
              I, A, S3) == T
  /\ typeOf(Sigma, Delta, Gamma (x : T1),
              I, A, S4) == T
 .
```

Proof:

```
x : T1 <-
  if M1
   then if M2 then R1 else R2
   else if M2 then R3 else R4 ;
if M1
   then if M2 then S1 else S2
   else if M2 then S3 else S4
=> (by if-ext for M2)
if M2
then
 x : T1 <-
  if M1
   then if True then R1 else R2
   else if True then R3 else R4 ;
if M1
   then if True then S1 else S2
   else if True then S3 else S4
else
x : T1 <-
  if M1
   then if False then R1 else R2
   else if False then R3 else R4 ;
if M1
   then if False then S1 else S2
   else if False then S3 else S4
=> (by if-left ,
        if-right
    under the right congruence rules)
if M2 then
 x : T1 <- if M1 then R1 else R3 ;
 if M1 then S1 else S3
else
 x : T1 <- if M1 then R2 else R4 ;
 if M1 then S2 else S4
=> (by if-ext for M1)
```

```
if M1
then
if M2 then
 x : T1 <- if True then R1 else R3 ;
 if True then S1 else S3
else
 x : T1 <- if True then R2 else R4 ;
 if True then S2 else S4
else
if M2 then
 x : T1 <- if False then R1 else R3 ;
 if False then S1 else S3
else
 x : T1 <- if False then R2 else R4 ;
 if False then S2 else S4
=> (by if-left ,
        if-right
    under the right congruence rules)
if M1
        then if M2 then (x : T1 <- R1 ; S1)
                    else (x : T1 <- R2 ; S2)
        else if M2 then (x : T1 <- R3 ; S3)
                    else (x : T1 <- R4 ; S4)
```

ALPHA :

```
var vx vy : Qid .

crl [alpha] :
    rConfig(Sigma, Delta, Gamma,
            vx : T1 <- R1 ; R2 ,
            I, A, T2 )
    =>
    rConfig(Sigma, Delta, Gamma,
             vy : T1 <- R1 ;
             (R2 [vx / vy]),
             I, A, T2)
if typeOf(Sigma, Delta, Gamma,
          I, A, R1) == T1 /\
   typeOf(Sigma, Delta, Gamma (vx : T1),
          I, A, R2) == T2 [nonexec] .
```

Follows by `sym{bind-ret} then bind-bind then ret-bind`:

```
load ../src/strategies
mod ALPHA-SOUND is
```

```
  including APPROX-EQUALITY .

  *** constants without definitions
  *** will be interpreted as any value of that type

  op Sigma : -> Signature .
  op Delta : -> ChannelContext .
  op Gamma : -> TypeContext .
  ops vx vy : -> Qid .
  ops R1 R2 : -> Reaction .
  op I : -> Set{CNameBound} .
  op A : -> Set{BoolTerm} .
  ops T1 T2 : -> Type .

  ***  I need this because I want
  ***  R2 to typecheck if the type context
  ***  has more than Gamma and (vx : T1)
  var Gamma' : TypeContext .

  *** assumptions
  eq typeOf(Sigma, Delta, Gamma, I, A, R1) = T1 .
  eq typeOf(Sigma, Delta,
            Gamma (vx : T1) Gamma',
            I, A, R2) = T2 .

endm

srew [1]
  rConfig(Sigma, Delta, Gamma,
          vx : T1 <- R1 ; R2,
          I, A, T2)
  using sym[R1:Reaction <-
             vx : T1 <- (vy : T1 <- R1 ;
                         return vy) ;
                         R2
           ]
       {cong-bind{bind-ret, idle}}
     ; bind-bind
     ; cong-bind{idle, ret-bind}
  .

  *** we get
  *** result ReactionConfig:
  *** rConfig(Sigma, Delta, Gamma,
  ***          vy : T1 <- R1 ; (R2[vx / vy]),
  ***          I, A, T2)
```

SAMP-OVER-IF :

```
crl [samp-over-if] :
    rConfig(Sigma, Delta, Gamma,
              x : T1 <- samp D ;
              if M then R1 else R2,
              I, A, T)
   =>
    rConfig(Sigma, Delta, Gamma,
              if M then x : T1 <- samp D ;
                        R1
                   else x : T1 <- samp D ;
                        R2,
              I, A, T)
if typeOf(Sigma, Delta, Gamma (x : T1),
            I, A, R1) == T
/\ typeOf(Sigma, Delta, Gamma (x : T1),
            I, A, R2) == T
/\ typeOf(Sigma, Gamma, D) == T1
    .
```

Proof:

```
x : T1 <- samp D ;
if M then R1 else R2
=> (by if-ext)
if M then
  x : T1 <- samp D ;
  if True then R1 else R2
else
 x : T1 <- samp D ;
 if False then R1 else R2
=> (by cong-branch-refl{
          cong-bind{idle, if-left},
          cong-bind{idle, if-right}
         })
if M then x : T1 <- samp D ;
                    R1
      else x : T1 <- samp D ;
                    R2
```

3 Normal Forms

3.1 Reactions

We introduce normal forms of reactions to avoid the use of the rule EXCH and CONG-BIND. The main idea is that instead of writing

```
x1 : T1 <- read C1 ;
...
xN : TN <- read CN ;
R
```

we turn the binds into a commutative list

```
nf(
(x1 : T1 <- read C1)
...
(xN : TN <- read CN) ,
R
)
```

and thus we can select any of them to apply reaction equality rules. The reaction `R` is bind free. We can relax this restriction and also the one that all binds read from channels, and then we obtain a pre-normal form instead.

The normal form of a reaction can be computed with a function `computeNF`, and we can also assume a selection among the reactions that are equivalent modulo their normal form that allows us to pick a certain order for the list of binds. This amounts to using a rule

```
crl[select-plain]:
rConfig(Sigma, Delta, Gamma,
        nf(BRL, R), I, A, T)
=>
rConfig(Sigma, Delta, Gamma,
        R', I, A, T)
if computeNF(R') == nf(BRL, R)
[nonexec]
```

in one direction and

```
rl[compute-nf]:
rConfig(Sigma, Delta, Gamma,
        R, I, A, T)
=>
rConfig(Sigma, Delta, Gamma,
        computeNF(R), I, A, T)
```

in the other. These rules are sound by definition.

alpha-nf :

```
crl [alpha-nf] :
   rConfig(Sigma, Delta, Gamma,
           nf((vx : T1 <- R1) BRL,
              R2
             ),
           I, A, T2
          )
    =>
    rConfig(Sigma, Delta, Gamma,
           nf((vy : T1 <- R1) BRL,
              R2 [vx / vy]
             ),
           I, A, T2
          )
if typeOf(Sigma, Delta, Gamma,
          I, A, R1) == T1 /\
   typeOf(Sigma, Delta,
          addDeclarations BRL (Gamma (vx : T1)),
          I, A, R2) == T2
     [nonexec]
.
```

We start with `nf((vx : T1 <- R1) BRL, R2)`. we can turn this into a plain reaction `R'` by selecting the order of binds where `vx` comes last. We can then define a Maude strategy

```
strat S @ ReactionConfig .
sd S :=
 alpha[vy:Qid <- vy]
 or-else
 cong-bind{idle, S}
.
```

By applying it recursively, we leave all binds in `BRL` unchanged. When we reach `vx : T1 <- R1 ; R2` we notice that the conditions of the ALPHA rule hold and we can do the renaming `vy : T1 <- R1 ; R2[vx / vy]`. The result of applying `S` to `R'` is then a reaction `R''` that starts with the binds in `BRL` and ends with `vy : T1 <- R1 ; R2[vx / vy]`. The normal form of `R''` is precisely `nf((vy : T1 <- R1) BRL, R2[vx / vy])`.

cong-nf :

```
crl [cong-nf] :
    rConfig(Sigma, Delta, Gamma,
            nf(BRL , R1), I, A, T)
    =>
    rConfig(Sigma, Delta, Gamma,
```

```
            nf(BRL , R2), I, A, T)
    if
    rConfig(Sigma, Delta,
            addDeclarations BRL Gamma,
            R1, I, A, T)
    =>
    rConfig(Sigma, Delta, Gamma',
             R2, I, A, T)
    /\  Gamma' == addDeclarations BRL Gamma .
```

We start with `nf(BRL, R1)` and by assumption we know that

```
rConfig(Sigma, Delta,
            addDeclarations BRL Gamma,
            R1, I, A, T)
    =>
    rConfig(Sigma, Delta, Gamma',
             R2, I, A, T)
```

by a rewrite that we call `rew`. We can turn `nf(BRL, R1)` into a plain reaction `R'` by selecting any order of binds. We then define a Maude strategy

```
strat S @ ReactionConfig .
sd S :=
 cong-bind{idle, rew}
 or-else
 cong-bind{idle, S}
.
```

By applying it recursively, we leave all binds in `BRL` unchanged and when we reach `R1` we can rewrite it to `R2` using `rew`, as `cong-bind` adds all declarations in `BRL` to `Gamma` by repeated application. The result of applying `S` to `R'` is a reaction `R''` that starts with the binds in `BRL` and ends with `R2`. The normal form of `R''` is precisely `nf(BRL, R2)`.

read-det-pre :

```
  crl [read-det-pre] :
    rConfig(Sigma, Delta, Gamma,
              preNF( (x : T1 <- read i)
                     (y : T1 <- read i) BL ,
                     R ),
           I, A, T2)
    =>
    rConfig(Sigma, Delta, Gamma,
              preNF( (x : T1 <- read i) BL ,
```

```
                        R [y / x] ),
              I, A, T2)
  if isElemB(i, I, A)  /\
     elem (toBound i) T1 Delta A /\
     typeOf(Sigma, Delta,
            addDeclarations BL
              (Gamma (x : T1) (y : T1)),
            I, A, R)
     == T2 .
```

We start with the reaction

```
 preNF( (x : T1 <- read i)(y : T1 <- read i) BL , R )
```

and select its plain reaction equivalent `R'` that starts with the binds in BL and ends with

```
   x : T1 <- read i ;
   y : T1 <- read i ;
   R
```

We then define a Maude strategy

```
strat S @ ReactionConfig .
sd S :=
 cong-bind{idle, read-det-pre}
 or-else
 cong-bind{idle, S}
.
```

By applying it recursively, we leave all binds in `BL` unchanged until we reach the left-hand side of the rule `read-det`. This rule has the same conditions as `read-det-pre`, and we know these hold by assumption, since the declarations in `BL` are added to `Gamma` by repeatedly applying `cong-bind`. We can apply `read-det` to get

```
   x : T1 <- read i ;
   (R[y / x])
```

The result of applying the strategy to `R'` is then a reaction `R''` that starts with the binds in BL and ends with

```
   x : T1 <- read i ;
   (R[y / x])
```

Its pre-normal form is precisely

```
 preNF( (x : T1 <- read i) BL ,
                        R [y / x] )
```

and we obtain it by calling `computeNF(R'')` and applying `nf2Pre` to the result if needed.

read-det-nf :

```
crl [read-det-nf] :
    rConfig(Sigma, Delta, Gamma,
               nf( (x : T1 <- read i)
                     (y : T1 <- read i) BRL ,
                    R),
            I, A, T2)
    =>
    rConfig(Sigma, Delta, Gamma,
               nf( (x : T1 <- read i) BRL ,
                     R [y / x]),
            I, A, T2)
  if isElemB(i, I, A)   /\
     elem (toBound i) T1 Delta A /\
     typeOf(Sigma, Delta,
            addDeclarations BRL
             (Gamma (x : T1) (y : T1)),
            I, A, R)
     == T2 .
```

Same proof as above, only use `read-det` in the strategy and turn the final plain reaction to a normal form instead of a pre-normal form.

bind-ret-pre :

```
crl [bind-ret-pre] :
    rConfig(Sigma, Delta, Gamma,
               preNF( (x : T1 <~ R1) BL ,
                       return x ),
            I, A, T1)
    =>
    rConfig(Sigma, Delta, Gamma,
               preNF( BL , R1 ),
            I, A, T1)
  if typeOf(Sigma, Delta, Gamma,
            I, A, R1)
     == T1 .
```

We start with

```
preNF( (x : T1 <~ R1) BL ,
        return x )
```

We can turn it into a plain reaction `R'` by selecting the order of binds that starts with `BL` and ends with `x : T1 <- R1 .` We then define a Maude strategy

```
strat S @ ReactionConfig .
sd S :=
 cong-bind{idle , bind-ret}
 or-else
 cong-bind{idle , S}
.
```

By applying it recursively, we leave all binds in `BL` unchanged and when we reach `x : T <- R1 ; return x` we rewrite it to `R1` using `bind-ret`. The result of applying `S` to `R'` is a reaction `R''` that starts with the binds in `BL` and ends with `R1`. The normal form of `R''` is precisely `preNF(BL, R1)`.

read2Binds :

```
crl [read2Binds] :
    rConfig(Sigma, Delta, Gamma,
              preNF(BL (x : T1 <~ read i),
                    R ),
           I, A, T)
   =>
    rConfig(Sigma, Delta, Gamma,
              preNF(BL (x : T1 <- read i),
                    R ),
            I, A, T)
    if isElemB(i, I, A) and
       elem (chn i) T1 Delta A .
```

Both reactions have the same plain forms.

pre2Nf :

```
crl [pre2Nf] : preNF(BRL, R ) => nf(BRL, R)
if R : BindFreeReaction .
```

Both reactions have the same plain forms. The condition that `R` should be bind free and the requirement that `BRL` is a list of read binds ensures that the normal form is well-formed.

nf2Pre :

```
rl [nf2Pre] : nf(BRL, R) => preNF(BRL, R) .
```

Both reactions have the same plain forms.

merge-pre :

```
crl [merge-pre] :
    rConfig(Sigma, Delta, Gamma,
            preNF(BL (x : T1 <~ R1) ,
                  R2 ),
            I, A, T2)
   =>
    rConfig(Sigma, Delta, Gamma,
             preNF(BL ,
                   x : T1 <- R1 ; R2 ),
            I, A, T2)
  if typeOf(Sigma, Delta, Gamma,
            I, A, R1)
     == T1
  /\ typeOf(Sigma, Delta,
            addDeclarations BL (Gamma (x : T1)),
            I, A, R2)
     == T2 .
```

Both reactions have the same plain forms.

bind2R-pre-reverse :

```
crl [bind2R-pre-reverse] :
    rConfig(Sigma, Delta, Gamma,
            preNF(BL ,
                  x : T1 <- read i ; R2 ),
            I, A, T2)
   =>
    rConfig(Sigma, Delta, Gamma,
            preNF(BL (x : T1 <- read i) ,
                  R2 ),
            I, A, T2)
if isElemB(i, I, A)
/\ elem (chn i) T1 Delta A
/\ typeOf(Sigma, Delta,
          addDeclarations BL (Gamma (x : T1)),
          I, A, R2)
   == T2 .
```

Follows by the previous rule and symmetry.

ret-bind-pre :

```
crl [ret-bind-pre] :
    rConfig(Sigma, Delta, Gamma,
             preNF((x : T1 <~ (return M)) BL,
                   R ),
            I, A, T2)
```

```
    =>
    rConfig(Sigma, Delta, Gamma,
            preNF(BL,
                  R [x / M] ),
            I, A, T2)
if typeOf(Sigma, Gamma, M) == T1
/\ typeOf(Sigma, Delta,
          addDeclarations BL (Gamma (x : T1)),
          I, A, R)
   == T2
```

Start with

```
preNF((x : T1 <~ (return M)) BL, R )
```

and select its plain form that starts with the binds in BL and ends with `x : T1 <- return M ; R`. We then define a Maude strategy

```
strat S @ ReactionConfig .
sd S :=
 cong-bind{idle, ret-bind}
 or-else
 cong-bind{idle, S}
.
```

By applying it recursively, we leave all binds in `BL` unchanged and when we reach `x : T <- return M ; R` we rewrite it to `R[x / M]` using `ret-bind`. The normal form of the resulting reaction is `preNF(BL, R [x / M] )`, possibly applying `nf2Pre` if BL has only read binds and `R [x / M]` is bind free.

bind-bind-pre :

```
crl [bind-bind-pre] :
    rConfig(Sigma, Delta, Gamma,
             preNF((x2 : T2 <~ nf(BRL, R2)) BL,
                   R1),
           I, A, T1)
    =>
    rConfig(Sigma, Delta, Gamma,
             preNF(BRL (x2 : T2 <~ R2) BL,
                  R1),
            I, A, T1)
if typeOf(Sigma, Delta,
           addDeclarations BRL Gamma,
          I, A, R2)
   == T2
/\ typeOf(Sigma, Delta,
```

```
            addDeclarations BL (Gamma (x2 : T2)),
            I, A, R1)
    == T1 .
```

We start with

```
preNF( (x2 : T2 <~ nf(BRL, R2)) BL, R1)
```

and select the plain representation `P` that starts with the binds in BL and ends with `x2 : T2 <- R' ; R1` where `R'` is any plain representation of `nf(BRL, R2)`. We define two Maude strategies. The first one will extract the binds in `BRL` from the inner reaction and lift them to the outer level:

```
start S1 @ ReactionConfig .
sd S1 :=
  bind-bind
  or-else
  cong-bind{idle , S1}
.
```

while the other will leave unchanged the outer binds:

```
strat S2 @ ReactionConfig .
sd S2 :=
 S1
 or-else
 cong-bind{idle , S2}
.
```

When applying `S2` to `P` we obtain a reaction `P'` that has first the binds in `BL`, then the ones in `BRL`, and finally `x2 : T2 <- R2 ; R1`. The pre-normal form of `P'` is

```
preNF(BRL (x2 : T2 <~ R2) BL, R1)
```

bind-bind-pre-pre :

```
  crl [bind-bind-pre-pre] :
     rConfig(Sigma, Delta, Gamma,
               preNF((x2 : T2 <~ preNF(BRL, R2)) BL,
                      R1),
             I, A, T1)
     =>
     rConfig(Sigma, Delta, Gamma,
               preNF(BRL (x2 : T2 <~ R2) BL,
                      R1),
             I, A, T1)
 if typeOf(Sigma, Delta,
           addDeclarations BRL Gamma,
```

```
              I, A, R2)
      == T2
   /\ typeOf(Sigma, Delta,
              addDeclarations BL (Gamma (x2 : T2)),
              I, A, R1)
      == T1 .
```

The proof is similar to the one above, namely the same strategies are used, and the only thing that changes is that we start with an inner pre-normal form.

3.2 Protocols

We introduce normal forms of protocols to avoid the use of the rule NEW-EXCH. The main idea is that instead of writing

```
new cn1 : T1 in
new cn2 : T2 in
...
new cnN : TN in
P
```

we turn the hidden channels into a commutative list

```
newNF(
< C1 : T1 >
...
< CN : TN > ,
P
)
```

and thus we can select any of them to apply protocol equality rules.

The normal form of a protocol can be computed with a function `new2NF`, and we can also assume a selection among the protocols that are equivalent modulo their normal form that allows us to pick a certain order for the list of hidden channels. This amounts to using a rule

```
 crl [select-plain] :
     pConfig(Sigma, Delta,
              newNF(ltq, P1),
              I, O, A)
     =>
     pConfig(Sigma, Delta,
               P,
              I, O, A)
 if new2NF(P) == newNF(ltq, P1)
 .
```

in one direction, that chooses the plain form of a protocol in normal form given by the alphabetical order of names of hidden channels, and

```
rl [sugar-newNF] :
    pConfig(Sigma, Delta,
            P,
            I, O, A)
   =>
    pConfig(Sigma, Delta,
            new2NF(P),
            I, O, A)
.
```

in the other. These rules are sound by definition.

delete-empty-newNF

```
rl [delete-empty-newNF] :
    pConfig(Sigma, Delta,
            newNF(emptyTypedCNameList, P),
            I, O, A)
   =>
    pConfig(Sigma, Delta,
             P,
            I, O, A)
.
```

The empty list of hidden channels doesn't add anything to the normal form of `P`, so both `P` and `newNF(emptyTypedCNameList, P)` have the same plain representations.

CONG-NEW-NF

```
crl [CONG-NEW-NF] :
    pConfig(Sigma, Delta1,
              newNF(ltq, P1),
            I, O1, A)
   =>
    pConfig(Sigma,
              diff
               Delta2
               (addChannels ltq emptyChannelCtx),
            newNF(ltq, P2),
            I, O2 \ (chansInList ltq), A)
    if
    pConfig(Sigma, addChannels ltq Delta1,
              P1,
              I, union(chansInList ltq, O1), A)
   =>
    pConfig(Sigma, Delta2,
```

```
              P2,
             I, O2, A)
    /\ *** the channels in ltq have not changed
    diff
     (addChannels ltq emptyChannelCtx)
     Delta2
    == emptyChannelCtx
    /\
    O2 == getOutputs(P2)
    /\
    (addChannels ltq Delta1) equiv Delta2
    /\
    O2 equiv (chansInList ltq, O1)
  .
```

Let `rew` be the rewrite in the condition of the rule. We start with the protocol `newNF(ltq, P1)` and select any plain representation of it `Q1`. We define the following Maude strategy:

```
strat S @ ProtocolConfig .
sd S :=
    rew
    or-else CONG-NEW{S}
  .
```

The strategy adds arbitrarily many hidden channels to the current context and then applies `rew`. The result of applying `S` to `Q1` is a protocol `Q2` that has the same hidden channels as `Q1` followed by `P2`. By taking its normal form we obtain precisely `newNF(ltq, P2)`.

absorb-new-nf

```
crl [absorb-new-nf] :
     pConfig(Sigma, Delta,
               newNF(< c : T > ltq,
                      P || (c ::= R)
               ),
              I, O, A)
    =>
     pConfig(Sigma, Delta,
               newNF(ltq, P),
              I, O, A)
 if typeOf(Sigma,
             addChannels ltq
               (Delta (chn c :: T)),
             emptyTypeContext,
             (chn c, (I, getOutputs(P))),
             A, R)
    == T
```

```
 /\  typeOf(Sigma,  addChannels ltq  Delta,
            I,  A,  P)
 /\  getOutputs(newNF2New(newNF(ltq,  P)))
     == O  .
```

We start with `newNF(< c : T > ltq, P || (c ::= R))` and let $Q1$ be its plain representation that starts with the hidden channels in `ltq` then with the hidden channel `c` and the protocol `P || (c ::= R)`. We define the following Maude strategy:

```
strat  S @ ProtocolConfig  .
sd  S  :=
 (COMP-NEW-2  ;  ABSORB-LEFT)
 or-else
 CONG-NEW{S}
.
```

The strategy adds arbitrarily many hidden channels to the current context and then applies `COMP-NEW-2` to turn `new c : T in (P || c ::= R)` into `P || (new c : T in c ::= R)`. The assumptions of `absorb-new-nf` ensure that`P` type checks in the absence of `c` from the channel context and that `new c : T in c ::= R` type checks with the outputs of `P` as inputs, so we can apply `ABSORB-LEFT` to eliminate `new c : T in c ::= R`. The result of applying `S` to `Q1` is a protocol `Q2` that starts with the hidden channels in `ltq` and ends with `P`. The normal form of `Q1` is precisely `newNF(ltq, P)`.

fold-bind-new-nf

```
crl  [fold-bind-new-nf]  :
  pConfig(Sigma,  Delta,
          newNF(<  c  :  T  >  ltq,
                P  ||
                (  c  ::=  R  )  ||
                (  o  ::=  nf((x  :  T <- read  c)  BRL,
                              S)
                )
               ),
          I,  O,  A)
 =>
  pConfig(Sigma,  Delta,
          newNF(ltq,
                P  ||
                (o  ::=  preNF((x  :  T <~ R)  BRL,
                               S))
                ),
          I,  O,  A)
  if  typeOf(Sigma,
```

```
                addChannels ltq
                   ( Delta ( (chn c):: T) ) ,
                  emptyTypeContext,
                (chn o,
                  (chn c, (I, getOutputs(P)) )
                ),
                A, R)
         == T
      /\ typeOf(Sigma, addChannels ltq Delta,
                addDeclarations
                  ((x : T <- read c) BRL)
                  emptyTypeContext,
                (chn o,
                  (I, getOutputs(P)) ), A, S
         )
         ==
         typeInCtx(chn o, A, addChannels ltq Delta)
      /\ typeOf(Sigma, addChannels ltq Delta,
            (I, chn o), A, P)
   .
```

We start with

```
newNF(< c : T > ltq,
       P || ( c ::= R ) ||
      ( o ::= nf((x : T <- read c) BRL, S) )
)
```

and we select the plain representation `Q1` that starts with the hidden channels in `ltq` and ends with

```
new c : T in
  P ||
  ( c ::= R ) ||
  ( o ::= nf((x : T <- read c) BRL, S) )
```

We define the following Maude strategy

```
strat S @ ProtocolConfig .
sd S :=
 COMP-NEW-2 ; CONG-COMP-RIGHT{FOLD-BIND}
 or-else
 CONG-NEW{S}
.
```

The strategy `S` leaves the hidden channels in `ltq` unchanged, then the rule `COMP-NEW-2` rewrites

```
new c : T in
    P ||
    ( c ::= R ) ||
    ( o ::= nf((x : T <- read c) BRL, S) )
```

to

```
  P || new c : T in
          ( c ::= R ) ||
          ( o ::= nf((x : T <- read c) BRL, S) )
```

Then `CONG-COMP-RIGHT{FOLD-BIND}` leaves P unchanged (by `CONG-COMP-RIGHT`) and rewrites

```
 new c : T in
   ( c ::= R ) ||
   ( o ::= nf((x : T <- read c) BRL, S)
```

to

```
 o ::= preNF((x : T <~ R) BRL, S)
```

(by `FOLD-BIND`). The result of applying `S` to `Q1` is then a protocol `Q2` that starts with the hidden channels in `ltq` and ends with

```
 P || (o ::= preNF((x : T <~ R) BRL, S))
```

. The normal form of `Q2` is the protocol in the right hand side of the rule `fold-bind-new-nf`.

fold-bind-new-nf-0

```
crl [fold-bind-new-nf-0] :
  pConfig(Sigma, Delta,
          newNF(< c : T > ltq,
                (c ::= R) ||
                (o ::= nf((x : T <- read c) BRL,
                          S))
               ),
          I, O, A)
  =>
  pConfig(Sigma, Delta,
          newNF(ltq,
                (o ::= preNF((x : T <~ R) BRL,
                             S))
               ),
          I, O, A)
  if typeOf(Sigma,
            addChannels ltq
              (Delta ((chn c):: T)),
```

```
                emptyTypeContext,
               (chn o, (chn c, I )),
               A, R)
          == T  /\ typeOf(Sigma, addChannels ltq Delta,
                addDeclarations
                   ((x : T <- read c) BRL)
                   emptyTypeContext,
                (chn o, I), A, S
        )
        ==
        typeInCtx(chn o, A, addChannels ltq Delta)
        .
```

The proof is similar to the one above, the only difference is that the strategy `S` doesn't apply `CONG-COMP-RIGHT` anymore:´

```
strat S @ ProtocolConfig .
sd S :=
 COMP-NEW-2 ; FOLD-BIND
 or-else
 CONG-NEW{S}
.
```

fold-bind-new-prenf

```
   crl [fold-bind-new-prenf] :
     pConfig(Sigma, Delta,
              newNF(< c : T > ltq,
                     P ||
                     (c ::= R) ||
                     (o ::= preNF((x : T <- read c)
                                  BRL,
                                   S))
                     ),
             I, O, A)
    =>
     pConfig(Sigma, Delta,
              newNF(ltq,
                     P ||
                     (o ::= preNF((x : T <~ R) BRL,
                                  S))
                    ),
             I, O, A)
  if typeOf(Sigma,
            addChannels ltq
              (Delta ((chn c) :: T)),
              emptyTypeContext,
```

```
                    (chn o, (chn c,
                      union(I, getOutputs(P)))),
                    A, R)
        == T .
```

The proof is the same as for `fold-bind-new-nf`.

COMP-NEW-newNF

```
crl [COMP-NEW-newNF] :
    pConfig(Sigma, Delta,
             P || newNF(ltq, Q),
           I, O, A)
   =>
    pConfig(Sigma, Delta,
             newNF(ltq, P || Q),
            I, O, A)
if typeOf(Sigma,
            addChannels ltq Delta,
            union(I, getOutputs(P)),
            A, Q)
/\ typeOf(Sigma, Delta,
          union(I, getOutputs(newNF(ltq, Q))),
           A, P) .
```

Start with `newNF(ltq, P || Q)` and consider the plain representation `Q1` that starts with any order of the hidden channels in `ltq` and ends with `P || Q`.

We define the following Maude strategies

```
strat S @ ProtocolConfig.
sd S :=
   COMP-NEW
   or-else
   CONG-NEW{S}
```

By applying `S!` to `Q1` (using the `!` strategy operator that applies a strategy as many times as possible), we obtain the protocol `P || Q2`, where `Q2` starts with the hidden channels in `ltq` and ends with `Q`. The normal form of `Q2` is `newNF(ltq, Q)`, so we can plug it next to `P` using `CONG-COMP-RIGHT` to obtain `P || newNF(ltq, Q)`. The proof is completed by applying the `SYM` rule to the proof above.

COMP-NEW-newNF-inside-new

```
crl [COMP-NEW-newNF-inside-new] :
    pConfig(Sigma, Delta,
             newNF(ltq1,
                   P || newNF(ltq, Q)),
```

```
             I, O, A)
       =>
       pConfig(Sigma, Delta,
                 newNF(ltq1 ltq,
                         P || Q),
              I, O, A)
  if typeOf(Sigma, addChannels ltq Delta,
             (I, getOutputs(P)),
             A, Q)
  /\ typeOf(Sigma, Delta,
             ( I, getOutputs(newNF(ltq, QL)) ),
            A, P) .
```

The proof is similar to the one above, but we restrict the number of applications of `S` to the length of `ltq`, then using `CONG-NEW` for the hidden channels in `ltq1`.

DROP-nf

```
crl [DROP-nf] :
     pConfig(Sigma, Delta,
             (cn1 ::= nf(emptyBRList, samp Dist)) ||
             (cn2 ::= nf( (x : T1 <- read cn1) BRL ,
                             R2) ),
             I, O, A)
     =>
      pConfig(Sigma, Delta,
             (cn1 ::= nf(emptyBRList, samp Dist)) ||
             (cn2 ::= nf( BRL , R2) ),
             I, O, A)
     if
      typeOf(Sigma, Delta,
              addDeclarations BRL (x : T1),
              (chn cn1, (chn cn2, I)),
              A, R2)
      ==
      typeInCtx(chn cn2, A, Delta)
      /\
      elem (chn cn1) T1 Delta A .
```

We start with

```
(cn1 ::= nf(emptyBRList, samp Dist)) ||
(cn2 ::= nf( (x : T1 <- read cn1) BRL , R2) )
```

and we replace the reactions assigned to the channels with `samp Dist` and the reaction that starts with the binds in `x : T1 <- read cn1` and ends with the binds in `BRL` followed by `R2` (we call this reaction `R'`), respectively. Working in reverse: by the rule `samp-pure`, we can rewrite the reaction `x : T1 <- samp Dist ; R'` to `R'`. Thus, by the rule `DROP`, we can rewrite the protocol

```
cn1 ::= samp Dist || cn2 ::= x : T1 <- read cn1 ; R'
```

to `cn1 ::= samp Dist || cn2 ::= R'`. The proof ends by replacing the reactions assigned to the channels `cn1` and `cn2` with their normal forms.

DROP-pre-nf

```
crl [DROP-pre-nf] :
  pConfig(Sigma, Delta,
           (cn1 ::= samp Dist) ||
           (cn2 ::= preNF( (x : T1 <- read cn1) BRL ,
                           R2) ),
          I, O, A)
  =>
 pConfig(Sigma, Delta,
          (cn1 ::= samp Dist) ||
          (cn2 ::= preNF( BRL , R2) ),
          I, O, A)
 if typeOf(Sigma, Delta,
            addDeclarations BRL (x : T1),
            (chn cn1, (chn cn2, I)), A, R2)
    ==
    typeInCtx(chn cn2, A, Delta)
 /\ elem (chn cn1) T1 Delta A .
```

The proof is identical to the one above, except we turn the reactions to their pre-normal form at the end.

DROP-SUBSUME-channels

```
crl [DROP-SUBSUME-channels] :
pConfig(Sigma, Delta,
         (cn1 ::= nf(BRL, samp Dist)) ||
         (cn2 ::= nf( (x : T1 <- read cn1) BRL' ,
                      R2) ),
         I, O, A)
=>
 pConfig(Sigma, Delta,
         (cn1 ::= nf(BRL, samp Dist)) ||
         (cn2 ::= nf(BRL BRL' , R2) ),
         I, O, A)
if
 typeOf(Sigma, Delta,
          addDeclarations BRL' (x : T1),
          (chn cn1, (chn cn2, I)),
          A, R2)
```

```
    ==
    typeInCtx(chn cn2, A, Delta)
    /\
    elem (chn cn1) T1 Delta A .
```

The proof is similar to the one of `DROP-nf` but before using the `DROP` rule we apply the reverse of the `SUBSUME` rule to duplicate the binds in `BRL` in the reaction assigned to the channel `cn2`. Each time a bind is added, we use the `EXCH` rule to move the read from `cn1` in front.

DROP-SUBSUME-channels-pre

```
crl [DROP-SUBSUME-channels-pre] :
pConfig(Sigma, Delta,
        (cn1 ::= nf(BRL, samp Dist)) ||
        (cn2 ::= preNF( (x : T1 <- read cn1) BRL' ,
                         R2) ),
        I, O, A)
=>
pConfig(Sigma, Delta,
        (cn1 ::= nf(BRL, samp Dist)) ||
        (cn2 ::= preNF(BRL BRL' , R2) ),
        I, O, A)
if typeOf(Sigma, Delta,
          addDeclarations BRL' (x : T1),
          (chn cn1, (chn cn2, I)), A, R2)
   ==
   typeInCtx(chn cn2, A, Delta)
/\ elem (chn cn1) T1 Delta A .
```

The proof is identical to the previous one, except at the end we compute the pre-normal form.

SUBST-nf

```
crl [SUBST-nf] :
    pConfig(Sigma, Delta,
            (cn1 ::= R1) ||
            (cn2 ::= nf( (x1 : T1 <- read cn1)
                         BRL ,
                         R2) ),
            I, O, A)
    =>
    pConfig(Sigma, Delta,
            (cn1 ::= R1) ||
            (cn2 ::= preNF( (x1 : T1 <~ R1)
                            BRL ,
                            R2)),
            I, O, A)
    if isSampFree(R1) /\
```

```
    O == insert(chn cn1, chn cn2) /\
  typeOf(Sigma, Delta, emptyTypeContext,
         (chn cn1,  (chn cn2, I)), A, R1)
  == T1 /\
  typeOf(Sigma, Delta,
        addDeclarations BRL (x1 : T1),
         (chn cn1,  (chn cn2, I)), A, R2)
  == typeInCtx(chn cn2, A, Delta)
  /\
  elem (chn cn1) T1 Delta A .
```

We start by showing that if a reaction is samp-free, the condition

```
rConfig(Sigma, Delta, emptyTypeContext,
          x1 : T1 <- R1 ;
          x2 : T1 <- R1 ;
          return pair(x1, x2),
          insert(chn cn1, insert(chn cn2, I)), A,
          T1 * T1 )
  =>
  rConfig(Sigma, Delta, emptyTypeContext,
          x1 : T1 <- R1 ;  return pair(x1, x1),
          I', A, T1 * T1 )
```

from the assumptions of the rule `SUBST` holds. To do that we will prove a stronger statement, namely that if a reaction is samp-free, then

```
rConfig(Sigma, Delta, emptyTypeContext,
          x1 : T1 <- R1 ;
          x2 : T1 <- R1 ;
          S(x1, x2),
          insert(chn cn1, insert(chn cn2, I)), A,
          T1 * T1 )
  =>
  rConfig(Sigma, Delta, emptyTypeContext,
          x1 : T1 <- R1 ; S(x, x),
          I', A, T1 * T1 )
```

holds for any reaction `S(x, y)`.

We proceed by structural induction.

Case `R1 = return M`: we start with

```
x1 : T1 <- return M ;
x2 : T1 <- return M ;
S(x1, x2)
```

By applying `ret-bind` two times, we get `S(M, M)`, and this is also what we get by applying `ret-bind` to `x1 : T1 <- return M ;  S(x1, x1)`.
Case `R = read c`: we start with

```
x1 : T1 <- read c ;
x2 : T1 <- read c ;
S(x1, x2)
```

and we can apply `read-det` to obtain

```
x1 : T1 <- read c ;
S(x1, x1)
```

Case `R = if M then R1 else R2`, with `R1, R2` samp-free: we start with

```
x1 : T1 <- if M then R1 else R2 ;
x2 : T1 <- if M then R1 else R2 ;
S(x1, x2)
```

and we notice that we can write it as

```
  if M then
       x1 : T1 <- R1 ;
       x2 : T1 <- R1 ;
       S(x1, x2)
         else
       x1 : T1 <- R2 ;
       x2 : T1 <- R2 ;
       S(x1, x2)
```

by applying to this latter reaction two times the derived rule `if-over-bind-same` followed by `same-reaction-if`. We can now use the inductive hypothesis for `R1` and `R2` to get

```
  if M then
       x1 : T1 <- R1 ;
       S(x1, x1)
         else
       x1 : T1 <- R2 ;
       S(x1, x1)
```

But this is also what we get if we apply `if-over-bind` to the right hand side reaction

```
  x1 : T1 <- if M then R1 else R2 ;
  S(x1, x1)
```

Case `R = (a : t <- R1) ; R2(a)`, with `R1, R2(a)` samp-free:

We start with

```
   x : T <-  a : t <- R1 ; R2(a) ;
   y : T <-  a : t <- R1 ; R2(a) ;
   S(x, y)
```

By bind-bind and exchange we get

```
a : t <- R1 ;
a : t <- R1 ;
x : T <-  R2(a) ;
y : T <-  R2(a) ;
S(x, y)
```

By the induction hypothesis for `R2(a)`

```
a : t <- R1 ;
a : t <- R1 ;
x : T <-  R2(a) ;
S(x, x)
```

By the induction hypothesis for `R1`

```
a : t <- R1 ;
x : T <-  R2(a) ;
S(x, x)
```

)

which is what we get from the right hand side as well by applying `bind-bind`.

We now proceed to showing that `SUBST-nf` is sound. We start with

```
(cn1 ::= R1) ||
          (cn2 ::= nf( (x1 : T1 <- read cn1)
                        BRL ,
                        R2) )
```

and we can choose the plain form of the reaction assigned to `cn2` that starts with `x1 : T1 <- read cn1` and ends with the binds in `BRL` followed by `R2`. Let `Q` denote this last fragment. Since `R1` is samp-free, we know that the assumptions of the `SUBST` rule hold, using the first statement we proved, and we get

```
(cn1 ::= R1) ||
          (cn2 ::= x1 : T1 <- R1 ; Q )
```

The normal form of `x1 : T1 <- R1 ; Q` is precisely

```
 preNF( (x1 : T1 <~ R1) BRL , R2)
```

.

SUBST-nf-read

```
crl [SUBST-nf-read] :
   pConfig(Sigma, Delta,
          (cn1 ::= nf((x2 : T1 <- read cn),
                      return x2)) ||
```

```
            (cn2 ::= nf( (x1 : T1 <- read cn1)
                          BRL ,
                          R2) ),
            I, O, A)
    =>
    pConfig(Sigma, Delta,
            (cn1 ::= nf((x2 : T1 <- read cn),
                        return x2)) ||
            (cn2 ::= nf((x2 : T1 <- read cn)
                        BRL ,
                        R2 [x1 / x2])),
            I, O, A)
    if
    isElemB(cn, I, A) /\
    O == (chn cn1, chn cn2) /\
    typeOf(Sigma, Delta,
           addDeclarations BRL (x1 : T1),
             (chn cn1,  (chn cn2, I)), A, R2)
    ==
    typeInCtx(chn cn2, A, Delta)
            /\
    elem (chn cn1) T1 Delta A
            /\
    elem (chn cn) T1 Delta A
    .
```

Follows immediately from soundness of `SUBST-nf`, for the particular case of `R1 = read cn` and applying the substitution strategy.

subst-diverge

```
crl [subst-diverge] :
  pConfig(Sigma, Delta,
            (cn1 ::= nf(x1 : T1 <- read cn1,
                        return x1))
            ||
            (cn2 ::= nf( (x2 : T1 <- read cn1)
                          BRL ,
                          R2)),
            I, O, A)
  =>
  pConfig(Sigma, Delta,
            (cn1 ::= nf(x1 : T1 <- read cn1,
                        return x1))
            ||
            (cn2 ::= nf(x3 : T2 <- read cn2,
                        return x3)),
            I, O, A)
```

```
if
 O == (chn cn1, chn cn2)
 /\
 elem (chn cn1) T1 Delta A
 /\
 elem (chn cn2) T2 Delta A
 /\
 typeOf(Sigma, Delta,
        addDeclarations BRL (x2 : T1),
        (chn cn1,  (chn cn2, I)), A, R2)
  ==
  typeInCtx(chn cn2, A, Delta)
        [nonexec]
.
```

Follows immediately from soundness of `SUBST-nf` for the particular case of `R1 = read cn1`, followed by application of the `DIVERGE` rule.

moveReadInnerNf

```
crl [moveReadInnerNf] :
  pConfig(Sigma, Delta,
          cn1 ::= nf((x : T <- read cn2)
                     BRL ,
                     R1) ,
          I, O, A)
   =>
    pConfig(Sigma, Delta,
          cn1 ::= preNF(BRL ,
                        x : T <- read cn2 ;
                        R1) ,
          I, O, A)
 if elem (chn cn2) T Delta A
 /\ typeOf(Sigma, Delta,
           addDeclarations BRL (x : T),
           (chn cn1, I), A, R1)
 == typeInCtx(chn cn1, A, Delta) .
```

The two protocols have the same plain forms.

moveReadInnerPreNf

```
crl [moveReadInnerPreNf] :
pConfig(Sigma, Delta,
        cn1 ::= preNF((x : T <- read cn2)
                      BRL ,
                      R1) ,
        I, O, A)
 =>
  pConfig(Sigma, Delta,
```

```
                 cn1 ::= preNF(BRL ,
                               x : T <- read cn2 ;
                               R1) ,
                 I, O, A)
    if elem (chn cn2) T Delta A
    /\ typeOf(Sigma, Delta,
                    addDeclarations BRL (x : T),
                 (chn cn1, I), A, R1)
       == typeInCtx(chn cn1, A, Delta) .
```

The two protocols have the same plain forms.