An Extensible Approach to Session Polymorphism
A brief introduction to the Coq formalization

Matthew Goto Radha Jagadeesan Alan Jeffrey
Corin Pitcher James Riely

2013-01-29

Contents

Chapter 1

Introduction

This is a very brief guide to the Coq formalization of the type system for extensible session polymorphism described in [GJJ$^+$]. The Coq scripts described here can be downloaded from `http://fpl.cs.depaul.edu/projects/xpol`.

This file is generated using `coq-tex`. The text after `Coq < Check lemma_name` is the statement of `lemma_name` and indicates that Coq has verified the proof of the statement.

The Coq scripts have been verified with the following version of Coq:

```
The Coq Proof Assistant, version 8.3 (December 2010)
compiled on Dec 03 2010 11:47:45 with OCaml 3.11.2
```

To verify the scripts, unzip the zip archive of scripts and execute `compile.sh`.

Chapter 2

Basic Definitions

Processes are defined in `Process.v`. A locally-nameless representation is used, and so the variables and names bound in `Input` and `New` processes are referenced using deBruijn indices.

```
Inductive proc : Set :=
| Input : value -> proc -> proc
| Output : value -> value -> proc -> proc
| IsEq : value -> value -> proc -> proc
| IsNotEq : value -> value -> proc -> proc
| New : proc -> proc
| Par : proc -> proc -> proc
| Sum : proc -> proc -> proc
| Rep : proc -> proc
| Zero : proc.
```

The following syntactic sugar is used. The unconventional notation for parallel composition `P ||| Q` and sum `+++` avoids syntactic conflicts with Coq and its libraries.

```
Notation "u ? ; P" := (Input u P) (...).
Notation "u ! v ; P" := (Output u v P) (...).
Notation "P ||| Q" := (Par P Q) (..., left associativity).
Notation "P +++ Q" := (Sum P Q) (..., left associativity).
Notation "! P" := (Rep P) (...).
```

The definition of structural equivalence is in `Process.v`. The `insert_proc` and `open_proc` functions used in rules for the `New` process manipulate deBruijn indices in connection with the locally-nameless representation.

```
Reserved Notation "P == Q" (...).

Inductive struct_equiv : proc -> proc -> Prop :=
| StrRefl : forall P, P == P
```

```
| StrSym : forall P Q, P == Q -> Q == P
| StrTrans : forall P Q R, P == Q -> Q == R -> P == R
| StrParZeroRight : forall P, P == (P ||| Zero)
| StrSumZeroRight : forall P, P == (P +++ Zero)
| StrParComm : forall P Q, (P ||| Q) == (Q ||| P)
| StrSumComm : forall P Q, (P +++ Q) == (Q +++ P)
| StrParAssoc : forall P Q R, ((P ||| Q) ||| R) == (P ||| (Q ||| R))
| StrSumAssoc : forall P Q R, ((P +++ Q) +++ R) == (P +++ (Q +++ R))
| StrRep : forall P, (! P) == (P ||| (! P))
| StrNewParRight : forall P Q, (P ||| (New Q)) == (New ((insert_proc 0 P) ||| Q))
| StrNewSumRight : forall P Q, (P +++ (New Q)) == (New ((insert_proc 0 P) +++ Q))
| StrParCongRight : forall P Q1 Q2, Q1 == Q2 -> (P ||| Q1) == (P ||| Q2)
| StrSumCongRight : forall P Q1 Q2, Q1 == Q2 -> (P +++ Q1) == (P +++ Q2)
| StrNewCong : forall P Q,
    (forall i,
      ~ (In i (free_ids_proc P))
      ->
      ~ (In i (free_ids_proc Q))
      ->
      open_proc i 0 P == open_proc i 0 Q)
    ->
    (New P) == (New Q)
where "P == Q" := (struct_equiv P Q).
```

The unannotated reduction relation is defined in `Process.v`. The function `subst_open_proc` in the interaction rule performs substitution for the locally-nameless representation.

```
Reserved Notation "p ---> q" (...).

Inductive reduction : proc -> proc -> Prop :=
| RedInteract : forall m n i v P1 Q1 P2 Q2,
    dual_name m = n
    ->
    ~ (In i (free_ids_proc (((m ? ; P1) +++ P2) ||| ((n ! v; Q1) +++ Q2))))
    ->
      (((m ? ; P1) +++ P2) ||| ((n ! v; Q1) +++ Q2))
      --->
      ((subst_open_proc P1 i 0 v) ||| Q1)
| RedStr : forall P1 P2 Q1 Q2, P1 == P2 -> P2 ---> Q2 -> Q2 == Q1 -> P1 ---> Q1
| RedIsEq : forall k : token, forall P, (IsEq k k P) ---> P
| RedIsNotEq : forall k1 k2 : token, forall P, k1 <> k2 -> (IsNotEq k1 k2 P) ---> P
| RedPar : forall P1 P2 Q, P1 ---> P2 -> (P1 ||| Q) ---> (P2 ||| Q)
| RedNew : forall P Q L,
    (forall i,
      ~ (In i (L ++ (free_ids_proc P)))
      ->
```

```
      open_proc i 0 P ---> open_proc i 0 Q)
    ->
    (New P) ---> (New Q)
where "P ---> Q" := (reduction P Q).
```

The annotated reduction relation is defined in `ResultPreservation.v`. The difference is that reduction yields a visible observation annotation `vo` of type `vis_obs` (see below).

```
Inductive vis_reduction : proc -> proc -> vis_obs -> Prop :=
| VRedInteract : forall m vo n i v P1 Q1 P2 Q2,
    dual_name m = n
    ->
    ~ (In i (free_ids_proc (((m ? ; P1) +++ P2) ||| ((n ! v; Q1) +++ Q2))))
    ->
    mk_vis_obs m v vo
    ->
    vis_reduction
      (((m ? ; P1) +++ P2) ||| ((n ! v; Q1) +++ Q2))
      ((subst_open_proc P1 i 0 v) ||| Q1)
      vo
| VRedStr : forall P1 P2 Q1 Q2 vo,
    P1 == P2 -> (vis_reduction P2 Q2 vo) -> Q2 == Q1 -> (vis_reduction P1 Q1 vo)
| VRedIsEq : forall k : token, forall P,
    vis_reduction (IsEq k k P) P VONone
| VRedIsNotEq : forall k1 k2 : token, forall P,
    k1 <> k2 -> vis_reduction (IsNotEq k1 k2 P) P VONone
| VRedPar : forall P1 P2 Q vo,
    (vis_reduction P1 P2 vo) -> (vis_reduction (P1 ||| Q) (P2 ||| Q) vo)
| VRedNew : forall P Q L vo noc,
    new_obs (free_ids_proc P) vo noc
    ->
    (forall i,
      ~ (In i (L ++ (free_ids_proc P)))
      ->
      (vis_reduction (open_proc i 0 P) (open_proc i 0 Q) (expand_new_obs_choice noc i)))
    ->
    (vis_reduction (New P) (New Q) vo).
```

The visible observation annotations and their components are defined as follows.

```
Inductive vis_obs : Set :=
| VONone : vis_obs
| VOInteract : vis_dir -> free_id -> vis_value -> vis_obs.

Inductive mk_vis_value : value -> vis_value -> Prop :=
```

```
| MVVNm : forall f, mk_vis_value (ValName (Nm f)) VVChan
| MVVCoNm : forall f, mk_vis_value (ValName (CoNm f)) VVChan
| MVVToken : forall k, mk_vis_value (ValToken k) (VVToken k).

Inductive mk_vis_obs : name -> value -> vis_obs -> Prop :=
| MVONm : forall f v vv,
    mk_vis_value v vv -> mk_vis_obs (Nm (Free f)) v (VOInteract VDInp f vv)
| MVOCoNm : forall f v vv,
    mk_vis_value v vv -> mk_vis_obs (CoNm (Free f)) v (VOInteract VDOut f vv).

Definition free_ids_vis_obs (vo : vis_obs) : list free_id :=
  match vo with
    | VONone => nil
    | VOInteract vd f vv => f :: nil
  end.

Inductive new_obs_choice : Set :=
| NOCVisObs : vis_obs -> new_obs_choice
| NOCHidden : vis_dir -> vis_value -> new_obs_choice.

Inductive new_obs : list free_id -> vis_obs -> new_obs_choice -> Prop :=
| NOEq : forall l vo,
    (forall x, In x (free_ids_vis_obs vo) -> In x l) -> new_obs l vo (NOCVisObs vo)
| NOHidden : forall l vd vv,
    new_obs l VONone (NOCHidden vd vv).

Definition expand_new_obs_choice (noc : new_obs_choice) (i : free_id) : vis_obs :=
  match noc with
    | NOCVisObs vo => vo
    | NOCHidden vd vv => VOInteract vd i vv
  end.
```

Chapter 3

Session Types

Data types, messages (expressing direction of an interaction), and session types are defined by mutual induction `TypeAssignmentPoly.v`. The session types can be broken down into general purpose session types and session types that are used for particular examples (illustrating the need for *extensible* session types).

```
Inductive type : Set :=
| TChannel : session -> type
| TSingleton : token -> type

with message : Set :=
| MInp : type -> message
| MOut : type -> message

with session : Set :=
(* general purpose *)
| SEpsilon : session
| SPrefix : message -> session -> session
| SPlus : session -> session -> session
| SSeq : session -> session -> session
| SRep : session -> session
| SDual : session -> session

(* sink *)
| SSink : session

(* uni-forwarder *)
| SFwd : session -> session
| SInOut : session
| SInOut1 : session -> session

(* abp *)
```

```
| SToks : session -> session
| SNack : session -> token -> session -> token -> session
| SNack1 : session -> token -> session -> token -> session
| SAck : session -> token -> session
| SAck1 : session -> token -> session
| SAck2 : session -> token -> session

(* abp_send *)
| SSend : bool -> session (* a fresh channel for passing args *)
(* a channel for passing args where the token has been passed *)
| SSend1 : bool -> session -> session -> session
(* a channel for passing args where the token and "inp" channel has been
   passed, and "err1" is next *)
| SSend2 : bool -> session -> session

(* abp_recv *)
| SRecv : bool -> session (* a fresh channel for passing args *)
(* a channel for passing args where err2 has been passed.
  need bool to bind i, 1st session for s, and 2nd session for t *)
| SRecv1 : bool -> session -> session -> session

(* abp_lossy *)
| SLossy : session (* a fresh channel for passing args *)
(* a channel for passing args where err1 has been passed.
  need to bind s and t: 1st param is s, 2nd is t *)
| SLossy1 : session -> session -> session.
```

The transition relation gives meaning to each of the session-type constructors above, i.e., indicating which inputs / outputs are possible for a channel with that session type. The transitions for the general purpose session type constructors are:

```
Inductive transition : session -> message -> session -> Prop :=
| TRPrefix : forall s m, (SPrefix m s) --m--> s
| TRPlus1 : forall s1 s2 t1 m, s1 --m--> t1 -> (SPlus s1 s2) --m--> t1
| TRPlus2 : forall s1 s2 t2 m, s2 --m--> t2 -> (SPlus s1 s2) --m--> t2
| TRSeq1 : forall s1 s2 t1 m, s1 --m--> t1 -> (SSeq s1 s2) --m--> (SSeq t1 s2)
| TRSeq2 : forall s1 s2 t2 m, terminates s1 -> s2 --m--> t2 -> (SSeq s1 s2) --m--> t2
| TRRep : forall s t m, s --m--> t -> (SRep s) --m--> (SSeq t (SRep s))
| TRDual : forall s t m, s --m--> t -> (SDual s) --(m_dual m)--> (SDual t)
...other transitions hidden...
where "s -- m --> t" := (transition s m t).
```

For the sink example in [GJJ+]:

```
| TRSink : forall s, SSink --(MInp (TChannel s))--> SSink
```

For the unidirectional forwarder example in [GJJ+], the session type `SInOut` is used with single-use channels that carry two channels and are then not used again.

```
| TRFwd : forall s, SFwd s --(MInp (TChannel s))--> SFwd s
| TRInOut : forall s, SInOut --(MInp (TChannel s))--> SInOut1 s
| TRInOut1 : forall s, SInOut1 s --(MInp (TChannel (SDual s)))--> SEpsilon
```

The alternating bit protocol typing in [GJJ+] uses a larger number of transitions with the session type constructors declared above. The transitions are defined in `TypeAssignmentPoly.v`, but to illustrate, we include the transitions for the `SLossy` session type. Note that there are four rules for transitions from `SLossy` for the four "states", but once the choice of session types `s` and `t` has been made, the next transition from `SLossy1 s t` is forced.

```
| TRLossyA : forall i r s t,
    s = (SAck r (token_of_bool i)) ->
    t = (SAck r (token_of_bool i)) ->
    SLossy --(MInp (TChannel (SDual s)))--> SLossy1 s t
| TRLossyB : forall i r k r' s t,
    s = (SNack r k r' (token_of_bool i)) ->
    t = (SNack r k r' (token_of_bool i)) ->
    SLossy --(MInp (TChannel (SDual s)))--> SLossy1 s t
| TRLossyC : forall i r k r' s t,
    s = (SNack r k r' (token_of_bool (negb i))) ->
    t = (SAck r (token_of_bool i)) ->
    SLossy --(MInp (TChannel (SDual s)))--> SLossy1 s t
| TRLossyD : forall i r k r' s t,
    s = (SNack r k r' (token_of_bool i)) ->
    t = (SAck r' (token_of_bool i)) ->
    SLossy --(MInp (TChannel (SDual s)))--> SLossy1 s t
| TRLossy1A : forall s t,
    SLossy1 s t --(MInp (TChannel t))--> SEpsilon
```

Chapter 4

Type Assignment

Type assignment and auxiliary definitions are also in `TypeAssignmentPoly.v`. The definitions include:

1. `|-st rho` — the type `rho` is stateless.
2. `G |-wf` — the context `G` is well-formed.
3. `G |-part GL |$| GR` — the context `G` can be partitioned as `GL` and `GR`, where a context entry is allowed to appear in both `GL` and `GR` if it has the same type on both sides and the type is stateless.
4. `G |-v u : rho` — the value `u` has type `rho` in the context `G`.
5. `G |-p P` — the process `P` is well-typed in the context `G`.

The type assignment rules for parallel composition, sum, (mis)matching, repetition, and zero are straightforward.

```
| TypPar :
  forall G GL GR P Q,
      G |-part GL |$| GR
      ->
      GL |-p P
      ->
      GR |-p Q
      ->
      G |-p (P ||| Q)
| TypSum : forall G P Q,
      G |-p P
      ->
      G |-p Q
      ->
      G |-p P +++ Q
```

```
| TypIsEq :
    forall G P u v K L,
      G |-v u : TSingleton K
      ->
      G |-v v : TSingleton L
      ->
      free_values_in_context G P
      ->
      (K = L -> G |-p P)
      ->
      G |-p (IsEq u v P)
| TypIsNotEq :
    forall G P u v K L,
      G |-v u : TSingleton K
      ->
      G |-v v : TSingleton L
      ->
      free_values_in_context G P
      ->
      (K <> L -> G |-p P)
      ->
      G |-p (IsNotEq u v P)
| TypRep :
    forall G G' P,
      G |-wf
      ->
      CTX.Subset G' G
      ->
      (forall u rho, CTX.In (u, rho) G' -> |-st rho)
      ->
      G' |-p P
      ->
      G |-p (Rep P)
| TypZero :
    forall G,
      G |-wf
      ->
      G |-p Zero
```

The rule for output expresses different possibilities for the constraints on inequality between the value transmitted and the channel upon which it is sent, whether the value is removed from the context when typing the continuation process, and whether the type of the transmitted value is stateless.

```
| TypPrefixOutput :
  forall G G' G'' u v P s rho t,
    (transition s (MOut rho) t)
```

```
    ->
    (u <> v \/ |-st rho)
    ->
    (G |-v u : TChannel s)
    ->
    (G |-v v : rho)
    ->
    (G' = CTX.remove (v, rho) G \/ (G' = G /\ |-st rho))
    ->
    G'' = ctx_replace u (TChannel s) (TChannel t) G'
    ->
    G'' |-p P
    ->
    G |-p (u ! v ; P)
```

The rule for input insists that the continuation process is typed under the assumption that there is a transition from the session of the input channel. For the degenerate case when there are no transitions from the session type of the input channel, we insist that the continuation process does not have free values outisde the context. Note that `open_proc` is used again with `x` and `L` for cofinite quantification in the locally-nameless style.

```
| TypPrefixInput :
  forall G u P s L,
    (G |-v u : TChannel s)
    ->
    (* The following condition controls free_values when there are no
       transitions from s.
    *)
    free_values_in_context G P
    ->
    (forall G' rho t x,
      ~ (In x L)
      ->
      (transition s (MInp rho) t)
      ->
      G' = (CTX.add (ValVariable (Var (Free x)), rho)
             (ctx_replace u (TChannel s) (TChannel t)
               G))
      ->
      (G' |-p (open_proc x 0 P)))
    ->
    (G |-p (u ? ; P))
```

Finally, the typing rule for the `New` process is:

```
| TypNew :
```

```
forall G P s L,
  (forall x G',
    ~ (In x L)
    ->
    G' = (CTX.add (ValName (Nm (Free x)), TChannel s)
           (CTX.add (ValName (CoNm (Free x)), TChannel (SDual s))
             G))
    ->
    G' |-p open_proc x 0 P)
  ->
  G |-p (New P)
```

Chapter 5

Typing of Examples

The sink, unidirectional forwarder, bidirectional forwarder, and alternating bit protocol examples in [GJJ+] have been formalized in Coq. All the examples can be found in the source files with names of the form Example*.v. Here we highlight parts of the process definitions and the corresponding typing result. In the discussion below, we use some channel names, which we define as

```
Definition namec : value := (Nm (Free "c")).
Definition named : value := (Nm (Free "d")).
```

5.1 Sink Process

The process for the sink example is defined in `ExampleSinkTyping.v` as:

```
Definition sink : proc :=
  New (((CoNm (Bound 0)) ! namec ; Zero)
        |||
       (Rep (Nm (Bound 0) ? ; ((Var (Bound 0)) ? ;
              ((CoNm (Bound 2)) ! (Var (Bound 1)); Zero))))).
```

Note the use of deBruijn indices for both names and variables.

The typing result is found in the same file. The statement of the typing result is:

```
Coq < Check sink_typing.
sink_typing
     : forall s : session,
       CTX.add (ExampleCommon.namec, TChannel s) CTX.empty |-p sink
```

Note that the type assignment result quantifies over all session types `s`.

5.2 Unidirectional Forwarder Process

The unidirectional forwarder process is defined in `ExampleUniForwarderMultiStepTyping.v`.

```
Definition uni_forwarder_aux_init ARGS i o :=
  New (ARGS ! ((Nm (Bound 0))); (((CoNm (Bound 0))) ! i;
    ((CoNm (Bound 0))!o; Zero))).

Definition uni_forwarder_aux_rep :=
  Nm (Bound 0) ? ; (Var (Bound 0) ?; (Var (Bound 1) ?; (Var (Bound 1) ?;
    (Var (Bound 1) ! (Var (Bound 0));
      uni_forwarder_aux_init (CoNm (Bound 5)) (Var (Bound 3))
        (Var (Bound 2))))))).

(* Def fwd(in,out) = in?z; out!z; fwd(in,out) *)
Definition uni_forwarder_multi_step i o :=
  New ((uni_forwarder_aux_init (CoNm (Bound 1)) i o)
    ||| (Rep (uni_forwarder_aux_rep))).
```

The typing result is found in the same file. The result is:

```
Coq < Check uni_forwarder_multi_step_zero_typing.
uni_forwarder_multi_step_zero_typing
     : forall s : session,
       CTX.add (ExampleCommon.namec, TChannel s)
         (CTX.add (ExampleCommon.named, TChannel (SDual s)) CTX.empty)
       |-p uni_forwarder_multi_step ExampleCommon.namec
            ExampleCommon.named
```

Again the type assignment result quantifies over all session types `s`, and so the forwarder can be used at any session type as described in [GJJ$^+$].

5.3 Recursion

[GJJ$^+$] defines some syntactic sugar to make use of recursive processes. This syntax is defined in the file `ExampleRecursion.v`. We define a new Set, which is similar to a proc but allows recursion.

```
Inductive procR : Set :=
| InputR : value -> procR -> procR
| OutputR : value -> value -> procR -> procR
| IsEqR : value -> value -> procR -> procR
| IsNotEqR : value -> value -> procR -> procR
| NewR : procR -> procR
| ParR : procR -> procR -> procR
| SumR : procR -> procR -> procR
| RepR : procR -> procR
```

```
| ZeroR : procR
| RecurseR : string -> list value -> procR.
```

We transform `procR` to `proc` by using

```
Fixpoint transformProcR (P : procR) : proc :=
  match P with
    | InputR u P' => u ? ; transformProcR P'
    | OutputR u v P' => u ! v ; transformProcR P'
    | IsEqR u v P' => IsEq u v (transformProcR P')
    | IsNotEqR u v P' => IsNotEq u v (transformProcR P')
    | NewR P' => New (transformProcR P')
    | ParR P1 P2 => (transformProcR P1) ||| (transformProcR P2)
    | SumR P1 P2 => (transformProcR P1) +++ (transformProcR P2)
    | RepR P' => ! (transformProcR P')
    | ZeroR => Zero
    | RecurseR name us => send_args name us
  end.
```

These definitions, along with some auxilary functions also found in `ExampleRecursion.v`, allow us to define and type the following two examples the way that [GJJ$^+$] states.

5.4 Bidirectional Forwarder Process

The bidirectional forwarder process is defined and typed in `ExampleBiForwarderTyping.v`. This forwarder process is similar to the forwarder found in `ExampleUniForwarderMultiStepTyping.v`, with the added ability of the reciever to send messages back as well. The bi-forwarder is defined as

```
(* proc bifwd(right, left) =
     (left?x.right!x.bifwd(left, right))
     +(right?x.left!x.bifwd(left, right)) in bifwd(left, right)
*)
Definition bifwd_procR :=
SumR
(InputR (var_value_of_string "left")
  (OutputR (var_value_of_string "right") (ValVariable (Var (Bound 0)))
          (RecurseR "bifwd" ((var_value_of_string "left")
            :: (var_value_of_string "right") :: nil))))
(InputR (var_value_of_string "right")
  (OutputR (var_value_of_string "left") (ValVariable (Var (Bound 0)))
          (RecurseR "bifwd" ((var_value_of_string "left")
            :: (var_value_of_string "right") :: nil)))).
```

The type of this process is proved in the `bifwd_typing` lemma.

```
Coq < Check bifwd_typing.
bifwd_typing
     : forall s : session,
       CTX.add (Nm (Free "left"), TChannel s)
         (CTX.add (CoNm (Free "right"), TChannel (SDual s)) CTX.empty)
       |-p bifwd (Nm (Free "left")) (CoNm (Free "right"))
```

As stated in [GJJ+], this process has the same type as the uni-directional forwarder and is quantified over all session types `s`, which allows this forwarder to also be used at any session type.

5.5 Alternating Bit Protocol Process

The alternating bit protocol process is defined in `ExampleABP.v`.

```
Definition abp inp out :=
  New
    (New
      (    (inp? ; (abp_send (ValVariable (Var (Bound 0))) inp (ValName (Nm (Bound 2)))))
       ||| (abp_lossy (ValName (CoNm (Bound 1))) (ValName (Nm (Bound 0))))
       ||| (abp_recv (ValName (CoNm (Bound 0))) out))).
```

The two fresh channels represent the lossy channels connecting the sender process to the lossy process and the lossy process to the receiver process. The `abp_send`, `abp_lossy`, and `abp_recv` component processes are respectively defined in `ExampleABPSendBase.v`, `ExampleABPLossyBase.v`, and `ExampleABPRecvBase.v`. Their definitions use the `procR` syntactic sugar for recursion discussed above which we translate into the process language discussed at the beginning of this documentation.

The typing of the alternating bit protocol process is developed in a number of the `ExampleABP*.v` files. The main result, found in `ExampleABP.v`, is the following typing:

```
Coq < Check abp_typing.
abp_typing
     : forall s : session,
       CTX.add (Nm (Free "in"), TChannel (SToks s))
         (CTX.add (Nm (Free "out"), TChannel (SDual (SToks s)))
            CTX.empty) |-p abp (Nm (Free "in")) (Nm (Free "out"))
```

where `SToks s` ensures that only tokens are passed along a channel of this type. As the ABP protocol can resend messages internally, a more general session type, which includes names, is not valid due to this non-linear usage of channel names.

Chapter 6

Results

We have established a subject reduction (preservation result) for the extensible polymorphic session type system described here. The subject reduction result has been established for the annotated reduction relation.

In the development of the subject reduction result, the main intermediate results are strengthening:

```
Coq < Check strengthening.
strengthening
     : forall (G1 G2 : ctx) (P : proc),
       (G2 |-p P) ->
       CTX.Subset G1 G2 -> free_values_in_context G1 P -> G1 |-p P
```

Weakening:

```
Coq < Check weakening.
weakening
     : forall (G G1 G2 : ctx) (P : proc),
       (G1 |-p P) -> (G |-part G1(+)G2) -> G |-p P
```

Preservation of typing across structural equivalence:

```
Coq < Check struct_equiv_preserves_typed.
struct_equiv_preserves_typed
     : forall P Q : proc,
       (P == Q) -> forall G : ctx, G |-p P <-> G |-p Q
```

The subject reduction result establishes that if a well-typed process `P` has a reduction to `Q`, then `Q` is also well-typed. Moreover, the context used to type `Q` is related to the the context used to type `P` and the annotation `vo` describing the interaction (i.e., the channel and value involved in an interaction). This relationship is described by the relation `ctx_preservation` defined in `ResultPreservation.v`, and states that the contexts must be equal or must involve a transition for the session types of matching names and conames. The

annotation must be consistent with the relationship between the contexts. In order for this result to hold, it is necessary to demand that the initial context consists only of names and conames (so variables are excluded), and names and conames must have dual session types when both are present. This precondition is encoded in the relation `balanced`.

```
Coq < Check subject_reduction.
subject_reduction
     : forall (G1 : ctx) (P Q : proc) (alpha : obs),
       reduction P Q alpha ->
       (G1 |-p P) ->
       balanced G1 ->
       exists G2 : ctx, (G2 |-p Q) /\ ctx_preservation G1 G2 alpha
```

The runtime safety result shows that no errors can occur in well-typed processes with balanced contexts.

```
Coq < Check runtime_safety.
runtime_safety
     : forall (G : ctx) (P Q : proc) (alphas : list obs),
       (G |-p P) -> balanced G -> reductions P Q alphas -> ~ error Q
```

The conformance result establishes the relationship between the interaction on a channel that occurs in a well-typed process and transitions that are possible for the session type associated with the channel.

```
Coq < Check conformance.
conformance
     : forall (G : ctx) (P Q : proc) (f : free_id)
         (s : session) (alphas : list obs),
       reductions P Q alphas ->
       (G |-p P) ->
       balanced G ->
       CTX.In (Nm (Free f), TChannel s) G ->
       traces f (project f alphas) s
```

Bibliography

[GJJ+] Matthew Goto, Radha Jagadeesan, Alan Jeffrey, Corin Pitcher, and James Riely. An extensible approach to session polymorphism. `http://fpl.cs.depaul.edu/projects/xpol`. Under consideration for publication in Math. Struct. in Comp. Science.