A Beluga Formalization of the Harmony Lemma in the π-Calculus

Gabriele Cecilia
Dipartimento di Matematica
Università degli Studi di Milano, Italy

Alberto Momigliano
Dipartimento di Informatica,
Università degli Studi di Milano, Italy

The “Harmony Lemma”, as formulated by Sangiorgi & Walker, establishes the equivalence between the labelled transition semantics and the reduction semantics in the π-calculus. Despite being a widely known and accepted result for the standard π-calculus, a comprehensive formal or informal proof has never been carried out, and its validity may not be immediately apparent when considering extensions of the π-calculus. Contributing to the second challenge of the Concurrent Calculi Formalization Benchmark — a set of challenges tackling the main issues related to the mechanization of concurrent systems — we present a formalization of this result for the fragment of the π-calculus examined in the Benchmark. Our formalization is implemented in Beluga and draws inspiration from the HOAS formalization of the LTS semantics popularized by Honsell et al. In passing, we introduce a couple of useful encoding techniques for handling telescopes and lexicographic induction.

1 Introduction

At page 51 of their “bible” on the π-calculus [17], Sangiorgi & Walker state the *Harmony Lemma*, regarding the relationship between the reduction semantics with the transitional one (LTS). The sketch of the proof starts as follows:

> *Rather than giving the whole (long) proof, we explain the strategy and invite the reader to check some of the details [...]*

While this informal style of proof, akin to the infamous “proof on a napkin” championed by de Millo and colleagues[1], may be suitable for a (long) textbook, it might not be applicable to emerging calculi with more unconventional operational semantics. Although the theorem is undisputed within the well-established framework of the π-calculus, this assurance may not extend to these developing calculi. In such instances, a more rigorous approach, potentially in the form of a machine-checked proof, is advisable.

These considerations are of course not novel: they have been prominently argued for in the POPLMark challenge [2] and subsequent follow-ups [6, 1]. The recent Concurrent Calculi Formalization Benchmark [9] (CCFB in brief) introduces a new collection of benchmarks addressing challenges encountered during the mechanization of models of concurrent and distributed programming languages, with an emphasis on process calculi. As with POPLMark, the idea is to explore the state of the art in the formalization in this subarea, finding the best practices to address their typical issues and improving the tools for their mechanization.

CCFB considers in isolation three aspects that may be problematic when mechanizing concurrency theory: *linearity*, *scope extrusion*, and *coinductive reasoning*. Scope extrusion is, of course, the method by which a process can transfer restricted names to another process, as long as the restriction can be safely expanded to include the receiving process. This phenomenon has been captured in two different, yet equivalent ways of formulating the operational semantics of the π-calculus:

[1] “Social Processes and Proofs of Theorems and Programs”, CACM 22-5, 1979.

To appear in EPTCS.

1. a reduction system, which avoids explicit reasoning about scope extrusion by using structural congruence;
2. a labelled transition system, which introduces a new kind of action to handle extrusion directly: in doing so, it breaks shared conventions such as α-equivalence.[2]

The second challenge in the Concurrent Calculi Formalization Benchmark (CCFB.2) consists in mechanizing these two operational semantics and relating them via the aforementioned Harmony Lemma.

Obviously, we are not the first to address the mechanization of the π-calculus: given the challenges that it poses (various kind of binders with somewhat unusual properties compared to the λ-calculus), there is a long tradition starting with [10] and mostly developed with encodings based on first-order syntax such as de Brujin indexes — see [9] for a short review of the literature w.r.t. scope extrusion. As often remarked, concrete encodings will get you there, but not effortlessly: an estimation of 75% of the development being devoted to the infrastructure of names handling is not uncommon.

It is not surprising that specifications based on higher-order abstract syntax (HOAS) soon emerged, first only as animations, see [11] in λProlog and [7] in LF. Moving to meta-reasoning, we can roughly distinguish two main approaches:

1. "squeezing" HOAS into a general proof assistant: there is a plethora of approaches, but w.r.t the π-calculus this has been investigated by Despeyroux [5] and then systematically by Honsell and his colleagues, starting with [8] and then addressing other calculi;
2. the Pfenning-Miller "two-level approach" of separating the specification from the reasoning logic, whose culmination, as far as the π-calculus is concerned, is the most elegant version presented in [19] and later implemented in Abella.

We fall in the second camp and we offer a Beluga [16] mechanization of CCFB.2 together with a detailed informal proof, filling all the gaps left by the quoted sketch. Along the way, we introduce (or simply rediscover) a couple of Beluga tricks to encode *telescopes* (i.e. n-ary sequences of binders) and to simulate lexicographic induction. We also prove another folk result, namely the equivalence between the early and late LTS, as well as what is sometimes called "internal adequacy" [8], that is the equivalence between the LTS encoding from the latter paper with the one in [19].

Informal and formal proofs in all their glory can be found here [4]. In the following we will assume a working knowledge of Beluga, both of its syntax and more importantly of its approach to proof checking.

2 The π-Calculus and its Operational Semantics

In this section, we quickly recall the main notions involved, so as to make the paper self-contained. For more details see [17].

2.1 Syntax

We assume the existence of a countably infinite set of *names*, ranged over by $x, y, \ldots$ We make no other assumption about names, since the syntax of *processes* in CCFB.2 does not consider (mis)match. In fact, to concentrate in isolation on scope extrusion, sums and replications are ignored as well:

$$P, Q \ ::= \ \mathbf{0} \ \mid \ x(y).P \ \mid \ \bar{x}y.P \ \mid \ (P \mid Q) \ \mid \ (\nu x)P$$

[2] There are also intermediate approaches that save α-equivalence, such as Parrow's LTS with structural congruence [14] or Milner's notion of abstraction and concretion as formalized for example in [3].

The input prefix $x(y).P$ and the restriction $(\nu y)P$ both bind the name y in P. Any other occurrence of names in a process is free. The sets of free and bound names occurring in a process ($\mathsf{fn}(P)$ and $\mathsf{bn}(P)$ respectively) are defined as usual.

In the mathematical presentation of the operational semantics, we adopt the following slightly weaker variable convention[3]: 1) given a process, it is *possible* to α-rename the bound occurrences of variables within it; 2) the bound names of any processes or actions under consideration *can* be chosen different from the names occurring free in any other entities under consideration.

2.2 Reduction Semantics

We define the *structural congruence* relation $\equiv$ and the *reduction* relation $\rightarrow$ as the smallest binary relations over processes, satisfying the axioms in Fig. 1. The notation $Q\{y/z\}$ represents capture-avoiding substitution of y for z in the process Q. Note the we have chosen to present congruence as the compatible refinement of the six basic axioms, rather than using process *contexts* as in [17], since the latter tend to be problematic w.r.t. a HOAS formalization.

$$\textsc{Par-Assoc}\ \frac{}{P \mid (Q \mid R) \equiv (P \mid Q) \mid R} \qquad \textsc{Par-Unit}\ \frac{}{P \mid \mathbf{0} \equiv P} \qquad \textsc{Par-Comm}\ \frac{}{P \mid Q \equiv Q \mid P}$$

$$\textsc{Sc-Ext-Zero}\ \frac{}{(\nu x)\mathbf{0} \equiv \mathbf{0}} \qquad \textsc{Sc-Ext-Par}\ \frac{x \notin \mathsf{fn}(Q)}{(\nu x)P \mid Q \equiv (\nu x)(P \mid Q)} \qquad \textsc{Sc-Ext-Res}\ \frac{}{(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P}$$

$$\textsc{C-In}\ \frac{P \equiv Q}{x(y).P \equiv x(y).Q} \qquad \textsc{C-Out}\ \frac{P \equiv Q}{\bar{x}y.P \equiv \bar{x}y.Q} \qquad \textsc{C-Par}\ \frac{P \equiv P'}{P \mid Q \equiv P' \mid Q} \qquad \textsc{C-Res}\ \frac{P \equiv Q}{(\nu x)P \equiv (\nu x)Q}$$

$$\textsc{C-Ref}\ \frac{}{P \equiv P} \qquad \textsc{C-Sym}\ \frac{P \equiv Q}{Q \equiv P} \qquad \textsc{C-Trans}\ \frac{P \equiv Q \qquad Q \equiv R}{P \equiv R}$$

$$\textsc{R-Com}\ \frac{}{\bar{x}y.P \mid x(z).Q \;\rightarrow\; P \mid Q\{y/z\}} \qquad \textsc{R-Par}\ \frac{P \rightarrow Q}{P \mid R \rightarrow Q \mid R}$$

$$\textsc{R-Res}\ \frac{P \rightarrow Q}{(\nu x)P \rightarrow (\nu x)Q} \qquad \textsc{R-Struct}\ \frac{P \equiv P' \qquad P' \rightarrow Q' \qquad Q' \equiv Q}{P \rightarrow Q}$$

Figure 1: Congruence and reduction rules.

[3] Variable conventions are used in a rather loose way in the literature, e.g. Parrow and Sangiorgi & Walker adopt the same convention, but end up with different provisos in the operational semantics rules.

2.3 Labelled Transition System Semantics

The syntax of *actions* is the following:

$$\alpha := x(y) \mid \bar{x}y \mid \bar{x}(y) \mid \tau$$

In the input action $x(y)$ and in the bound output action $\bar{x}(y)$, the name x is free and y is bound; in the free output action $\bar{x}y$, both x and y are free. The *transition* relation $\xrightarrow{(-)}$ is the smallest relation which satisfies the rules in Fig. 2.

$$\textsc{S-In}\quad \frac{}{x(z).P \xrightarrow{x(z)} P} \qquad\qquad \textsc{S-Out}\quad \frac{}{\bar{x}y.P \xrightarrow{\bar{x}y} P}$$

$$\textsc{S-Par-L}\quad \frac{P \xrightarrow{\alpha} P' \qquad \mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad\qquad \textsc{S-Par-R}\quad \frac{Q \xrightarrow{\alpha} Q' \qquad \mathsf{bn}(\alpha) \cap \mathsf{fn}(P) = \emptyset}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

$$\textsc{S-Com-L}\quad \frac{P \xrightarrow{\bar{x}y} P' \qquad Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'\{y/z\}} \qquad\qquad \textsc{S-Com-R}\quad \frac{P \xrightarrow{x(z)} P' \qquad Q \xrightarrow{\bar{x}y} Q'}{P \mid Q \xrightarrow{\tau} P'\{y/z\} \mid Q'}$$

$$\textsc{S-Res}\quad \frac{P \xrightarrow{\alpha} P' \qquad z \notin \mathsf{n}(\alpha)}{(\nu z)P \xrightarrow{\alpha} (\nu z)P'} \qquad\qquad \textsc{S-Open}\quad \frac{P \xrightarrow{\bar{x}z} P' \qquad z \neq x}{(\nu z)P \xrightarrow{\bar{x}(z)} P'}$$

$$\textsc{S-Close-L}\quad \frac{P \xrightarrow{\bar{x}(z)} P' \qquad Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} (\nu z)(P' \mid Q')} \qquad\qquad \textsc{S-Close-R}\quad \frac{P \xrightarrow{x(z)} P' \qquad Q \xrightarrow{\bar{x}(z)} Q'}{P \mid Q \xrightarrow{\tau} (\nu z)(P' \mid Q')}$$

Figure 2: Transition rules.

Unlike the reduction semantics, the transitional semantics directly addresses scope extrusion via the two S-CLOSE rules in interaction with S-OPEN: recall how the former rules are *not* closed under α-conversion, since the bound name z must occur free in the other premise.

The LTS introduced here is the *late* semantics, as opposed to the *early* one adopted by the Benchmark. However, as remarked in [14], "it is a matter of taste which semantics to adopt". We indeed prove this equivalence in Appendix A.

2.4 The Harmony Lemma

In [17], the Harmony Lemma reads as:

i. $P \equiv\xrightarrow{\alpha} Q$ implies $P \xrightarrow{\alpha}\equiv Q$.

ii. $P \xrightarrow{\tau}\equiv Q$ iff $P \rightarrow Q$.

The first assertion is a direct consequence of Lemma 2.6, which is instrumental to prove the right-to-left direction of the equivalence result. The latter breaks down into the following theorems:

1. Every transition through a τ action corresponds to a reduction;
2. Given a reduction of P to Q, P is able to make a τ-transition to some Q' congruent to Q.

In the interest of setting the stage for anybody who wishes to give a solution to CCFB.2, we start by stating a few technical lemmas about substitutions that are used in both directions of the Harmony Lemma, while being often left unsaid.

Lemma S1 *$Q\{x/x\} = Q$.*

Lemma S2 *If $x \notin \mathsf{fn}(Q)$, then $Q\{y/x\} = Q$.*

These two lemmas are proved by induction on the structure of the process Q. A consequence of the latter is the following: if $x \notin \mathsf{fn}(Q)$, then $P\{y/x\} \mid Q = (P \mid Q)\{y/x\}$.

Finally, we state a stability result for structural congruence under substitutions, used in the second direction only:

Lemma S3 *If $P \equiv Q$, then $P\{y/x\} \equiv Q\{y/x\}$.*

This lemma is proved by induction on the structure of the given derivation.

2.4.1 Theorem 1: τ-Transition Implies Reduction

The proof of the first direction relies on three key lemmas which describe rewriting (up to structural congruence) of processes involved in input and output transitions.

Lemma 1.1 *If $Q \xrightarrow{x(y)} Q'$ then there exist a finite (possibly empty) set of names $w_1,\dots,w_n$ (with $x,y \neq w_i \ \forall i = 1,\dots,n$) and two processes R,S such that $Q \equiv (\nu w_1)\dots(\nu w_n)(x(y).R \mid S)$ and $Q' \equiv (\nu w_1)\dots(\nu w_n)(R \mid S)$.*

Lemma 1.2 *If $Q \xrightarrow{\bar{x}y} Q'$ then there exist a finite (possibly empty) set of names $w_1,\dots,w_n$ (with $x,y \neq w_i \ \forall i = 1,\dots,n$) and two processes R,S such that $Q \equiv (\nu w_1)\dots(\nu w_n)(\bar{x}y.R \mid S)$ and $Q' \equiv (\nu w_1)\dots(\nu w_n)(R \mid S)$.*

Lemma 1.3 *If $Q \xrightarrow{\bar{x}(z)} Q'$ then there exist a finite (possibly empty) set of names $w_1,\dots,w_n$ (with $x \notin \{z,w_1,\dots,w_n\}$) and two processes R,S such that $Q \equiv (\nu z)(\nu w_1)\dots(\nu w_n)(\bar{x}z.R \mid S)$ and $Q' \equiv (\nu w_1)\dots(\nu w_n)(R \mid S)$.*

These three lemmas are proved by induction over the structure of the given transition.

Theorem 1 *$P \xrightarrow{\tau} Q$ implies $P \rightarrow Q$.*

The theorem is proved by induction on the structure of the given transition. If the latter consists of an explicit interaction of processes in a parallel composition, we apply the aforementioned lemmas to rewrite processes involved in specific transitions up to congruence; we then construct the desired reduction through a chain of congruence and reduction rules.

Corollary 1.1 *$P \xrightarrow{\tau}\equiv Q$ entails $P \rightarrow Q$.*

2.4.2 Theorem 2: Reduction Implies τ-Transition

The other direction starts with five technical lemmas regarding free and bound names in specific transitions. They are instrumental, together with the variable convention, to the firing of the appropriate transitions.

Lemma 2.1 *If $P \xrightarrow{\bar{x}y} P'$, then $x, y \in fn(P)$.*

Lemma 2.2 *If $P \xrightarrow{x(y)} P'$, then $x \in fn(P)$.*

Lemma 2.3 *If $P \xrightarrow{\bar{x}(z)} P'$, then $x \in fn(P)$ and $z \in bn(P)$.*

Lemma 2.4 *If $P \xrightarrow{\alpha} P'$, $x \notin n(\alpha)$ and $x \notin fn(P)$, then $x \notin fn(P')$.*

Lemma 2.5 *If $P \equiv P'$, then $x \in fn(P) \Leftrightarrow x \in fn(P')$.*

The first four lemmas follow by induction over the structure of the given transition. The last by induction on the congruence judgment.

The next key ingredient is establishing that structural congruence is a strong late bisimulation.

Lemma 2.6 *Let $P \equiv Q$.*

1. *If $P \xrightarrow{\alpha} P'$, then there exists a process Q' such that $Q \xrightarrow{\alpha} Q'$ and $P' \equiv Q'$.*
2. *If $Q \xrightarrow{\alpha} Q'$, then there exists a process P' such that $P \xrightarrow{\alpha} P'$ and $P' \equiv Q'$.*

The two statements need to be proven at the same time by mutual induction over the derivation of the congruence judgment and case analysis on the given transition.

Finally, a rewriting lemma for reduction, again proven by induction on the structure of the given reduction:

Lemma 2.7 *If $P \to Q$ then there exist three names x, y and z, a finite (possibly empty) set of names $w_1, \ldots, w_n$ and three processes R_1, R_2 and S such that $P \equiv (\nu w_1)\ldots(\nu w_n)((\bar{x}y.R_1 \mid x(z).R_2) \mid S)$ and $Q \equiv (\nu w_1)\ldots(\nu w_n)((R_1 \mid R_2\{y/z\}) \mid S)$.*

Theorem 2 *$P \to Q$ implies the existence of a Q' such that $P \xrightarrow{\tau} Q'$ and $Q \equiv Q'$.*

The proof follows immediately from the application of Lemmas 2.6 and 2.7.

3 Beluga Formalization

This section provides an overview of the formalization of the definitions and proofs introduced in the previous section with the proof assistant Beluga. The complete formalization is accessible at [4].

3.1 Syntax

Fig. 3 displays the syntax of names and processes. Since names are just an infinite set, we encode them with an LF type `names` without any constructor, which will be extended dynamically in the operational semantics. This is made possible by the declaration

```
schema ctx = names;
```

indicating that all relevant judgments involving open terms (processes) are formulated in contexts categorized by the above schema.

Processes are encoded using (weak) HOAS, as well trodden in the literature: input and restrictions abstract over names, in particular the input process $x(y).P$ is encoded by the term `p_in X \y.(P y)`, where the bound name y in P is represented by the implicit argument of the LF function `\y.(P y)`. As usual, α-renaming and capture-avoiding substitutions are automatically implemented by the meta-language. From now on, semicolons and infix constructor declarations will be omitted from code snippets for brevity.

```
LF names: type =
;
LF proc: type =
  | p_zero: proc
  | p_in: names → (names → proc) → proc
  | p_out: names → names → proc → proc
  | p_par: proc → proc → proc --infix p_par 11 left.
  | p_res: (names → proc) → proc
;
```

Figure 3: Encoding of names and processes.

3.2 Reduction Semantics

Congruence and reduction are encoded by the type families `cong` and `red` respectively, as presented in Fig. 4. As usual, we use universal quantification such as `{x:names}` to descend into binders, e.g. in the compatibility rule for restriction. Scope extension is realized in rule `sc_ext_par` by simply *not* having Q depend on the restricted channel, hence meeting the side condition $x \notin \mathsf{fn}(Q)$ in the SC-EXT-PAR automatically. In the same vein, substitution in rule R-COM is encoded by meta-level β-reduction.

3.3 Labelled Transition System Semantics

We follow Honsell et al. [8] for the encoding of the late LTS semantics. We declare two different relations for transitions via free and bound actions. The result of a free transition is a process, while the result of a bound transition is a process abstraction: instead of explicitly stating the bound name involved in the transition, that name is the argument of the aforementioned function. This is reflected by the encoding of free and bound actions, which only mention the free names involved. Fig. 5 shows the types `f_act` and `b_act` encoding free and bound actions and the two mutually defined type families `fstep` and `bstep` which encode free and bound transitions respectively[4]. Note that none of the side conditions of the transition rules needs to be explicitly stated, nor do we need the axioms and additional freshness predicates as in [8].

HOAS encodings customarily come with an (informal) *adequacy* proof, ensuring that there is a compositional bijection between the mathematical model and its encoding (in canonical form). While this is fairly obvious for processes, congruence and reduction, it is less so w.r.t. the LTS semantics. Luckily,

[4] Interestingly, a similar approach is pursued by Cheney in its nominal encoding in αProlog, see `https://homepages.inf.ed.ac.uk/jcheney/programs/aprolog/examples/picalc.apl`

```
% Structural Congruence
LF cong: proc → proc → type =
% Abelian Monoid Laws for Parallel Composition
  | par_assoc: cong (P p_par (Q p_par R)) ((P p_par Q) p_par R)
  | par_unit: cong (P p_par p_zero) P
  | par_comm: cong (P p_par Q) (Q p_par P)
% Scope Extension Laws
  | sc_ext_zero: cong (p_res (\x.p_zero)) p_zero
  | sc_ext_par: cong ((p_res P) p_par Q) (p_res (\x.((P x) p_par Q)))
  | sc_ext_res: cong (p_res \x.(p_res \y.(P x y))) (p_res \y.(p_res \x.(P x y)))
% Compatibility Laws
  | c_in: ({y:names} cong (P y) (Q y)) → cong (p_in X P) (p_in X Q)
  | c_out: cong P Q → cong (p_out X Y P) (p_out X Y Q)
  | c_par: cong P P’ → cong (P p_par Q) (P’ p_par Q)
  | c_res: ({x:names} cong (P x) (Q x)) → cong (p_res P) (p_res Q)
% Equivalence Relation Laws
  | c_ref: cong P P
  | c_sym: cong P Q → cong Q P
  | c_trans: cong P Q → cong Q R → cong P R

% Reduction
LF red: proc → proc → type =
  | r_com: red ((p_out X Y P) p_par (p_in X Q)) (P p_par (Q Y))
  | r_par: red P Q → red (P p_par R) (Q p_par R)
  | r_res: ({x:names} red (P x) (Q x)) → red (p_res P) (p_res Q)
  | r_str: P cong P’ → red P’ Q’ → Q’ cong Q → red P Q
```

Figure 4: Encoding of congruence and reduction.

this has been carefully proven both in [8] and in [19] for a related version. We refer to those papers for further details and to the repository for a Beluga proof of the “internal” adequacy of those two encodings.

3.4 The Harmony Lemma

One of the standard, but still very much appreciated payoff of a HOAS encoding is that most (in fact all but Lemma 2.4) boilerplate lemmas about names, occurrences and substitution vanish. We are referring to the substitution Lemmas S1—S3, as well as the free/bound names Lemmas 2.1, 2.2, 2.3 and 2.5.

3.4.1 Theorem 1: τ-Transition Implies Reduction

We start with the encoding of Lemmas 1.1, 1.2 and 1.3. There are two issues, one standard, the other slightly more challenging. For one, the conclusion of these lemmas includes an existential quantification, and Beluga lacks such a construct; the usual workaround consists of defining a new type family which encodes the existential quantification. Secondly, and more seriously, the statements refer to sequences of binders (here restrictions), sometimes referred to as *telescopes*. We can give a combined solution to these items by encoding them *inductively*, where the base case describes when the existential holds for the empty sequence and the inductive one adds one more binder. More specifically for Lemma 1.1, we say that the congruences ($\star$) ($Q \equiv (\nu w_1)\ldots(\nu w_n)(x(y).R \mid S)$ and $Q' \equiv (\nu w_1)\ldots(\nu w_n)(R \mid S)$) hold for two processes Q and Q' iff one of the following holds:

```
% Free Actions                             % Bound Actions
LF f_act: type =                           LF b_act: type =
  | f_tau: f_act                             | b_in: names → b_act
  | f_out: names → names → f_act             | b_out: names → b_act

% Transition Relation
LF fstep: proc → f_act → proc → type =
  | fs_out: fstep (p_out X Y P) (f_out X Y) P
  | fs_par1: fstep P A P' → fstep (P p_par Q) A (P' p_par Q)
  | fs_par2: fstep Q A Q' → fstep (P p_par Q) A (P p_par Q')
  | fs_com1: fstep P (f_out X Y) P' → bstep Q (b_in X) Q'
    → fstep (P p_par Q) f_tau (P' p_par (Q' Y))
  | fs_com2: bstep P (b_in X) P' → fstep Q (f_out X Y) Q'
    → fstep (P p_par Q) f_tau ((P' Y) p_par Q')
  | fs_res: ({z:names} fstep (P z) A (P' z)) → fstep (p_res P) A (p_res P')
  | fs_close1: bstep P (b_out X) P' → bstep Q (b_in X) Q'
    → fstep (P p_par Q) f_tau (p_res \z.((P' z) p_par (Q' z)))
  | fs_close2: bstep P (b_in X) P' → bstep Q (b_out X) Q'
    → fstep (P p_par Q) f_tau (p_res \z.((P' z) p_par (Q' z)))

and bstep: proc → b_act → (names → proc) → type =
  | bs_in: bstep (p_in X P) (b_in X) P
  | bs_par1: bstep P A P' → bstep (P p_par Q) A \x.((P' x) p_par Q)
  | bs_par2: bstep Q A Q' → bstep (P p_par Q) A \x.(P p_par (Q' x))
  | bs_res: ({z:names} bstep (P z) A (P' z))
    → bstep (p_res P) A \x.(p_res \z.(P' z x))
  | bs_open: ({z:names} fstep (P z) (f_out X z) (P' z)) → bstep (p_res P)(b_out X) P'
```

Figure 5: Encoding of actions and transition.

i. $Q \equiv x(y).R \mid S$ and $Q' \equiv R \mid S$;

ii. $Q \equiv (\nu w)P$, $Q' \equiv (\nu w)P'$ and the congruences $(\star)$ hold for P and P'.

We list the definition of the type family `ex_inp_rew` that encodes the above judgment; the types `ex_fout_rew` and `ex_bout_rew` are defined analogously.

```
LF ex_inp_rew: proc → names → (names → proc) → type =
  | inp_base: Q cong ((p_in X R) p_par S) → ({y:names} (Q' y) cong ((R y) p_par S))
    → ex_inp_rew Q X Q'
  | inp_ind: Q cong (p_res P) → ({y:names} (Q' y) cong (p_res (P' y)))
    → ({w:names} ex_inp_rew (P w) X \y.(P' y w)) → ex_inp_rew Q X Q'
```

We prove each lemma by defining a total recursive function which receives a contextual derivation of an input/free output/bound output transition respectively and returns a contextual object of the corresponding existential type. We provide the proof term of the `bs_in_rew` function, which proves Lemma 1.1, in Fig. 6; the functions encoding Lemmas 1.2 and 1.3 are omitted.

The proof proceeds by induction on the structure of the given assumption. If it has been obtained through the S-IN rule, we are in the base case of the existential with no binders: hence, we conclude immediately modulo symmetry of congruence. In case the input transition has been obtained through the S-PAR-L rule, it can be rewritten as $P \mid R \xrightarrow{x(y)} P' \mid R$, with the transition B1: $P \xrightarrow{x(y)} P'$ as hypothesis. In

```
rec bs_in_rew: (g:ctx) [g ⊢ bstep Q (b_in X) \y.Q’[..,y]]
  → [g ⊢ ex_inp_rew Q X \y.Q’[..,y]] =
/ total b (bs_in_rew _ _ _ _ b) /
fn b ⇒ case b of
   | [g ⊢ bs_in] ⇒ [g ⊢ inp_base (c_sym par_unit) \y.(c_sym par_unit)]
   | [g ⊢ bs_par1 B1]:[g ⊢ bstep (P p_par R) (b_in X) (\y.(P’ p_par (R[..])))] ⇒
     let [g ⊢ D1] = bs_in_rew [g ⊢ B1] in
     let [g ⊢ D2] = bs_in_rew_par1 [g ⊢ R] [g ⊢ D1] in [g ⊢ D2]
   | [g ⊢ bs_par2 B2]:[g ⊢ bstep (R p_par P) (b_in X) (\y.((R[..]) p_par P’))] ⇒
     let [g ⊢ D1] = bs_in_rew [g ⊢ B2] in
     let [g ⊢ D2] = bs_in_rew_par2 [g ⊢ R] [g ⊢ D1] in [g ⊢ D2]
   | [g ⊢ bs_res \y.B[..,y]] ⇒
     let [g,y:names ⊢ D1[..,y]] = bs_in_rew [g,y:names ⊢ B[..,y]] in
     let [g ⊢ D2] = bs_in_rew_res [g,y:names ⊢ D1[..,y]] in [g ⊢ D2]
```

Figure 6: Proof of Lemma 1.1.

such situation we can apply a recursive call of the bs_in_rew function to B1, obtaining an object D1 of type ex_inp_rew encoding the rewriting for P and P'; to conclude, we appeal to the lemma bs_in_rew_par1 in order to unfold D1 and build the required existential object encoding the rewriting for $P \mid R$ and $P' \mid R$. We provide the signature of the above lemma:

```
rec bs_in_rew_par1: (g:ctx) {R:[g ⊢ proc]} [g ⊢ ex_inp_rew Q X \y.Q’[..,y]]
  → [g ⊢ ex_inp_rew (Q p_par R) X \y.(Q’[..,y] p_par R[..])] = ...
```

The other two cases of the main proof follow the same pattern and require similar auxiliary lemmas.

We are now ready to discuss the proof of the the main result of this section, namely that $P \xrightarrow{\tau} Q$ implies $P \rightarrow Q$ (Theorem 1):

```
rec fstep_impl_red: (g:ctx) [g ⊢ fstep P f_tau Q] → [g ⊢ P red Q] = ...
```

The proof proceeds by induction on the derivation f of [g ⊢ fstep P f_tau Q]. Mirroring the informal proof, in certain subcases it is enough to apply a recursive call of the function on a structurally smaller τ-transition and return the desired object of type [g ⊢ P red Q]. In the other subcases, we apply the functions bs_in_rew, fs_out_rew and bs_out_rew on the given input/output/bound output transitions, obtaining the corresponding objects of existential type, viz. the cited ex_inp_rew and its “siblings” ex_fout_rew and ex_bout_rew. Then, we would like to conclude by applying some auxiliary functions which unfold these objects and build the desired reduction. Here, we face a major hurdle, not so much in writing down the proof terms encoding the proof, but in having them checked for *termination*, which is, after all, what guarantees that the inductive structure of the proof is correct. Let us see one of such lemmas, which emerges in the subcase where a S-COM-L rule is applied:

```
rec fs_com1_impl_red: (g:ctx) [g ⊢ ex_fout_rew P1 X Y Q1]
  → [g ⊢ ex_inp_rew P2 X \x.Q2[..,x]]
  → [g ⊢ (P1 p_par P2) red (Q1 p_par Q2[..,Y])] = ...
```

The proof needs to consider both hypotheses, in order to unfold the two existential types – recall that the telescopes force upon us an inductive encoding of those existentials. In other terms, verification of the fact that this function is decreasing would need to appeal to a form of *lexicographic* induction.

Termination checkers are of course incomplete and adopt strict syntactic criteria to enforce it; in particular, Beluga’s checker currently does not support lexicographic induction. One way out is to define the fs_com1_impl_red function so that it is decreasing on the first argument only and it relies on an auxiliary function that addresses its base case and is decreasing on the second argument. The signature of such a function is:

```
rec fs_com1_impl_red_base: (g:ctx) [g ⊢ P2 cong ((p_in X \x.R[..,x]) p_par S)]
  → [g,w:names ⊢ Q2[..,w] cong (R[..,w] p_par S[..])] → [g ⊢ ex_fout_rew P1 X Y Q1]
  → [g ⊢ (P1 p_par P2) red (Q1 p_par Q2[..,Y])] = ...
```

Once these lemmas are in place, the proof of the first direction of the Harmony Lemma follows without any further drama.

3.4.2 Theorem 2: Reduction Implies τ-Transition

In the other direction, HOAS having disposed of most of the technical lemmas about names and substitutions, the workhorses are the reduction rewriting Lemma 2.7 and the congruence-as-bisimilarity Lemma 2.6. Since the former does not introduce new ideas, we discuss it first.

Given its similarity to Lemmas 1.1–1.3, Lemma 2.7 is implemented with the same strategy: we define an existential type ex_red_rew that inductively encodes the existence of the telescopes and the two congruences stated in the conclusion of the lemma.

```
LF ex_red_rew: proc → proc → type =
  | red_base: P cong (((p_out X Y R1) p_par (p_in X R2)) p_par S)
    → Q cong ((R1 p_par (R2 Y)) p_par S) → ex_red_rew P Q
  | red_ind: P cong (p_res P') → Q cong (p_res Q')
    → ({w:names} ex_red_rew (P' w) (Q' w)) → ex_red_rew P Q
```

We then implement some some auxiliary functions to unfold objects of the existential type ex_red_rew in specific subcases, for example:

```
rec red_impl_red_rew_par: (g:ctx) {R:[g ⊢ proc]} [g ⊢ ex_red_rew P Q]
  → [g ⊢ ex_red_rew (P p_par R) (Q p_par R)] = ...
```

Having established those, it is straightforward to prove

```
rec red_impl_red_rew: (g:ctx) [g ⊢ P red Q] → [g ⊢ ex_red_rew P Q] = ...
```

by induction on the structure of the given reduction.

Moving on to the proof of Lemma 2.6, there is a new technicality to address. We have seen how in a HOAS setting the provisos such as “$x \notin \mathsf{fn}(P)$” are realized by P being in the scope of a meta level abstraction binding x, but *not* actually depending on x. Sometimes (see [18] for other instances), we have to convince our proof environment of this non-dependency, which is basically the content of Lemma 2.4, in words: “given a transition $P \xrightarrow{\alpha} Q$ where x does not occur free in P, then x does not occur free in either α and Q”. Since in Beluga judgments over open terms are encapsulated in the context where they make sense, removing these spurious dependencies amounts to “strengthen” such a context, akin to strengthening lemmas in type theory. The lemma more formally reads as:

$$\text{If } \Gamma, x : \mathit{names} \vdash P \xrightarrow{\alpha_x} Q_x, \text{ then there are } \alpha', Q' \text{ such that } \alpha_x = \alpha', Q_x = Q' \text{ and } \Gamma \vdash P \xrightarrow{\alpha'} Q' \tag{1}$$

Not only Beluga does not have existentials (nor connections), but LF also has no built-in equality notion. However, this is easy to simulate via unification, by defining three type families encoding equality of processes, free actions and bound actions respectively. Process equality is defined as:

```
LF eqp: proc → proc → type =
  | prefl: eqp P P
```

Since we are now dealing with a property about a LF contextual object (the initial transition), the statement in (1) has to be encoded at the computation level as an *inductive* type [15]. We list this encoding, omitting the definition of its counterpart for bound transitions ex_str_bstep.

```
inductive ex_str_fstep: (g:ctx) [g,x:names ⊢ fstep P[..] A Q] → ctype =
  | ex_fstep: {F:[g,x:names ⊢ fstep P[..] A Q]} [g ⊢ fstep P A' Q']
    → [g,x:names ⊢ eqf A A'[..]] → [g,x:names ⊢ eqp Q Q'[..]]
    → ex_str_fstep [g,x:names ⊢ F]
```

Note how non-occurrence is realized using *weakening* substitutions, e.g. P[..] denotes that the process P depends only on the variables mentioned in g, excluding x.

The strengthening lemma is implemented through the two following mutually recursive functions:

```
rec strengthen_fstep: (g:ctx) {F:[g,x:names ⊢ fstep P[..] A Q]}
  → ex_str_fstep [g,x:names ⊢ F] = ...
and rec strengthen_bstep: (g:ctx) {B:[g,x:names ⊢ bstep P[..] A \z.Q[..,x,z]]}
  → ex_str_bstep [g,x:names ⊢ B] = ...
```

The proof follows by a long but straightforward induction on the structure of the given transition.

To state Lemma 2.6, we again need to code the existential in its conclusion; however, since the statement involves transitions through a generic action α, we actually require two new type families: one for free transitions and one for bound transitions.

```
LF ex_fstepcong: proc → proc → f_act → proc → type =
  | fsc: fstep Q A Q' → P' cong Q' → ex_fstepcong P Q A P'
LF ex_bstepcong: proc → proc → b_act → (names → proc) → type =
  | bsc: bstep Q A Q' → ({x:names} (P' x) cong (Q' x)) → ex_bstepcong P Q A P'
```

The reader may wonder why the process P is mentioned in the fsc and bsc constructors, as it does not play any role: indeed, it is a trick to please Beluga's coverage checker when this definition is unfolded in the rest of the development.

Lemma 2.6 is encoded through the definition of four mutual recursive functions: since the statement involves transitions via a generic action α, the first two prove the result for free transitions, while the last two demonstrate the result for bound transitions; moreover, since the proof is carried out by concurrently establishing two symmetrical assertions, the odd functions prove the left-to-right statement, while the even ones demonstrate the right-to-left statement. We present the signature of the first function cong_fstepleft_impl_stepright, which proves the left-to-right assertion for free transitions:

```
rec cong_fstepleft_impl_fstepright: (g:ctx) [g ⊢ P cong Q] → [g ⊢ fstep P A P']
  → [g ⊢ ex_fstepcong P Q A P'] = ...
```

Mirroring the informal proof, this lemma is proved by a long induction on the structure of the given congruence; in most of the subcases, case analysis of the given transition is performed as well.

A final ingredient for the proof of Theorem 2 is the auxiliary lemma:

```
rec red_rew_impl_fstepcong: (g:ctx) [g ⊢ ex_red_rew P Q]
  → [g ⊢ ex_fstepcong P P f_tau Q] = ...
```

The proof proceeds by induction on the structure of the given object of type [g ⊢ ex_red_rew P Q]; in both the base case and the inductive case, a key factor consists in the application of the function cong_fstepright_impl_fstepleft.

We are ready to prove Theorem 2:

```
rec red_impl_fstepcong: (g:ctx) [g ⊢ P red Q] → [g ⊢ ex_fstepcong P P f_tau Q] =
/ total r (red_impl_fstepcong _ _ _ r) /
fn r ⇒ let [g ⊢ D1] = red_impl_red_rew r in
       let [g ⊢ D2] = red_rew_impl_fstepcong [g ⊢ D1] in [g ⊢ D2]
```

Given r representing the reduction $P \longrightarrow Q$, we apply the function red_impl_red_rew to it returning an object D1 of type [g ⊢ ex_red_rew P Q] that encodes the following congruences: $P \equiv (\nu w_1)\cdots(\nu w_n)((\bar{x}y.R_1 \mid x(z).R_2) \mid S)$ and $Q \equiv (\nu w_1)\cdots(\nu w_n)((R_1 \mid R_2\{y/z\}) \mid S)$, for some R_1, R_2, S and $w_1,\dots,w_n$. Finally, we invoke the auxiliary function red_rew_impl_fstepcong, which unfolds the argument D1 and returns the desired object of type [g ⊢ ex_fstepcong P P f_tau Q].

4 Evaluation and Conclusions

The reader may wonder "Is that it?" We sympathize with the feeling: what is remarkable in this formalization is how (mostly) uneventful it has been. Once we had settled on using (weak) HOAS and a specialized proof environment such as Beluga — which, given our lineage, was not much of a stretch — the Honsell/Miller encoding of the labelled transition system removed all issues related to scope extrusion and Beluga's remarkable conciseness did the rest, turning 30 pages of LaTeX proofs, which still skip many steps, into some 700 lines of proof terms.

Remarkably, the structure of the formal proof closely mirrors the informal one: having eliminated the 7 technical lemmas thanks to the HOAS encoding, both proofs share the same 6 lemmas, proved in the same fashion. Some parts of the formal proof are covered by 22 additional lemmas which deal with the unfolding of the existential types,[5] while another 4 lemmas result from the mutual recursion induced by the encoding.

In our opinion, this uneventfulness does not trivialize the accomplishment: in our biased opinion, we have provided a compact and elegant solution of a benchmark problem, which, after all, is supposed to be challenging: the fact that this has not been a heroic feat is a testament to the merits of a HOAS encoding and to the long line of research that has made meta-level reasoning over HOAS specifications possible. It also suggests that, once we put on the HOAS "spectacles", the binders of the π-calculus are not that different from those of the λ-calculus, in the sense that, with the right encoding, the meta-level binder will gladly model scope extrusion.

Beluga has shown to be a reliable system: we did stress the termination checker, with a heavy use of mutually defined recursive functions. We managed to get around the current lack of support for lexicographic induction; this technique could be more generally useful, for example to implement results such as the admissibility of cut elimination that do not have a simpler termination proof — and, as such, cannot be fully checked in Beluga. We also established coverage, again getting around the minor glitch that we have mentioned above in the proof of the right-to-left direction of the Harmony Lemma.

Since the benchmark is amenable to a weak HOAS encoding, this begs the question of why not pursue its solution in a general proof assistant such as Coq. While weak HOAS is indeed consistent with monotonic inductive types, it is well known that the full dependent function space of a theory such as the Calculus of Constructions is incompatible with the intensional quantification sported by LF-like type theories. Workarounds exist: the most successful case in point is the "Theory of Contexts" [8], where additional predicates concerning freshness and non-occurrence are added to specifications such as of the LTS. Further, ToC assumes some axioms regulating the properties of names and abstraction over names

[5] Given how widespread existential (and conjunctive) statements are in this development, it would be helpful if Beluga could provide some syntactic sugar, similarly to Agda, and a way to unfold those definitions automatically.

(i.e. “contexts”) in order to reify what LF-like frameworks provide natively. To be fair, it is unclear to us which role ToC would play in a Coq solution of CCFB.2, but for the rest of the Concurrent Benchmark, the outcome is not pretty (believe us, we tried). Of course there is no obstacle in abandoning HOAS for concrete encodings and we are looking forward to comparing such solutions to ours.

Although CCFB.2 does not ask for it, we conjecture that it would be easy, albeit tedious, to extend our solution to account for other features of the π-calculus, namely sums, replication and match. Mismatch, which is handled in [8], is rather problematic in HOAS, since the systems that support it have no native notion of negation.

The adoption of Beluga as a proof environment for this formalization is motivated by our endeavor (together with Pientka’s group) to give an overall HOAS solution to all the challenges listed in CCFB, which include type safety for (linear) session types and turning strong barbed bisimilarity into a congruence. For this, Beluga is a strong candidate: in fact the type safety challenge is already in the bag, thanks to the techniques developed in [18]. The coinduction part is more challenging, but we have a good track record in a similar benchmark [13]. Solving the rest of CCFB in Beluga may also shed some light on the need of the ∇ quantifier [12] as a meta-reasoning tool, or whether Beluga’s use of contextual LF as a specification language seems to suffice.

References

[1] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer & Kathrin Stark (2019): *POPLMark Reloaded: Mechanizing proofs by logical relations*. *J. Funct. Program.* 29:19, doi:10.1017/S0956796819000170.

[2] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich & Steve Zdancewic (2005): *Mechanized Metatheory for the Masses: The POPLMark Challenge*. In Joe Hurd & Tom Melham, editors: *Theorem Proving in Higher Order Logics*, Springer, Berlin & Heidelberg, pp. 50–65, doi:10.1007/115418684.

[3] Jesper Bengtson & Joachim Parrow (2009): *Formalising the pi-calculus using nominal logic*. *Log. Methods Comput. Sci.* 5, doi:10.2168/LMCS-5(2:16)2009.

[4] Gabriele Cecilia (2024): *Formalizing the Operational Semantics of the π-Calculus*. Master’s thesis, Università degli Studi di Milano. Available at `https://github.com/GabrieleCecilia/concurrent-benchmark-solution`.

[5] Joëlle Despeyroux (2000): *A Higher-Order Specification of the pi-Calculus*. In Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses & Takayasu Ito, editors: *Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics, International Conference IFIP TCS 2000, Sendai, Japan, August 17-19, 2000, Proceedings*, *Lecture Notes in Computer Science* 1872, Springer, pp. 425–439, doi:10.1007/3-540-44929-9_30.

[6] Amy P. Felty, Alberto Momigliano & Brigitte Pientka (2018): *Benchmarks for reasoning with syntax trees containing binders and contexts of assumptions*. *Math. Struct. Comput. Sci.* 28(9), pp. 1507–1540, doi:10.1017/S0960129517000093.

[7] Furio Honsell, Marina Lenisa, Ugo Montanari & Marco Pistore (1998): *Final semantics for the pi-calculus*. In David Gries & Willem P. de Roever, editors: *Programming Concepts and Methods, IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods (PROCOMET ’98) 8-12 June 1998, Shelter Island, New York, USA*, *IFIP Conference Proceedings* 125, Chapman & Hall, pp. 225–243.

[8] Furio Honsell, Marino Miculan & Ivan Scagnetto (2001): *π-Calculus in (Co)Inductive Type Theory*. *Theor. Comput. Sci.* 253(2), pp. 239–285, doi:10.1016/S0304-3975(00)00095-5.

[9] Frederik Krogsdal Jacobsen, Marco Carbone, David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, Alberto Momigliano, Luca Padovani, Alceste Scalas, Martin Vassor & Nobuko Yoshida (2024): *The Concurrent Calculi Formalisation Benchmark*. To appear in COORDINATION 2024.

[10] T. F. Melham (1994): *A Mechanized Theory of the π-Calculus in HOL*. *Nordic J. of Computing* 1(1), p. 50–76.

[11] Dale Miller (1994): *Specification of the pi-calculus*. Available at `http://www.lix.polytechnique.fr/Labo/Dale.Miller/lProlog/examples/pi-calculus/toc.html`.

[12] Dale Miller & Alwen Tiu (2005): *A proof theory for generic judgments*. *ACM Trans. Comput. Log.* 6(4), pp. 749–783, doi:10.1145/1094622.1094628.

[13] Alberto Momigliano, Brigitte Pientka & David Thibodeau (2019): *A case study in programming coinductive proofs: Howe's method*. *Math. Struct. Comput. Sci.* 29(8), pp. 1309–1343, doi:10.1017/S0960129518000415.

[14] Joachim Parrow (2001): *An Introduction to the π-Calculus*. In Jan A. Bergstra, Alban Ponse & Scott A. Smolka, editors: *Handbook of Process Algebra*, North-Holland / Elsevier, pp. 479–543, doi:10.1016/B978-044482830-9/50026-6.

[15] Brigitte Pientka & Andrew Cave (2015): *Inductive Beluga: Programming Proofs*. In Amy P. Felty & Aart Middeldorp, editors: *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, *Lecture Notes in Computer Science* 9195, Springer, pp. 272–281, doi:10.1007/978-3-319-21401-6_18.

[16] Brigitte Pientka & Jana Dunfield (2010): *Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description)*. In Jürgen Giesl & Reiner Hähnle, editors: *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, *Lecture Notes in Computer Science* 6173, Springer, pp. 15–21, doi:10.1007/978-3-642-14203-1_2.

[17] Davide Sangiorgi & David Walker (2001): *The π-Calculus - a Theory of Mobile Processes*. Cambridge University Press.

[18] Chuta Sano, Ryan Kavanagh & Brigitte Pientka (2023): *Mechanizing Session-Types Using a Structural View: Enforcing Linearity without Linearity*. *Proc. ACM Program. Lang.* 7(OOPSLA), pp. 235:374–235:399, doi:10.1145/3622810.

[19] Alwen Tiu & Dale Miller (2010): *Proof Search Specifications of Bisimulation and Modal Logics for the π-Calculus*. *ACM Trans. Comput. Log.* 11(2), pp. 13:1–13:35, doi:10.1145/1656242.1656248.

A Appendix: Late vs Early Transitions

An alternative to the late semantics presented in the paper is the *early* semantics. As its name suggests, it is characterized by the fact that substitutions of names received in interaction are performed as soon as possible, namely during the execution of the S-IN rule. The syntax of actions is the same as in the late semantics; however, the name y occurring in an input action $x(y)$ is considered to be free. As for transitions, now denoted as $\xrightarrow{(-)}$, the rules S-IN, the two S-COM rules and the two S-CLOSE are replaced by those in Fig. 7 ("right" rules are omitted for brevity). Note how the S-IN rule now exhibits the substitution of the input name; conversely, in the S-COM rules, both of the names x and y occurring in the actions of the given transitions must coincide and no substitution takes place in the conclusion.

In order to prove the equivalence of the two semantics, we follow Parrow's approach in [14]: namely, our objective is to demonstrate that the two semantics allow to infer the same τ-transitions. Both directions are achieved by induction on the depth of the inference of the given transitions and rely on some additional lemmas, which state a correspondence between input/output transitions of the two semantics

$$\frac{}{x(z).P \xrightarrow{x(y)} P\{y/z\}}\ \textsc{S-In} \qquad \frac{P \xrightarrow{\bar{x}y} P' \qquad Q \xrightarrow{x(y)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}\ \textsc{S-Com-L} \qquad \frac{P \xrightarrow{\bar{x}(z)} P' \qquad Q \xrightarrow{x(z)} Q' \qquad z \notin \mathsf{fn}(Q)}{P \mid Q \xrightarrow{\tau} (\nu z)(P' \mid Q')}\ \textsc{S-Close-L}$$

Figure 7: Early transition semantics rules.

as well. The only non-trivial correspondence lies between input (in the late semantics) and free input (in the early semantics), which is defined as follows:

$$P \xrightarrow{x(y)} Q \text{ iff there are } Q' \text{ and } w \text{ such that } P \xrightarrow{x(w)} Q' \text{ and } Q = Q'\{y/w\}. \tag{2}$$

We begin the Beluga formalization by encoding the two semantics in the same environment (Fig. 8). The type `f_act` presents a new constructor `f_in` for free input actions in the early semantics; as for transitions, the constructors expressing identical rules in the two semantics are omitted for brevity. We observe that the `ebs_in` constructor representing bound input transitions in the early semantics cannot be eliminated, since bound input transitions are needed as premises in the rules introduced by the `efs_close` constructors. For consistency of notation, the types `fstep` and `bstep` for late transitions have been renamed as `late_fstep` and `late_bstep`.

```
% Free Actions                           % Bound Actions
LF f_act: type =                         LF b_act: type =
  | f_in: names → names → f_act            | b_in: names → b_act
  | f_out: names → names → f_act           | b_out: names → b_act
  | f_tau: f_act

% Early Transition Relation
LF early_fstep: proc → f_act → proc → type =
  | efs_in: early_fstep (p_in X P) (f_in X Y) (P Y)
  | efs_com1: early_fstep P (f_out X Y) P’ → early_fstep Q (f_in X Y) Q’
    → early_fstep (P p_par Q) f_tau (P’ p_par Q’)
  | efs_com2: early_fstep P (f_in X Y) P’ → early_fstep Q (f_out X Y) Q’
    → early_fstep (P p_par Q) f_tau (P’ p_par Q’)
  | efs_close1: early_bstep P (b_out X) P’ → early_bstep Q (b_in X) Q’
    → early_fstep (P p_par Q) f_tau (p_res \z.((P’ z) p_par (Q’ z)))
  | efs_close2: early_bstep P (b_in X) P’ → early_bstep Q (b_out X) Q’
    → early_fstep (P p_par Q) f_tau (p_res \z.((P’ z) p_par (Q’ z)))
  ...
and early_bstep: proc → b_act → (names → proc) → type =
  | ebs_in: early_bstep (p_in X P) (b_in X) P
  ...
```

Figure 8: Encoding of actions and early transition.

The next ingredient for the formalization of the semantics equivalence is the definition of the type family `ex_latebs`, which encodes the existence of a late transition such as in the correspondence (2):

```
LF ex_latebs: proc → names → names → proc → type =
  | lbs: late_bstep P (b_in X) \w.(Q’ w) → eqp Q (Q’ Y) → ex_latebs P X Y Q
```

We then list the signature of the functions `finp_earlytolate` and `finp_latetoearly`, encoding the correspondence between free input transitions in the early semantics and input transitions in the late semantics. The correspondence between the other types of actions is performed analogously with two functions for each case; since their formalization is straightforward, it is omitted.

```
rec finp_earlytolate: (g:ctx) [g ⊢ early_fstep P (f_in X Y) Q]
  → [g ⊢ ex_latebs P X Y Q] = ...

rec finp_latetoearly: (g:ctx) {Y:[g ⊢ names]} [g ⊢ ex_latebs P X Y Q]
  → [g ⊢ early_fstep P (f_in X Y) Q] = ...
```

In the second statement, the input name `Y` needs to be passed as an explicit argument, otherwise Beluga would not be able to reconstruct it during some further calls of this lemma. The proofs of both lemmas are straightforward inductions on the given derivation.

We can now state the signature of the functions `tau_earlytolate` and `tau_latetoearly`, which encode the equivalence of the two semantics:

```
rec tau_earlytolate: (g:ctx) [g ⊢ early_fstep P f_tau Q]
  → [g ⊢ late_fstep P f_tau Q] = ...

rec tau_latetoearly: (g:ctx) [g ⊢ late_fstep P f_tau Q]
  → [g ⊢ early_fstep P f_tau Q] = ...
```

The proof follows by induction on the given transition. In case the transition is obtained through a S-COM or S-CLOSE rule, we apply the previously defined lemmas in order to turn an early input/output transition into a corresponding late input/output transition and then conclude.