A Formalization of the Reversible Concurrent Calculus CCSK$^{\mathrm{P}}$ in Beluga

Gabriele Cecilia

School of Computer & Cyber Sciences,
Augusta University, Augusta, USA

gcecilia@augusta.edu

Reversible concurrent calculi are abstract models for concurrent systems in which any action can potentially be undone. Over the last few decades, different formalisms have been developed and their mathematical properties have been explored; however, none have been machine-checked within a proof assistant. This paper presents the first Beluga formalization of the Calculus of Communicating Systems with Keys and Proof labels (CCSK$^{\mathrm{P}}$), a reversible extension of CCS. Beyond the syntax and semantics of the calculus, the encoding covers state-of-the-art results regarding three relations over proof labels – namely, dependence, independence and connectivity – which offer new insights into the notions of causality and concurrency of events. As is often the case with formalizations, our encoding introduces adjustments to the informal proof and makes explicit details which were previously only sketched, some of which reveal to be less straightforward than initially assumed. We believe this work lays the foundations for future reversible concurrent calculi formalizations.

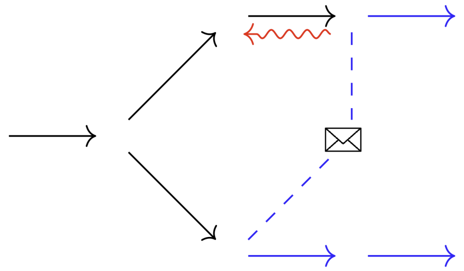
1 Introduction

Concurrency in computer science refers to the simultaneous execution of multiple operations or computations in a shared environment. It is a fundamental aspect of modern computing, with practical use in several domains such as operating systems, networking and distributed systems. Process calculi like CCS [13] and the π-calculus [14] are well-studied and established mathematical models for formally describing and reasoning about concurrent systems.

In recent years, reversing computations in concurrent systems has gained significant attention, with applications in fields like hardware, software and biochemistry [20]. Enriching concurrent systems with reversibility poses its own set of challenges: for instance, it requires providing some kind of history-preserving mechanism to take track of past actions. Additionally, undoing computation steps in a parallel setting is more complex than in a sequential system: as explained in Fig. 1, reversing a specific action performed by a single thread may require knowing, and eventually undoing, the actions of the other threads it has previously interacted with.

Reversible concurrent calculi address such challenges in various ways. For example, Reversible CCS (RCCS) [8] equips processes with a memory that records information about past computations; conversely, CCS with Keys (CCSK) [17] associates unique keys to each forward action. The latter has been recently upgraded to CCSK with Proof labels (CCSK$^{\mathrm{P}}$) [3], which features a proved transition system in the fashion of Degano and Priami [9]; proof labels enable the definition of dependence and independence for both forward and backward transitions. In this framework, the contributions brought by Aubert et al. [2] merit attention. The authors are the first to introduce separate axioms for the relations of dependence, independence and connectivity on proof labels: such relations are proved to be sound, interrelated, and linked to the broader notions of concurrency and causality of events. Additionally, the authors outline the difference between various kinds of bisimulations, such as the history preserving bisimulation for CCS.

To appear in EPTCS.



The diagram on the left illustrates the duplication of a thread, followed by a message exchange between the resulting branches and some further independent computation steps. To undo the computation step marked in red, which belongs to one of the two threads and precedes their interaction, it is necessary to first undo each of the steps marked in blue, including those of the other thread.

Figure 1: Example of reversal of computation steps in a concurrent setting.

Formal verification has become a cornerstone in the development of new systems, certifying the correctness of their syntax, semantics and behavioral properties in the most reliable way. In the case of concurrent calculi, there is a long tradition starting with the early mechanizations of CCS and the π-calculus in HOL [15][12]; we also recall the Rocq formalization of the π-calculus by Honsell et al. [11], which has been the baseline for numerous higher-order abstract syntax (HOAS) [16] mechanizations. When it comes to reversible concurrent calculi, however, the landscape looks rather different. Despite the availability of C# and Java implementations of CCSK [7][1], no machine-checked formalization of reversible concurrent calculi currently exists – at least to the best of our knowledge.

This paper presents the first formalization of CCSK$^{\text{P}}$ in Beluga [19]. Our encoding covers the core definitions of the system: its syntax, semantics, and the relations of dependence, independence and connectivity on proof labels. Additionally, this work formalizes the central results presented in Sections 3 and 4 of [2], including the complementarity of dependence and independence and the relationship between connectivity of transitions and proof labels. The proofs come along with a library of auxiliary lemmas regarding processes, keys and transitions. We also establish what is referred to as *informal adequacy* of the encoding [6], ensuring that our formalization faithfully mirrors the informal definitions and proofs.

Formalizations typically require small adjustments to fit the proof assistant's framework, while carefully addressing any of the details which are taken for granted in the informal proof. This encoding is no exception: the formalization process led to minor refinements in the definition of connectivity over proof labels and clarified the proof of one of the aforementioned results, which called for a different approach separating base and inductive cases. Beluga, as a proof assistant, is ideal for reasoning about deductive systems together with their meta-theory, as it naturally supports encodings of object-level binding constructs through higher-order abstract syntax and allows pairing terms with the contexts that give them meaning [19]. Although our HOAS encoding leverages Beluga's strengths and showcases its versatility, it also deals with limitations such as the lack of syntactic sugar for existentials or conjunctions; considerations on its adoption are further elaborated in the conclusions.

The paper is structured as follows. Section 2 provides an informal description of CCSK$^{\text{P}}$ and the results under our study. Section 3 presents the Beluga formalization of such notions and properties. Section 4 contains a technical overview of the formalization, a summary of our contributions and possible future work directions. Appendix A outlines the proof of the adequacy of the encoding. The full artifact is available at `https://github.com/CinRC/A-Beluga-Formalization-of-CCSKP`.

2 CCSK$^{\mathrm{P}}$

In this section we recall the main definitions and properties of CCSK$^{\mathrm{P}}$, as outlined in [2]. We assume familiarity with the basic notions of CCS. Definitions are hyperlinked to their encoding in the repository.

2.1 Syntax

As in the standard CCS, we assume the existence of an infinite set N of *names*, ranged over by a,b,c, with a bijection $\overline{\cdot}\colon \mathsf{N} \to \overline{\mathsf{N}}$ denoting the *complement* of a name; names and complementary names respectively denote input and output ports for processes. We define the set of *labels* L as $\mathsf{N} \cup \overline{\mathsf{N}} \cup \{\tau\}$, where τ denotes the interaction of concurrent processes. L is ranged over by α, while $\mathsf{L} \setminus \{\tau\}$ is ranged over by λ.

To introduce reversibility, CCSK extends the syntax of CCS with a denumerable set K of *keys*, ranged over by k,m,n. Labels are paired with keys to define *keyed labels*, which are elements of the cartesian product $\mathsf{L} \times \mathsf{K}$ and are represented as $a[k], b[m]$; the set of keyed labels is also denoted as L_{K}.

Processes are defined as in the ordinary CCS, with the addition of keyed prefixes and without operators for recursion or replication:

$$\begin{array}{rllll} X,Y ::= & \mathbf{0} & \text{(Inactive)} & \mid \alpha.X & \text{(Prefix)} \\ & \mid \alpha[k].X & \text{(Keyed prefix)} & \mid X+Y & \text{(Sum)} \\ & \mid X \mid Y & \text{(Parallel composition)} & \mid X \setminus a & \text{(Restriction)} \end{array}$$

The set of processes is denoted as $\mathbb{X}$. When preceded by a (keyed) prefix, the inactive process $\mathbf{0}$ is usually omitted; the binding power of the operators, from highest to lowest, is $\setminus a, \alpha[k], \alpha, \mid$ and $+$. In restrictions $X \setminus a$, the occurrences of the name a in X are said to be bound; all other occurrences of names and keys in processes are considered free. Processes that are α-equivalent, i.e., that differ only in the choice of their bound names, will be identified. Unlike [2], restrictions only bind names and not complementary names: this choice does not rule out any significant process (since $X \setminus a$ or $X \setminus \overline{a}$ have the same behaviour) and leads to a clearer correspondence between processes and their encoding.

The set of keys occurring in a process is denoted as $\mathsf{keys}(X)$. A process for which $\mathsf{keys}(X)$ is empty is said to be *standard*: in this case, we write that $\mathsf{std}(X)$ holds.

2.2 Semantics

The key feature of CCSK$^{\mathrm{P}}$ is the notion of *proof keyed labels*:

$$\theta ::= \upsilon\alpha[k] \quad \mid \quad \upsilon\langle |_{\mathrm{L}}\upsilon_1\lambda[k], |_{\mathrm{R}}\upsilon_2\overline{\lambda}[k]\rangle$$

where υ, υ_1 and υ_2 range over strings of symbols $\{|_{\mathrm{L}}, |_{\mathrm{R}}, +_{\mathrm{L}}, +_{\mathrm{R}}\}$. We denote the set of proof keyed labels as $\mathsf{L}^{\mathsf{P}}_{\mathsf{K}}$, and refer to its elements simply as "proof labels" for brevity. The following functions ℓ and $\not{k}$ map each proof label to its underlying label and key, respectively:

$$\ell(\upsilon\alpha[k]) = \alpha \qquad \ell(\upsilon\langle |_{\mathrm{L}}\upsilon_1\lambda[k], |_{\mathrm{R}}\upsilon_2\overline{\lambda}[k]\rangle) = \tau \qquad \not{k}(\upsilon\alpha[k]) = k \qquad \not{k}(\upsilon\langle |_{\mathrm{L}}\upsilon_1\lambda[k], |_{\mathrm{R}}\upsilon_2\overline{\lambda}[k]\rangle) = k$$

Semantics is given by the *labelled transition system* $(\mathbb{X}, \mathsf{L}^{\mathsf{P}}_{\mathsf{K}}, \xmapsto{\theta})$, where $\xmapsto{\theta}$ denotes the union of the forward and backward transitions displayed in Fig. 2. We will refer to the union of forward and backward transitions as *combined* transitions. Given a transition $X \xmapsto{\theta} Y$, the process X is said to be its *source*, while Y is said to be its *target*.

Prefix and Keyed Prefix

Forward

$$\mathsf{std}(X)\quad \frac{}{\alpha.X \xmapsto{\alpha[k]} \alpha[k].X}\ \text{pref}$$

$$\cancel{k}(\theta) \neq k\quad \frac{X \xmapsto{\theta} X'}{\alpha[k].X \xmapsto{\theta} \alpha[k].X'}\ \text{kpref}$$

Backward

$$\mathsf{std}(X)\quad \frac{}{\alpha[k].X \overset{\alpha[k]}{\leadsto} \alpha.X}\ \underline{\text{pref}}$$

$$\cancel{k}(\theta) \neq k\quad \frac{X' \overset{\theta}{\leadsto} X}{\alpha[k].X' \overset{\theta}{\leadsto} \alpha[k].X}\ \underline{\text{kpref}}$$

Sum

Forward

$$\mathsf{std}(Y)\quad \frac{X \xmapsto{\theta} X'}{X+Y \xmapsto{+_{\mathrm{L}}\theta} X'+Y}\ +_{\mathrm{L}}$$

Backward

$$\mathsf{std}(Y)\quad \frac{X' \overset{\theta}{\leadsto} X}{X'+Y \overset{+_{\mathrm{L}}\theta}{\leadsto} X+Y}\ \underline{+_{\mathrm{L}}}$$

Parallel Composition

Forward

$$\cancel{k}(\theta) \notin \mathsf{keys}(Y)\quad \frac{X \xmapsto{\theta} X'}{X \mid Y \xmapsto{|_{\mathrm{L}}\theta} X' \mid Y}\ |_{\mathrm{L}}$$

$$\frac{X \xmapsto{\nu_{\mathrm{L}}\lambda[k]} X' \qquad Y \xmapsto{\nu_{\mathrm{R}}\overline{\lambda}[k]} Y'}{X \mid Y \xmapsto{\langle |_{\mathrm{L}}\nu_{\mathrm{L}}\lambda[k], |_{\mathrm{R}}\nu_{\mathrm{R}}\overline{\lambda}[k]\rangle} X' \mid Y'}\ \text{syn}$$

Backward

$$\cancel{k}(\theta) \notin \mathsf{keys}(Y)\quad \frac{X' \overset{\theta}{\leadsto} X}{X' \mid Y \overset{|_{\mathrm{L}}\theta\theta}{\leadsto} X \mid Y}\ \underline{|_{\mathrm{L}}}$$

$$\frac{X' \overset{\nu_{\mathrm{L}}\lambda[k]}{\leadsto} X \qquad Y' \overset{\nu_{\mathrm{R}}\overline{\lambda}[k]}{\leadsto} Y}{X' \mid Y' \overset{\langle |_{\mathrm{L}}\nu_{\mathrm{L}}\lambda[k], |_{\mathrm{R}}\nu_{\mathrm{R}}\overline{\lambda}[k]\rangle}{\leadsto} X \mid Y}\ \underline{\text{syn}}$$

Restriction

Forward

$$\ell(\theta) \notin \{a, \overline{a}\}\quad \frac{X \xmapsto{\theta} X'}{X\backslash a \xmapsto{\theta} X'\backslash a}\ \text{nu}$$

Backward

$$\ell(\theta) \notin \{a, \overline{a}\}\quad \frac{X' \overset{\theta}{\leadsto} X}{X'\backslash a \overset{\theta}{\leadsto} X\backslash a}\ \underline{\text{nu}}$$

Figure 2: Forward and backward transition rules for CCSK$^{\mathrm{P}}$ (right rules for $\mid$ and $+$ omitted).

Example 1 Consider a webpage that allows the user to interact via two independent buttons: one to toggle between light and dark mode, and another to switch between two different languages. The initial state of the system can be modeled as the parallel composition $m \mid l$, where the labels m and l represent the actions to switch the visual mode and the language, respectively.

The action of changing the visual mode can be represented by the following forward transition: $m \mid l \xmapsto{|_L m[k]} m[k] \mid l$. The target process preserves the label m and pairs it with a fresh key k. The proof label $|_L m[k]$ not only stores the label m and key k used in the transition, but also indicates that the action occurred on the left-hand side of a parallel composition.

Suppose the user now wishes to revert to the previous visual mode: pressing the mode button again can be interpreted as undoing the previously executed action. This can be modeled by the following backward transition, which removes the key associated with the earlier forward step: $m[k] \mid l \overset{|_L m[k]}{\leadsto} m \mid l$.

Two transitions are said to be *composable* if they can be performed consecutively – that is, the target of the first transition is the source of the second transition. A *path* is a (potentially empty) sequence of composable transitions and can be denoted as $X \mapsto\!\!\!\!\rightarrow^{*} Y$, where X is the source of the first transition (also called the *source* of the path) and Y is the target of the last transition (also called the *target* of the path); in other words, $\mapsto\!\!\!\!\rightarrow^*$ is the reflexive and transitive closure of $\mapsto\!\!\!\!\rightarrow$. A process X is *reachable* if there exists a path whose target is X and whose source is a standard process. This process, which can be proved to be unique (cf. Lemma B.13 in [2]), is called the *origin* of X and is denoted as O_X.

Reachability allows to rule out faulty processes which are syntactically well-formed, but whose particular selection of keys is inconsistent. For instance, this arises when the same key denotes successive actions, as in the process $a[k].b[k]$, or when keys internally form a cycle, as in the deadlocked process $a[k].b[m] \mid \bar{b}[m].\bar{a}[k]$, where neither action can be undone because of the presence of its associated key in the other thread. From this point on, each process will be assumed to be reachable.

The *loop lemma* (cf. Lemma 3.8 in [2]) is an important result characterizing reversible labelled transition systems. It states that any transition $X \overset{\theta}{\mapsto\!\!\!\!\rightarrow} Y$ can be reversed, yielding a transition $Y \overset{\theta}{\mapsto\!\!\!\!\rightarrow} X$; moreover, the reversing operator is an involution (i.e., reversing a transition twice returns the original transition). The validity of the loop lemma follows directly from the symmetry of the LTS (Labelled Transition System) rules presented in Fig. 2.

Connectivity Relation

Action

$$\frac{}{\alpha[k] \curlyvee \theta}\ \mathrm{A}^1 \qquad \frac{}{\theta \curlyvee \alpha[k]}\ \mathrm{A}^2$$

Choice

$$\frac{\theta_1 \curlyvee \theta_2}{+_{\mathrm{d}}\theta_1 \curlyvee +_{\mathrm{d}}\theta_2}\ \mathrm{C}^1_{\mathrm{d}} \qquad \frac{}{+_{\mathrm{d}}\theta_1 \curlyvee +_{\bar{\mathrm{d}}}\theta_2}\ \mathrm{C}^2_{\mathrm{d}}$$

Parallel

$$\frac{\theta_1 \curlyvee \theta_2}{|_{\mathrm{d}}\theta_1 \curlyvee |_{\mathrm{d}}\theta_2}\ \mathrm{P}^1_{\mathrm{d}} \qquad \frac{}{|_{\mathrm{d}}\theta_1 \curlyvee |_{\bar{\mathrm{d}}}\theta_2}\ \mathrm{P}^2_{\mathrm{d}}$$

Synchronization

$$\frac{\theta \curlyvee \theta_{\mathrm{d}}}{|_{\mathrm{d}}\theta \curlyvee \langle |_{\mathrm{L}}\theta_{\mathrm{L}}, |_{\mathrm{R}}\theta_{\mathrm{R}} \rangle}\ \mathrm{S}^1_{\mathrm{d}} \qquad \frac{\theta_{\mathrm{d}} \curlyvee \theta}{\langle |_{\mathrm{L}}\theta_{\mathrm{L}}, |_{\mathrm{R}}\theta_{\mathrm{R}} \rangle \curlyvee |_{\mathrm{d}}\theta}\ \mathrm{S}^2_{\mathrm{d}}$$

$$\frac{\theta_1 \curlyvee \theta'_1 \qquad \theta_2 \curlyvee \theta'_2}{\langle |_{\mathrm{L}}\theta_1, |_{\mathrm{R}}\theta_2 \rangle \curlyvee \langle |_{\mathrm{L}}\theta'_1, |_{\mathrm{R}}\theta'_2 \rangle}\ \mathrm{S}^3$$

Dependence Relation

Action

$$\frac{}{\alpha[k] \times \theta}\ \mathrm{A}^1 \qquad \frac{}{\theta \times \alpha[k]}\ \mathrm{A}^2$$

Choice

$$\frac{\theta \times \theta'}{+_{\mathrm{d}}\theta \times +_{\mathrm{d}}\theta'}\ \mathrm{C}^1_{\mathrm{d}} \qquad \frac{}{+_{\mathrm{d}}\theta \times +_{\bar{\mathrm{d}}}\theta'}\ \mathrm{C}^2_{\mathrm{d}}$$

Parallel

$$\frac{\theta \times \theta'}{|_{\mathrm{d}}\theta \times\ |_{\mathrm{d}}\theta'}\ \mathrm{P}^1_{\mathrm{d}} \qquad \frac{\mathscr{k}(\theta) = \mathscr{k}(\theta')}{|_{\mathrm{d}}\theta \times\ |_{\bar{\mathrm{d}}}\theta'}\ \mathrm{P}^2_{\mathrm{d}}$$

Synchronization

$$\frac{\theta \times \theta_{\mathrm{d}}}{|_{\mathrm{d}}\theta \times \langle |_{\mathrm{L}}\theta_{\mathrm{L}}, |_{\mathrm{R}}\theta_{\mathrm{R}} \rangle}\ \mathrm{S}^1_{\mathrm{d}} \qquad \frac{\theta_{\mathrm{d}} \times \theta}{\langle |_{\mathrm{L}}\theta_{\mathrm{L}}, |_{\mathrm{R}}\theta_{\mathrm{R}} \rangle \times\ |_{\mathrm{d}}\theta}\ \mathrm{S}^2_{\mathrm{d}}$$

$$\frac{\theta_i \times \theta'_i \qquad \theta_j \curlyvee \theta'_j \qquad i, j \in \{1,2\}, i \neq j}{\langle |_{\mathrm{L}}\theta_1, |_{\mathrm{R}}\theta_2 \rangle \times \langle |_{\mathrm{L}}\theta'_1, |_{\mathrm{R}}\theta'_2 \rangle}\ \mathrm{S}^3$$

Independence Relation

Action

(empty)

Choice

$$\frac{\theta \iota \theta'}{+_{\mathrm{d}}\theta\ \iota +_{\mathrm{d}}\theta'}\ \mathrm{C}^1_{\mathrm{d}}$$

Parallel

$$\frac{\theta \iota \theta'}{|_{\mathrm{d}}\theta\ \iota\ |_{\mathrm{d}}\theta'}\ \mathrm{P}^1_{\mathrm{d}} \qquad \frac{\mathscr{k}(\theta) \neq \mathscr{k}(\theta')}{|_{\mathrm{d}}\theta\ \iota\ |_{\bar{\mathrm{d}}}\theta'}\ \mathrm{P}^2_{\mathrm{d}}$$

Synchronization

$$\frac{\theta \iota \theta_{\mathrm{d}}}{|_{\mathrm{d}}\theta\ \iota \langle |_{\mathrm{L}}\theta_{\mathrm{L}}, |_{\mathrm{R}}\theta_{\mathrm{R}} \rangle}\ \mathrm{S}^1_{\mathrm{d}} \qquad \frac{\theta_{\mathrm{d}} \iota \theta}{\langle |_{\mathrm{L}}\theta_{\mathrm{L}}, |_{\mathrm{R}}\theta_{\mathrm{R}} \rangle\ \iota\ |_{\mathrm{d}}\theta}\ \mathrm{S}^2_{\mathrm{d}}$$

$$\frac{\theta_1 \iota \theta'_1 \qquad \theta_2 \iota \theta'_2}{\langle |_{\mathrm{L}}\theta_1, |_{\mathrm{R}}\theta_2 \rangle\ \iota \langle |_{\mathrm{L}}\theta'_1, |_{\mathrm{R}}\theta'_2 \rangle}\ \mathrm{S}^3$$

Figure 3: Causality relations on proof labels.

Finally, the binary relations of *connectivity*, *dependence* and *independence* on proof labels, respectively denoted as $\curlyvee, \times$ and ι, are defined by the rules displayed in Fig. 3, where the label d ranges over $\{\mathrm{L},\mathrm{R}\}$ and $\overline{\mathrm{d}}$ denotes the opposite of d (i.e., $\overline{\mathrm{L}} = \mathrm{R}$ and $\overline{\mathrm{R}} = \mathrm{L}$). Such relations will be referred to as *causality relations* for brevity. Compared to [2], the rule A^2 for connectivity and dependence has been slightly modified, ensuring that each relation is symmetric and simplifying their encoding. This comes at the cost of losing uniqueness in derivations of judgements such as $\theta_1 \curlyvee \theta_2$; however, this property has been shown not to be required for the purposes of our development.

Example 2 The process $m \mid l$, introduced in Example 1 to model a webpage, can perform a forward transition labelled by $|_L m[k_1]$, representing the toggling of the visual mode. It can also perform a transition $m \mid l \xmapsto{|_R l[k_2]} m \mid l[k_2]$, denoting the change of the language of the webpage. The two transitions are independent, as the order in which they are executed does not affect the resulting state. This is reflected in the independence of the two proof labels $|_L m[k_1]$ and $|_R l[k_2]$, which follows from the $\mathrm{P}^2_{\mathrm{L}}$ rule in Fig. 3.

Conversely, consider the process $a.b \mid \bar{b}$. It can perform a forward transition labelled by $|_L a[k]$, followed by another forward transition labelled by $|_L b[n]$. However, these transitions cannot be performed in reverse order, since the input action along b is only enabled after the input action along a has occurred; the two transitions are thus causally related. This is reflected in the dependence of the two proof labels $|_L a[k]$ and $|_L b[n]$, which follows from the $\mathrm{P}^1_{\mathrm{L}}$ and A^1 rules in Fig. 3.

2.3 Properties of causality relations

We now turn to the theorems and lemmas object of our study. Their original proof can be found in [2]. The following theorem specifies the relationship between connectivity of transitions and connectivity of proof labels:

Theorem 2.1 (cf. Proposition 4.4 in [2])

(i) If $t_1 : X_1 \xmapsto{\theta_1} X_1'$ *and* $t_2 : X_2 \xmapsto{\theta_2} X_2'$ *are connected, then* $\theta_1 \curlyvee \theta_2$. ↗

(ii) If $\theta_1 \curlyvee \theta_2$*, then there exist* $t_1 : X_1 \xmapsto{\theta_1} X_1'$ *and* $t_2 : X_2 \xmapsto{\theta_2} X_2'$ *such that* t_1 *and* t_2 *are connected.* ↗

The proof of Theorem 2.1(i) relies on the fact that $O_{X_1} = O_{X_2}$ and proceeds by induction over such origin process: recall that each process is assumed to be reachable and, therefore, has an origin. The equality of O_{X_1} and O_{X_2} follows from the two lemmas:

Lemma 2.2 *For all reachable processes* X *and* Y*, there exists a path* $X \mapsto^* Y$ *iff* $O_X = O_Y$.

Lemma 2.3 *If* $t_1 : X_1 \xmapsto{\theta_1} X_1'$ *and* $t_2 : X_2 \xmapsto{\theta_2} X_2'$ *are connected, then* $O_{X_1} = O_{X_2}$.

Conversely, the proof of Theorem 2.1(ii) proceeds by structural induction over the given hypothesis $\theta_1 \curlyvee \theta_2$ and relies on the following:

Definition 2.4 (Realisation) *A process* X realises the proof label θ *if there exist* X_1 *and* X_2 *such that* $X \mapsto^* X_1 \xmapsto{\theta} X_2$. ↗

Lemma 2.5 *For every proof label* θ*, there exists a process that realises it, and we denote it* $r(\theta)$. ↗

Next, the following theorem states the complementarity of the dependence and independence relations:

Theorem 2.6 (cf. Theorem 4.6 in [2])

For all θ_1, θ_2,

(i) If $\theta_1 \iota \theta_2$ *then* $\theta_1 \curlyvee \theta_2$. ↗

(ii) If $\theta_1 \times \theta_2$ *then* $\theta_1 \curlyvee \theta_2$. ↗

(iii) If $\theta_1 \curlyvee \theta_2$ *then either* $\theta_1 \iota \theta_2$ *or* $\theta_1 \times \theta_2$*, but not both.* ↗

This theorem is proved by induction over the structure of the given binary relation.

3 Beluga Formalization

In this section we outline the key points of the Beluga formalization of the notions presented in Section 2. Definitions and proofs omitted for brevity are hyperlinked to their encoding in the repository.

3.1 Syntax

Beluga is structured in two layers: the LF (Logical Frameworks [10]) level, which is used to specify the formal system under study, and the computation level, which supports programming with LF data [19]. To encode the syntax of our system, only the former level is deployed. Names, keys, labels and processes are encoded using the LF types displayed in Fig. 4.

```
LF names: type =;              LF proc: type =
LF keys: type =                  | null: proc                               % 0
  | z: keys                      | pref: labels → proc → proc               % A.X
  | s: keys → keys;              | kpref: labels → keys → proc → proc       % A[k].X
LF labels: type =                | sum: proc → proc → proc                  % X+Y
  | inp: names → labels          | par: proc → proc → proc                  % X|Y
  | out: names → labels          | nu: (names → proc) → proc;               % X\a
  | tau: labels;
```

Figure 4: Encoding of the syntax of CCSK^{P}.

Since names in CCSK^{P} are an infinite set without any additional assumption, they are represented by a type `names` without constructors; as explained in [5], this type will be dynamically inhabited by variables introduced through contexts. This is enabled by the following line of code:

```
schema ctx = names;
```

This line declares contexts made of a finite collection of distinct variables of type `names`, identified via the keyword `ctx`. Thanks to this setup, we can work with *contextual processes* of the form `[g ⊢ X]`, i.e., processes `X` whose free names are drawn from the context g. Contextual objects live in the computation level.

Keys are by assumption denumerable, and the LTS rules for keyed prefixes require equality of keys to be decidable. Both conditions are satisfied by encoding keys explicitly as natural numbers.[1] An alternative approach would be to rely on contexts, as is done for names: however, this would require managing mixed contexts of names and keys, and having a more complex encoding of transitions and paths.

Restrictions $X \setminus a$ are represented by terms of the form `(nu \a.(X a))`, where `\x.(f x)` is Beluga's notation for functions `f` mapping `x` to `f(x)`: following the higher-order abstract syntax (HOAS) paradigm, the bound name a is represented as the implicit argument of a meta-language function `\a.(X a)` from `names` to `proc`. In this way, we leverage the meta-language implementation of binders to achieve α-renaming and capture-avoiding substitutions for free.

An important but often overlooked aspect of formalizations is the *adequacy* of the encoding: the encoding must constitute a faithful representation of the original system into study [6]. Adequacy is generally established by proving the existence of a compositional bijection between the mathematical model and its formalized counterpart. The discussion of the adequacy of our encoding is deferred to Appendix A.

[1] Note that properties such as decidability of equality must be stated and proved manually, as Beluga does not provide a built-in library of properties of natural numbers.

3.2 Semantics

Proof labels are encoded by the type pr_lab in Fig. 5. Rather than directly modeling the informal definition of proof labels, by defining strings over the symbols $\{|_{\mathrm{L}}, |_{\mathrm{R}}, +_{\mathrm{L}}, +_{\mathrm{R}}\}$ as lists, we are introducing four constructors (pr_suml, pr_sumr, etc.) that build proof labels incrementally by appending one symbol at a time. This provides a stronger induction principle and streamlines the encoding of LTS rules and subsequent proofs.

```
LF pr_lab: type =                           LF lab: pr_lab → labels → type =
  | pr_base: labels → keys → pr_lab           | lab_base: lab (pr_base A K) A
  | pr_suml: pr_lab → pr_lab                  | lab_suml: lab T A → lab (pr_suml T) A
  | pr_sumr: pr_lab → pr_lab                  | lab_sumr: lab T A → lab (pr_sumr T) A
  | pr_parl: pr_lab → pr_lab                  | lab_parl: lab T A → lab (pr_parl T) A
  | pr_parr: pr_lab → pr_lab                  | lab_parr: lab T A → lab (pr_parr T) A
  | pr_sync: pr_lab → pr_lab → pr_lab;        | lab_sync: lab (pr_sync T1 T2) tau;
LF valid: pr_lab → type =
  | v_base: valid (pr_base A K)
  | v_suml: valid T → valid (pr_suml T)
  | v_sumr: valid T → valid (pr_sumr T)
  | v_parl: valid T → valid (pr_parl T)
  | v_parr: valid T → valid (pr_parr T)
  | v_synl: valid T1 → valid T2 → lab T1 (inp A) → key T1 K
        → lab T2 (out A) → key T2 K → valid (pr_sync T1 T2)
  | v_synr: valid T1 → valid T2 → lab T1 (out A) → key T1 K
        → lab T2 (inp A) → key T2 K → valid (pr_sync T1 T2);
LF fstep: proc → pr_lab → proc → type =
  | fs_pref: std X → fstep (pref A X) (pr_base A K) (kpref A K X)
  | fs_kpref: fstep X T X' → key T M → neq K M
          → fstep (kpref A K X) T (kpref A K X')
  | fs_suml: fstep X T X' → std Y → fstep (sum X Y) (pr_suml T) (sum X' Y)
  | fs_sumr: fstep Y T Y' → std X → fstep (sum X Y) (pr_sumr T) (sum X Y')
  | fs_parl: fstep X T X' → key T K → notin K Y
         → fstep (par X Y) (pr_parl T) (par X' Y)
  | fs_parr: fstep Y T Y' → key T K → notin K X
         → fstep (par X Y) (pr_parr T) (par X Y')
  | fs_synl: fstep X T1 X' → lab T1 (inp L) → key T1 K
         → fstep Y T2 Y' → lab T2 (out L) → key T2 K
         → fstep (par X Y) (pr_sync T1 T2) (par X' Y')
  | fs_synr: fstep X T1 X' → lab T1 (out L) → key T1 K
         → fstep Y T2 Y' → lab T2 (inp L) → key T2 K
         → fstep (par X Y) (pr_sync T1 T2) (par X' Y')
  | fs_nu: ({a:names} fstep (X a) T (X' a)) → fstep (nu X) T (nu X');
```

Figure 5: Encoding of the semantics of CCSK$^{\mathrm{P}}$.

In Beluga, predicates are encoded as type families, i.e., types parametrized by arguments: a predicate $P(x_1,\dots,x_n)$ holds iff the corresponding type (P $\mathtt{x_1}$ … $\mathtt{x_n}$) is not empty. Type families are also used to encode functions, identified with their graph, as in the case of the functions ℓ and $\mathscr{k}$ returning the label and key of a proof label: the former is encoded by the type family lab in Fig. 5, while the latter is encoded by the type family key, here omitted for brevity. For example, given a proof label θ and a label α, represented as T and A in the encoding, the type lab T A is inhabited iff $\ell(\theta) = \alpha$.

Our encoding of proof labels pays the price of being over-expressive: the constructor pr_sync accepts any two proof labels regardless of their key or label, generating terms that fall outside the original definition. For example, the term (pr_sync (pr_base A K) (pr_base B M)) has type pr_lab for any labels A, B and keys K, M, while its counterpart $\langle |_{\mathrm{L}}\alpha[k], |_{\mathrm{R}}\beta[m]\rangle$ is well-defined only if $\beta = \overline{\alpha}$ and $k = m$. While this is harmless in most of our development, since such spurious terms do not label any actual transition, it becomes an issue when proving theorems universally quantified on proof labels, such as Lemma 2.5. To address this problem, we introduce an additional predicate valid, displayed in Fig. 5, which filters out the spurious terms. In Appendix A we prove the existence of a bijection between proof labels and the set of elements T of type pr_lab for which valid T holds. From this point forward, we will refer to such terms as *valid* proof labels.

Forward and backward LTS rules are defined through the type families fstep and bstep in Fig. 5 (with the latter omitted here for brevity). These rules rely on the additional type families std, notin and neq, hyperlinked to their formalization in the repository, which respectively hold when a process is standard, when a key does not occur in a process, and when two keys are not equal. The parameters X and X' in the fs_nu rule are functions from names to proc, whose arguments represent the binders of the restrictions. The universal quantification {a:names} is used to abstract over the particular choice of the binder; moreover, a or $\bar{a}$ does not occur in the proof label T, since such parameter does not depend on a within the body of the universal quantification.

Combined transitions, paths, reachable processes and connected transitions are defined as follows:

```
LF step: proc → pr_lab → proc → type =
  | fw: fstep X T X' → step X T X'
  | bw: bstep X' T X → step X' T X;
LF step*: proc → proc → type =
  | id_s*: step* X X
  | st_s*: step X T Y → step* X Y
  | tr_s*: step* X Y → step* Y Z → step* X Z;
LF reachable: proc → type =
  | rch: std X → step* X Y → reachable Y;
LF conn_tr: step X T1 X' → step Y T2 Y' → type =
  | ct: {S1:step X T1 X'}{S2: step Y T2 Y'} step* X Y' → conn_tr S1 S2;
```

Paths, or multi-step transitions, can be encoded equivalently using only two constructors; in this development, the more verbose version has been adopted as it simplified the proof search.

Finally, the relations of connectivity, dependence and independence are encoded through three type families conn, dep and indep. The former is displayed in Fig. 6. While some of the rules in Fig. 3 are grouped together, by using the label d in the place of L and R, the encoding requires each rule to be stated separately, with its own constructor.

3.2.1 Basic properties of keys, proof labels and transitions

Before diving into the theorems related to the causality relations, our encoding requires a small library of properties of keys, proof labels and transitions: these include the *decidability of equality of keys*, the *functionality of* ℓ *and* ꝁ, the fact that *standard processes have no keys*, or the *loop lemma*. To provide an overview of how proofs are carried out in Beluga, we will walk through the proof of the following result: "for all proof labels θ, there exists a label α such that $\ell(\theta) = \alpha$". Its code is displayed in Fig. 7:

```
LF conn: pr_lab → pr_lab → type =
  | c_a1: conn (pr_base A K) T
  | c_a2: conn T (pr_base A K)
  | c_c1l: conn T1 T2 → conn (pr_suml T1) (pr_suml T2)
  | c_c1r: conn T1 T2 → conn (pr_sumr T1) (pr_sumr T2)
  | c_c2l: conn (pr_suml T1) (pr_sumr T2)
  | c_c2r: conn (pr_sumr T1) (pr_suml T2)
  | c_p1l: conn T1 T2 → conn (pr_parl T1) (pr_parl T2)
  | c_p1r: conn T1 T2 → conn (pr_parr T1) (pr_parr T2)
  | c_p2l: conn (pr_parl T1) (pr_parr T2)
  | c_p2r: conn (pr_parr T1) (pr_parl T2)
  | c_s1l: conn T TL → conn (pr_parl T) (pr_sync TL TR)
  | c_s1r: conn T TR → conn (pr_parr T) (pr_sync TL TR)
  | c_s2l: conn TL T → conn (pr_sync TL TR) (pr_parl T)
  | c_s2r: conn TR T → conn (pr_sync TL TR) (pr_parr T)
  | c_s3: conn T1 T1’ → conn T2 T2’ → conn (pr_sync T1 T2) (pr_sync T1’ T2’);
```

Figure 6: Encoding of the connectivity relation on proof labels.

```
LF ex_lab: pr_lab → type =
  | ex_l: lab T A → ex_lab T;
rec existence_of_lab: (g:ctx) {T:[g ⊢ pr_lab]} [g ⊢ ex_lab T] =
/ total t (existence_of_lab _ t) /
mlam T ⇒ case [_ ⊢ T] of
  | [g ⊢ pr_base _ _] ⇒ [g ⊢ ex_l lab_base]
  | [g ⊢ pr_suml T’] ⇒ let [g ⊢ ex_l L] = existence_of_lab [g ⊢ T’] in
    [g ⊢ ex_l (lab_suml L)]
  | [g ⊢ pr_sumr T’] ⇒ let [g ⊢ ex_l L] = existence_of_lab [g ⊢ T’] in
    [g ⊢ ex_l (lab_sumr L)]
  | [g ⊢ pr_parl T’] ⇒ let [g ⊢ ex_l L] = existence_of_lab [g ⊢ T’] in
    [g ⊢ ex_l (lab_parl L)]
  | [g ⊢ pr_parr T’] ⇒ let [g ⊢ ex_l L] = existence_of_lab [g ⊢ T’] in
    [g ⊢ ex_l (lab_parr L)]
  | [g ⊢ pr_sync _ _] ⇒ [g ⊢ ex_l lab_sync];
```

Figure 7: Proof of the existence of a label in a proof label.

The first two lines of code in Fig. 7 introduce a type family `ex_lab`, which captures the conclusions of the lemma to be proved: the type `ex_lab T` is inhabited whenever there exists a label `A` for which `lab T A` holds. Defining such additional type families is the standard workaround to the lack of syntactic sugar for existentials and conjunctions in Beluga.

Thanks to the Curry-Howard isomorphism, proofs by induction are encoded through recursive functions. In Beluga, these are computation-level entities introduced by the keyword `rec`. The function `existence_of_lab` takes as input a context g of schema `ctx` and a contextual object `T` of type `pr_lab` and returns an object of type `ex_lab T`. The second line of the proof asserts that the built function is total. This condition is verified by Beluga’s totality checker and guarantees that the recursive function constitutes a valid proof.

The proof itself begins by introducing the argument `T` through the keyword `mlam`; the other argument, the context g, is implicit due to the use of round brackets in the function declaration. The proof proceeds by pattern matching on the object `T`, which by the Curry-Howard isomorphism corresponds to case analysis on the structure of `T` in the informal proof. Underscores are used to omit parameters that Beluga can infer automatically. The `pr_base` and `pr_sync` cases are handled immediately, since we can already provide the required object of type `ex_lab T`; for the remaining four cases, the proof proceeds by recursively applying the function `existence_of_lab` to a subterm `T'` of `T`, and then using the result to build the desired object. Recursive calls on structurally smaller objects correspond to applications of the inductive hypothesis in the informal proof.

We conclude this subsection by addressing the *symmetry* and *irreflexivity* of the causality relations. We report the signature of the function which proves that connectivity is symmetric:

```
rec symmetric_conn: (g:ctx) [g ⊢ conn T1 T2] → [g ⊢ conn T2 T1] = ...
```

These lemmas are proved by straightforward inductions on the structure of the given predicate.

3.3 Properties of causality relations

Although the theorems in Section 2.3 are presented in a different order, here we start with the encoding of Theorem 2.6, as it is straightforward and mirrors the structure of the informal proof.

3.3.1 Encoding of Theorem 2.6

The three statements of the theorem are addressed by four recursive functions: this is because Theorem 2.6(iii) actually consists of two separate assertions, which here we prove separately. Moreover, the disjunction in the conclusions requires defining an additional type family `dep_or_indep`. Fig. 8 displays the proof of the final assertion: “two proof labels cannot be both dependent and independent”. We also present the signatures of the other recursive functions below.

```
rec indep_impl_conn: (g:ctx) [g ⊢ indep T1 T2] → [g ⊢ conn T1 T2] = ...
rec dep_impl_conn: (g:ctx) [g ⊢ dep T1 T2] → [g ⊢ conn T1 T2] = ...
LF dep_or_indep: pr_lab → pr_lab → type =
  | or_dep: dep T1 T2 → dep_or_indep T1 T2
  | or_ind: indep T1 T2 → dep_or_indep T1 T2;
rec conn_impl_dep_or_indep: (g:ctx) [g ⊢ conn T1 T2] → [g ⊢ dep_or_indep T1 T2] = ...
```

The proof in Fig. 8 is an example of proof by contradiction: given two objects of type `dep T1 T2` and `indep T1 T2`, the function `impossible_dep_and_indep` aims to derive an object of the empty type `false`, thereby establishing a contradiction. After introducing the arguments `d` and `i`, the proof proceeds by pattern matching on `d`; depending on the case, the contradiction is reached in one of three distinct ways.

In case `d` is built, e.g., through the constructor `d_a1` (corresponding to the case A^1: $\alpha[k] \times \theta$ in the informal proof), it is immediately clear that an object `i` of type `indep T1 T2` (i.e., $\alpha[k]\ \iota\ \theta$) does not exist: this contradiction is exhibited through the keyword `impossible`. In other subcases, such as when `d` is built via `d_c1l` (corresponding to C^1_L: $+_\mathrm{L}\theta \times +_\mathrm{L}\theta'$, given $\theta \times \theta'$), the contradiction is obtained by recursively invoking `impossible_dep_and_indep` on smaller arguments. Finally, in the `d_p2l` subcase ($|_\mathrm{L}\ \theta \times |_\mathrm{R}\ \theta'$, under the assumption $\mathscr{k}(\theta) = \mathscr{k}(\theta')$), we first examine the structure of `i` and find that it must have been constructed using `i_p2l`. This gives us an object `N` witnessing the inequality $\mathscr{k}(\theta) \neq \mathscr{k}(\theta')$, which clearly contradicts our assumption; however, to complete the proof in Beluga, it is first necessary to apply the auxiliary function `uniqueness_of_key` for some variable unification, yielding $\mathscr{k}(\theta) \neq \mathscr{k}(\theta)$, followed by the function `irreflexive_neq`, which states the irreflexivity of the inequality of keys.

```
rec impossible_dep_and_indep: (g:ctx) [g ⊢ dep T1 T2] → [g ⊢ indep T1 T2]
  → [g ⊢ false] =
/ total d (impossible_dep_and_indep _ _ _ d _) /
fn d,i ⇒ case d of
  | [g ⊢ d_a1] ⇒ impossible i
  | [g ⊢ d_a2] ⇒ impossible i
  | [g ⊢ d_c1l D] ⇒ let [g ⊢ i_c1l I] = i in impossible_dep_and_indep [g ⊢ D] [g ⊢ I]
  | [g ⊢ d_c1r D] ⇒ let [g ⊢ i_c1r I] = i in impossible_dep_and_indep [g ⊢ D] [g ⊢ I]
  | [g ⊢ d_c2l] ⇒ impossible i
  | [g ⊢ d_c2r] ⇒ impossible i
  | [g ⊢ d_p1l D] ⇒ let [g ⊢ i_p1l I] = i in impossible_dep_and_indep [g ⊢ D] [g ⊢ I]
  | [g ⊢ d_p1r D] ⇒ let [g ⊢ i_p1r I] = i in impossible_dep_and_indep [g ⊢ D] [g ⊢ I]
  | [g ⊢ d_p2l H1 H2] ⇒ let [g ⊢ i_p2l H1' H2' N] = i in
    let [g ⊢ refk] = uniqueness_of_key [g ⊢ H1] [g ⊢ H1'] in
    let [g ⊢ refk] = uniqueness_of_key [g ⊢ H2] [g ⊢ H2'] in irreflexive_neq [g ⊢ N]
  | [g ⊢ d_p2r H1 H2] ⇒ let [g ⊢ i_p2r H1' H2' N] = i in
    let [g ⊢ refk] = uniqueness_of_key [g ⊢ H1] [g ⊢ H1'] in
    let [g ⊢ refk] = uniqueness_of_key [g ⊢ H2] [g ⊢ H2'] in irreflexive_neq [g ⊢ N]
  | [g ⊢ d_s1l D] ⇒ let [g ⊢ i_s1l I] = i in impossible_dep_and_indep [g ⊢ D] [g ⊢ I]
  | [g ⊢ d_s1r D] ⇒ let [g ⊢ i_s1r I] = i in impossible_dep_and_indep [g ⊢ D] [g ⊢ I]
  | [g ⊢ d_s2l D] ⇒ let [g ⊢ i_s2l I] = i in impossible_dep_and_indep [g ⊢ D] [g ⊢ I]
  | [g ⊢ d_s2r D] ⇒ let [g ⊢ i_s2r I] = i in impossible_dep_and_indep [g ⊢ D] [g ⊢ I]
  | [g ⊢ d_s3l D1 _] ⇒ let [g ⊢ i_s3 I1 _] = i in
    impossible_dep_and_indep [g ⊢ D1] [g ⊢ I1]
  | [g ⊢ d_s3r _ D2] ⇒ let [g ⊢ i_s3 _ I2] = i in
    impossible_dep_and_indep [g ⊢ D2] [g ⊢ I2];
```

Figure 8: Proof of the statement "two proof labels cannot be both dependent and independent".

3.3.2 Encoding of Theorem 2.1

Recall that, throughout our development, each process is assumed to be reachable. Although this hypothesis is not explicitly stated in theorems and lemmas, it is in fact essential for proving Theorem 2.1 and some of its auxiliary lemmas. For this reason, before outlining its encoding, we refine its statement, making the reachability assumption explicit:

Theorem 2.1 (Refined)

(i) If $t_1 : X_1 \xmapsto{\theta_1} X_1'$ and $t_2 : X_2 \xmapsto{\theta_2} X_2'$ are connected and X_1 is reachable,[2] *then $\theta_1 \curlyvee \theta_2$.*

(ii) If $\theta_1 \curlyvee \theta_2$, then there exist $t_1 : X_1 \xmapsto{\theta_1} X_1'$ and $t_2 : X_2 \xmapsto{\theta_2} X_2'$, with X_1 reachable, such that t_1 and t_2 are connected.

Since the two statements are encoded by two distinct functions, we discuss them separately. The proof of Theorem 2.1(i) is given by the following function conn_rel_one:

```
rec conn_rel_one: (g:ctx) {S1:[g ⊢ step X1 T1 X1']} {S2:[g ⊢ step X2 T2 X2']}
  [g ⊢ reachable X1] → [g ⊢ conn_tr S1 S2] → [g ⊢ conn T1 T2] = ...
```

[2] The reachability of the only X_1 is enough to deduce the reachability of any other process in the statement, given the existence of a path from X_1 to such processes.

The function takes as inputs two transitions S1 and S2, whose typing judgments introduce the names of each involved parameter, such as the process X1; these are followed by the further assumptions of reachability of X1 and connectivity of S1 and S2. The function returns a derivation of the connectivity of the proof labels T1 and T2.

Although our encoding may appear different – and somewhat longer – than the proof presented in [2], it is, in essence, faithful to the same underlying structure. The original proof leverages Lemma 2.3 to establish the equality of the processes O_{X_1} and O_{X_2}, the origins of the sources of the connected transitions $t_1 : X_1 \overset{\theta_1}{\mapsto\!\!\!\twoheadrightarrow} X_1'$ and $t_2 : X_2 \overset{\theta_2}{\mapsto\!\!\!\twoheadrightarrow} X_2'$. It proceeds by induction on O_{X_1}, observing that its structure determines that of the processes and transitions in the same environment (e.g., if the outermost operator of O_{X_1} is a sum, the same applies to X_1). The proof then concludes either directly or by applying the induction hypothesis to transitions involving specific subprocesses.

Below, we outline the changes and technical considerations brought by our encoding of this argument:

- The formalized proof proceeds by pattern matching on an object D of type `std OX1`, rather than directly on the process OX1. This is essentially equivalent, since the type family `std proc`, which asserts that a process is standard, is itself defined by pattern matching on the underlying process.
- It is not necessary to encode Lemma 2.3. The reachability of X_1, together with the existence of a path from X_1 to X_2, provides us a path between O_{X_1} and X_2; this path is enough to determine the structure of X_2, known the structure of O_{X_1}.
- The proof requires analyzing the structure of the given transitions S1 and S2. Since combined transitions are either forward or backward, and each have their own constructors, this results in four levels of nested pattern matching. While most of the subcases can be unified in the informal proof, Beluga requires them to be treated separately: this is the primary reason for the proof's length. To improve efficiency, certain assertions have been moved earlier in the proof tree compared to their position in the informal version.
- The informal proof takes for granted structural properties such as: "given a path whose source is a sum process, the target is also a sum process", or "given a path between two sum processes, there exists a path between their left addends". In the encoding, these results must be explicitly stated and proved, resulting in 16 additional lemmas. Some of these require classical techniques such as mutual recursion or strengthening of contextual judgments, which are described in [5]. We report the signatures of two of these functions:

```
% Type family encoding sum processes
LF is_sum: proc → type =
  | sm: is_sum (sum X Y);
% A path starting from a sum process ends in a sum process
rec step*_from_sum: (g:ctx) [g ⊢ step* (sum X Y) Z] → [g ⊢ is_sum Z] = ...
% Given a path between sum processes, there is a path between their left addends
rec step*_betw_sums_left: (g:ctx) [g ⊢ step* (sum X1 X2) (sum Y1 Y2)]
  → [g ⊢ step* X1 Y1] = ...
```

The formalization of Theorem 2.1(ii) requires encoding Lemma 2.5, which states that every proof label θ is realised by some process $r(\theta)$ – that is, there exist processes X_1, X_2 and $r(\theta)$ such that $r(\theta) \mapsto\!\!\!\twoheadrightarrow^* X_1 \overset{\theta}{\mapsto\!\!\!\twoheadrightarrow} X_2$. The original proof in [2], however, goes further: it builds a process $r(\theta)$ which is standard and directly performs a single forward transition $r(\theta) \xmapsto{\theta} X_2$ (in other words, $r(\theta)$ and X_1 coincide). Our encoding reflects this stronger formulation by specializing the original Definition 2.4 with the following type family `realised`:

```
LF realised: pr_lab → type =
  | rl: std X → fstep X T X’ → realised T;
```

For any proof label `T`, `realised T` is non empty iff `std X` and `fstep X T X’` hold for some `X` and `X’`. The following recursive function `pr_lab_is_realised` encodes the proof of Lemma 2.5:

```
rec pr_lab_is_realised: (g:ctx) [g ⊢ valid T] → [g ⊢ realised T] = ...
```

The proof is a straightforward induction on the structure of the assumption `valid T`.

Other than relying on Lemma 2.5, the proof of Theorem 2.1(ii) in [2] assumes auxiliary results such as the following: “if O_X realises X and O_Y realises Y, then $O_X \mid O_Y$ realises $X \mid Y$”. While this result holds in the particular context of Theorem 2.1(ii), where $X \mid Y$ is known to be reachable and is able to perform a synchronization, it does not hold in general. For instance, consider $X_1 = a[k]$ and $X_2 = b[k]$: the parallel composition $a[k] \mid b[k]$ is not reachable from $a \mid b$. Moreover, even when such conditions are met, building a constructive proof is far from straightforward. These issues led us to revisit the entire argument and develop the following proof strategy for Theorem 2.1(ii):

1. First, we consider the case where neither $\ell(\theta_1)$ nor $\ell(\theta_2)$ is τ and prove that the diagram in Fig. 9a holds. The hypothesis excludes the cases in which θ_1 and θ_2 label synchronizations, thus ruling out the scenarios in which the aforementioned auxiliary lemma occurs. The proved result goes beyond establishing the connectivity of two combined transitions labelled by θ_1 and θ_2: both transitions are forward, and the processes X_1 and X_2' are either identical or connected by a single combined transition. Additionally, we show that at least one among X_1 and X_2' is standard.

2. We then move to the general case, proving that for any connected pair of proof labels θ_1 and θ_2 the diagram in Fig. 9b holds. Analogously to the previous point, the transitions labelled by θ_1 and θ_2 are forward, meaning that our statement is slightly more specific than the original formulation of Theorem 2.1(ii). This refinement helps eliminating non-existent subcases that would arise in the nested pattern matching of combined transitions.

$$\begin{tikzcd}
 & X_2 \\
X_1 \arrow[r, mapsto, two heads] \arrow[d, mapsto, "\theta_1"'] & X_2' \arrow[u, mapsto, "\theta_2"'] \\
X_1' &
\end{tikzcd}$$

(a) Base case of the proof strategy.

$$\begin{tikzcd}
 & & X_2 \\
X_1 \arrow[r, mapsto, two heads] \arrow[d, mapsto, "\theta_1"'] & \cdots \arrow[r, mapsto, two heads] & X_2' \arrow[u, mapsto, "\theta_2"'] \\
X_1' & &
\end{tikzcd}$$

(b) General case of the proof strategy.

Figure 9: Proof strategy for Theorem 2.1(ii).

In the general case, when θ_1 and θ_2 label synchronizations (e.g., when $\theta_1 = \langle|_{\mathrm{L}}\theta_L^1, |_{\mathrm{R}}\theta_R^1\rangle$), the labels of their subterms (e.g., θ_L^1 and θ_R^1) are not τ: this detail allows us to apply the base case of the proof strategy, which provides richer information than the inductive hypothesis of the general Theorem 2.1(ii). That additional information is essential: it enables us to build the desired path between X_1 and X_2', actually with at most two transition steps.

The encoding of Theorem 2.1(ii) follows the plan outlined. The base case makes use of the following elements: a type family `lab_not_tau`, characterizing proof keyed labels whose label is not τ;

a type family max_one_step, encoding the conclusions of the statement; and the recursive function conn_rel_two_base, which proves it.

```
LF lab_not_tau: pr_lab → type =
  | nt_inp: lab T (inp _) → lab_not_tau T
  | nt_out: lab T (out _) → lab_not_tau T;
LF max_one_step: pr_lab → pr_lab → type =
  | c_id: std X1 → fstep X1 T1 X1' → fstep X1 T2 X2 → max_one_step T1 T2
  | c_fw: std X1 → fstep X1 T1 X1' → fstep X1 T3 X2' → fstep X2' T2 X2
       → lab T1 L1 → lab T3 L1 → max_one_step T1 T2
  | c_bw: std X2' → fstep X1 T1 X1' → bstep X1 T3 X2' → fstep X2' T2 X2
       → lab T2 L2 → lab T3 L2 → max_one_step T1 T2;
rec conn_rel_two_base: (g:ctx) [g ⊢ valid T1] → [g ⊢ valid T2] → [g ⊢ conn T1 T2] →
  [g ⊢ lab_not_tau T1] → [g ⊢ lab_not_tau T2] → [g ⊢ max_one_step T1 T2] = ...
```

The proof of this result is given by a long induction on the structure of the given connectivity relation. The predicates (lab Ti Lj), for i,j in $\{1,2,3\}$, which occur in the type family max_one_step, are a technical detail which helps completing few subcases of the proof.

Next, the general case of the proof is addressed by the recursive function conn_rel_two_fstep below, which relies on a dedicated type family as well:

```
LF ex_conn_fstep: pr_lab → pr_lab → type =
  | ex_cf: fstep X1 T1 X1' → fstep X2' T2 X2 → step* X1 X2'
        → reachable X1 → ex_conn_fstep T1 T2;
rec conn_rel_two_fstep: (g:ctx) [g ⊢ valid T1] → [g ⊢ valid T2] → [g ⊢ conn T1 T2]
  → [g ⊢ ex_conn_fstep T1 T2] = ...
```

The proof is given by a long induction on the structure of the given connectivity relation. It requires encoding auxiliary lemmas such as the following: "given a path between two processes X and X', there is a path between $X+\mathbf{0}$ and $X'+\mathbf{0}$", or: "given a forward transition $X \xmapsto{\theta} X'$ where X is standard and $\not{k}(\theta)=k$, then any key $m \neq k$ does not occur in X'".

Finally, Theorem 2.1(ii) is encoded by the following function conn_rel_two. It calls the function conn_rel_two_fstep, applies the loop lemma to reverse one of the two forward transitions, and has all the ingredients to conclude:

```
LF ex_conn_tr: pr_lab → pr_lab → type =
  | ex_c: {S1: step X T1 X'} {S2: step Y T2 Y'} reachable X → conn_tr S1 S2
       → ex_conn_tr T1 T2;
rec conn_rel_two: (g:ctx) [g ⊢ valid T1] → [g ⊢ valid T2] → [g ⊢ conn T1 T2]
  → [g ⊢ ex_conn_tr T1 T2] =
/ total c (conn_rel_two _ _ _ _ _ c) /
fn v1,v2,c ⇒
let [g ⊢ ex_cf F1 F2 S* (rch D S0*)] = conn_rel_two_fstep v1 v2 c in
let [g ⊢ B2] = loop_lemma_one [g ⊢ F2] in
[g ⊢ ex_c (fw F1) (bw B2) (rch D S0*) (ct (fw F1) (bw B2) S*)];
```

4 Conclusions and Future Work

We begin with a brief technical overview of the encoding. The complete formalization consists of less than 2000 lines of code and includes a total of 49 theorems and lemmas. Among them, 13 are direct translations of results stated in Section 2, while the remaining 36 are technical and auxiliary lemmas introduced to support the encoding.

Beluga has proved to be a reliable and expressive proof assistant, well-suited to represent the definitions and properties of CCSK^P. Its use of higher-order abstract syntax (HOAS) offers a convenient approach to handling restrictions – even though CCSK^P does not feature a particularly complex binding structure, unlike, for instance, the π-calculus. Furthermore, Beluga's explicit proof style provides a transparency that is often lost in proof assistants that rely heavily on automation.

However, the lack of automation also comes with drawbacks, mainly the increased length of proof terms. This also follows from the lack of syntactic sugar for existentials, conjunctions and disjunctions, which leads to defining additional type families or splitting theorem statements. Additionally, Beluga provides no built-in mechanism to simplify repeated proof patterns, requiring each similar subcase to be handled individually.

Whether the overall outcome is favorable depends largely on the specific system one aims to formalize. For languages with rich binding structures, the benefits of HOAS alone may outweigh the trade-offs. In our case, however, this advantage is less significant, and we believe that other proof assistants (such as Rocq [4]) might be a better fit for formalizing the system at hand.

To the best of our knowledge, this work provides the first formalization of a reversible concurrent calculus in a proof assistant. We have formally verified the correctness of the notions and results presented in Section 2. We gained a deeper understanding of the system itself, leading to refinements in both definitions and proofs; in particular, we provided an alternative way to represent proof labels compared to the informal definition. We also outlined the adequacy of our encoding.

This work lays the foundations for future reversible concurrent calculi formalizations. The encoding can be adapted to cover the subsystems of CCSK^P, i.e., CCS and CCSK, and can be mapped to existing CCS formalizations. Moreover, it could be extended to include additional portions of [2]. Additionally, it could be translated into other proof assistants, such as Rocq, which are potentially better suited for representing this reversible process calculus. Finally, it can serve as a reference point for future formalizations of other reversible concurrent calculi, such as RCCS.

Acknowledgments

We warmly thank the anonymous reviewers, as well as Clément Aubert and Deivid Vale, for their insightful comments and suggestions, which greatly contributed to improving the paper. This work is supported by the National Science Foundation under Grant No. 2242786 (SHF:Small:Concurrency In Reversible Computations).

References

[1] Clément Aubert & Peter Browning (2023): *Implementation of a Reversible Distributed Calculus*. In Martin Kutrib & Uwe Meyer, editors: *Reversible Computation - 15th International Conference, RC 2023, Giessen, Germany, July 18-19, 2023, Proceedings*, *Lecture Notes in Computer Science* 13960, Springer, pp. 210–217, doi:10.1007/978-3-031-38100-3_13.

[2] Clément Aubert, Iain Phillips & Irek Ulidowski (2024): *Dependence and Independence for Reversible Process Calculi*. *CoRR* abs/2410.14699, doi:10.48550/ARXIV.2410.14699. arXiv:2410.14699.

[3] Clément Aubert (2022): *Concurrencies in Reversible Concurrent Calculi*. In Claudio Antares Mezzina & Krzysztof Podlaski, editors: *Reversible Computation - 14th International Conference, RC 2022, Urbino, Italy, July 5-6, 2022, Proceedings*, *LNCS* 13354, Springer, pp. 146–163, doi:10.1007/978-3-031-09005-9_10.

[4] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series, Springer, doi:10.1007/978-3-662-07964-5.

[5] Gabriele Cecilia & Alberto Momigliano (2024): *A Beluga Formalization of the Harmony Lemma in the π-Calculus*. In Florian Rabe & Claudio Sacerdoti Coen, editors: Proceedings Workshop on *Logical Frameworks and Meta-Languages: Theory and Practice*, Tallinn, Estonia, 8th July 2024, *Electronic Proceedings in Theoretical Computer Science* 404, Open Publishing Association, pp. 1–17, doi:10.4204/EPTCS.404.1.

[6] James Cheney, Michael Norrish & René Vestergaard (2012): *Formalizing Adequacy: A Case Study for Higher-order Abstract Syntax*. *J. Autom. Reason.* 49(2), pp. 209–239, doi:10.1007/S10817-011-9221-6.

[7] Gavin Cox (2009): *SimCCSK: simulation of the reversible process calculi CCSK*. Available at `https://figshare.le.ac.uk/articles/thesis/SimCCSK_simulation_of_the_reversible_process_calculi_CCSK/10091681`.

[8] Vincent Danos & Jean Krivine (2004): *Reversible Communicating Systems*. In Philippa Gardner & Nobuko Yoshida, editors: *CONCUR 2004*, *LNCS* 3170, Springer, pp. 292–307, doi:10.1007/978-3-540-28644-8_19.

[9] Pierpaolo Degano & Corrado Priami (2001): *Enhanced operational semantics*. *ACM Comput. Surv.* 33(2), pp. 135–176, doi:10.1145/384192.384194.

[10] Robert Harper, Furio Honsell & Gordon Plotkin (1993): *A Framework for Defining Logics*. *J. ACM* 40(1), pp. 143–184, doi:10.1145/138027.138060.

[11] 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.

[12] T. F. Melham (1994): *A Mechanized Theory of the π-Calculus in HOL*. *Nordic J. of Computing* 1(1), p. 50–76, doi:10.48456/tr-244.

[13] Robin Milner (1980): *A Calculus of Communicating Systems*. LNCS, Springer-Verlag, doi:10.1007/3-540-10235-3.

[14] Robin Milner, Joachim Parrow & David Walker (1992): *A Calculus of Mobile Processes, I*. *Inf. Comput.* 100(1), pp. 1–40, doi:10.1016/0890-5401(92)90008-4.

[15] Monica Nesi: *A formalization of the process algebra CCS in high order logic*. Technical Report UCAM-CL-TR-278, University of Cambridge, Computer Laboratory, doi:10.48456/tr-278.

[16] Frank Pfenning & Conal Elliott (1988): *Higher-Order Abstract Syntax*. In: *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, ACM Press, pp. 199–208, doi:10.1145/53990.54010.

[17] Iain Phillips & Irek Ulidowski (2007): *Reversing algebraic process calculi*. *J. Log. Algebr. Program.* 73(1-2), pp. 70–96, doi:10.1016/j.jlap.2006.11.002.

[18] Brigitte Pientka: *Beluga Reference Guide*. `https://complogic.cs.mcgill.ca/beluga/userguide2/userguide.pdf`.

[19] 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.

[20] Irek Ulidowski, Iain Phillips & Shoji Yuen (2014): *Concurrency and Reversibility*. In Shigeru Yamashita & Shin-ichi Minato, editors: *Reversible Computation - 6th International Conference, RC 2014, Kyoto, Japan, July 10-11, 2014. Proceedings*, *LNCS* 8507, Springer, pp. 1–14, doi:10.1007/978-3-319-08494-7_1.

A Appendix: Adequacy of the Encoding

To prove the adequacy of our encoding of CCSK$^{\text{P}}$, we follow the approach of Honsell et al. in their Coq formalization of the π-calculus [11]. Our goal is to define an encoding and a decoding function that establish a correspondence between the mathematical objects introduced in Section 2.1 and 2.2 and their counterparts in Beluga; we aim to prove that this correspondence is in fact a bijection. A key requirement for this bijection to be meaningful is compositionality: object-language substitution must map to meta-language substitution [6]. Proving this property involves reasoning about Beluga's meta-theory [18], and while it is essential for the adequacy argument, the proof itself is long and beyond the scope of this paper. For this reason, in the following we are going to assume that compositionality holds.

A.1 Adequacy of syntax

We begin by establishing a bijection between the set of keys K and the type `keys` that encodes them. Let $f\colon \mathsf{K} \to \mathbb{N}$ be an enumeration of K and $g\colon \mathbb{N} \to \texttt{keys}$ the usual bijection mapping 0 to `z`, 1 to `(s z)`, etc. The desired bijection and its inverse are denoted as $\varepsilon^k := g \circ f$ and $\delta^k := f^{-1} \circ g^{-1}$.

As in [11], given a finite set of names $A := \{a_1, \dots, a_n\}$, let $\Gamma_A := (\texttt{a}_1\texttt{:names}, \dots, \texttt{a}_\texttt{n}\texttt{:names})$ be the corresponding Beluga context. We denote as $\mathsf{L}_A = A \cup \bar{A} \cup \{\tau\}$ the subset of labels made from names in A (with the addition of τ). Moreover, we denote as $\texttt{labels}_A = \{\texttt{t} \mid \Gamma_A \vdash \texttt{t:labels}\}$ the collection of terms of type `labels` whose open variables are included in Γ_A: they are terms of the form `(inp a)` or `(out a)` for `a` in Γ_A, in addition to the constant `tau`. There is a bijection $\varepsilon^{\mathsf{L}}_A$ between L_A and $\texttt{labels}_A$, with inverse δ^{L}_A, respectively mapping a to `(inp a)`, $\bar{a}$ to `(out a)` and τ to `tau`.

Similarly, we can build a correspondence between processes and their encoding. Let $\mathbb{X}_A$ be the set of processes whose free names are included in A, and let $\texttt{proc}_A = \{\texttt{t} \mid \Gamma_A \vdash \texttt{t:proc}\}$. We define the *encoding* and *decoding* functions $\varepsilon^{\mathbb{X}}_A$ and $\delta^{\mathbb{X}}_A$ in Fig. 10. Similarly to [11], we consider a function fresh : $\mathscr{P}_{<\omega}(\mathsf{N}) \to \mathsf{N}$ which, given a finite set of names A, returns a fresh name not occurring in A (recall that N is infinite).

$$\begin{array}{rcl}
\varepsilon^{\mathbb{X}}_A : \quad \mathbb{X}_A & \to & \texttt{proc}_A \\
\mathbf{0} & \mapsto & \texttt{null} \\
\alpha.X & \mapsto & (\texttt{pref}\ \varepsilon^{\mathsf{L}}_A(\alpha)\ \varepsilon^{\mathbb{X}}_A(X)) \\
\alpha[k].X & \mapsto & (\texttt{kpref}\ \varepsilon^{\mathsf{L}}_A(\alpha)\ \varepsilon^{k}(k)\ \varepsilon^{\mathbb{X}}_A(X)) \\
X + Y & \mapsto & (\texttt{sum}\ \varepsilon^{\mathbb{X}}_A(X)\ \varepsilon^{\mathbb{X}}_A(Y)) \\
X \mid Y & \mapsto & (\texttt{par}\ \varepsilon^{\mathbb{X}}_A(X)\ \varepsilon^{\mathbb{X}}_A(Y)) \\
X \setminus a & \mapsto & (\texttt{nu}\ \backslash\texttt{a.}(\varepsilon^{\mathbb{X}}_{A,a}(X)))
\end{array}$$

$$\begin{array}{rcll}
\delta^{\mathbb{X}}_A : \quad \texttt{proc}_A & \to & \mathbb{X}_A & \\
\texttt{null} & \mapsto & \mathbf{0} & \\
(\texttt{pref alpha x}) & \mapsto & \delta^{\mathsf{L}}_A(\texttt{alpha}).\,\delta^{\mathbb{X}}_A(\texttt{x}) & \\
(\texttt{kpref alpha k x}) & \mapsto & \delta^{\mathsf{L}}_A(\texttt{alpha})\,[\delta^{k}(\texttt{k})].\,\delta^{\mathbb{X}}_A(\texttt{x}) & \\
(\texttt{sum}\ \texttt{x}_1\ \texttt{x}_2) & \mapsto & \delta^{\mathbb{X}}_A(\texttt{x}_1) + \delta^{\mathbb{X}}_A(\texttt{x}_2) & \\
(\texttt{par}\ \texttt{x}_1\ \texttt{x}_2) & \mapsto & \delta^{\mathbb{X}}_A(\texttt{x}_1) \mid \delta^{\mathbb{X}}_A(\texttt{x}_2) & \\
(\texttt{nu x}) & \mapsto & \delta^{\mathbb{X}}_{A,a}(\texttt{x a}) \setminus a & \text{with } a = \text{fresh}(A)
\end{array}$$

Figure 10: Encoding and decoding functions for processes.

Our encoding of processes is adequate in the sense given by the following theorem:

Theorem A.1 (Adequacy of syntax) *For $A \subset \mathsf{N}$ finite:*

1. $\delta_A^{\mathbb{X}} \circ \varepsilon_A^{\mathbb{X}} = id_{\mathbb{X}_A}$.
2. $\varepsilon_A^{\mathbb{X}} \circ \delta_A^{\mathbb{X}} = id_{\mathtt{proc}_A}$.

Proof. We proceed by induction on the structure of the given process or term of type `proc`. Here, we focus on the most interesting case, i.e., the restriction case. The remaining subcases follow directly from the definitions of $\varepsilon_A^{\mathbb{X}}$ and $\delta_A^{\mathbb{X}}$ and the application of the inductive hypothesis.

Consider $X \setminus a$. Assumed the compatibility between object-language and meta-language substitution, we can rewrite the definitions of $\delta_A^{\mathbb{X}}$ and $\varepsilon_A^{\mathbb{X}}$ and apply the induction hypothesis, obtaining that $(\delta_A^{\mathbb{X}} \circ \varepsilon_A^{\mathbb{X}})(X \setminus a) = (X\{b/a\}) \setminus b$. In this equation, $b = \text{fresh}(A)$ does not occur free in $X \setminus a$ and $X\{b/a\}$ denotes the substitution of b for a in X. Recalling that processes are considered equal up to α-equivalence, the proof of this case is complete.

Now consider `nu t`. Similarly as above, $(\varepsilon_A^{\mathbb{X}} \circ \delta_A^{\mathbb{X}})(\mathtt{nu\ t}) = (\mathtt{nu\ \backslash a.(t\ a)})$, which is equal to the given term `nu t` up to η-conversion. □

A.2 Adequacy of semantics

Before establishing a correspondence between proof labels and terms of type `pr_lab`, we define the notion of occurrence of a free name in a proof label. Given $\theta = \nu\alpha[k]$, we say that the name a occurs free in θ if $\alpha = a$ or $\alpha = \overline{a}$; given $\theta = \langle |_{\mathrm{L}}\nu_1\lambda[k], |_{\mathrm{R}}\nu_2\overline{\lambda}[k]\rangle$, we say that a occurs free in θ if $\lambda = a$ or $\lambda = \overline{a}$.

Given $A \subset \mathsf{N}$ finite, let Θ_A be the set of proof labels whose free names are included in A and let $\mathtt{pr\_lab}_A = \{\mathtt{t} \mid \Gamma_A \vdash \mathtt{t:pr\_lab}$ and $\mathtt{valid\ t}$ holds$\}$. We define the encoding and decoding functions ε_A^{Θ} and δ_A^{Θ} for proof labels as in Fig. 11.

$$\begin{array}{rcl}
\varepsilon_A^{\Theta}: \quad \Theta_A & \to & \mathtt{pr\_lab}_A \\
\alpha[k] & \mapsto & (\mathtt{pr\_base}\ \varepsilon_A^{\mathsf{L}}(\alpha)\ \varepsilon^k(k)\ ) \\
\langle |_{\mathrm{L}}\nu_1\lambda[k], |_{\mathrm{R}}\nu_2\overline{\lambda}[k]\rangle & \mapsto & (\mathtt{pr\_sync}\ \varepsilon_A^{\Theta}(\nu_1\lambda[k])\ \varepsilon_A^{\Theta}(\nu_2\overline{\lambda}[k])) \\
+_{\mathrm{L}}\theta & \mapsto & (\mathtt{pr\_suml}\ \varepsilon_A^{\Theta}(\theta)) \\
+_{\mathrm{R}}\theta & \mapsto & (\mathtt{pr\_sumr}\ \varepsilon_A^{\Theta}(\theta)) \\
|_{\mathrm{L}}\ \theta & \mapsto & (\mathtt{pr\_parl}\ \varepsilon_A^{\Theta}(\theta)) \\
|_{\mathrm{R}}\ \theta & \mapsto & (\mathtt{pr\_parr}\ \varepsilon_A^{\Theta}(\theta))
\end{array}$$

$$\begin{array}{rcl}
\delta_A^{\Theta}: \quad \mathtt{pr\_lab}_A & \to & \Theta_A \\
(\mathtt{pr\_base\ alpha\ k}) & \mapsto & \delta_A^{\mathsf{L}}(\mathtt{alpha})\,[\delta^k(\mathtt{k})] \\
(\mathtt{pr\_suml\ t}) & \mapsto & +_{\mathrm{L}}\,\delta_A^{\Theta}(\mathtt{t}) \\
(\mathtt{pr\_sumr\ t}) & \mapsto & +_{\mathrm{R}}\,\delta_A^{\Theta}(\mathtt{t}) \\
(\mathtt{pr\_parl\ t}) & \mapsto & |_{\mathrm{L}}\ \delta_A^{\Theta}(\mathtt{t}) \\
(\mathtt{pr\_parr\ t}) & \mapsto & |_{\mathrm{R}}\ \delta_A^{\Theta}(\mathtt{t}) \\
(\mathtt{pr\_sync}\ \mathtt{t}_1\ \mathtt{t}_2) & \mapsto & \langle |_{\mathrm{L}}\delta_A^{\Theta}(\mathtt{t}_1), |_{\mathrm{R}}\delta_A^{\Theta}(\mathtt{t}_2)\rangle
\end{array}$$

Figure 11: Encoding and decoding functions for proof labels.

The proof that the two functions ε_A^{Θ} and δ_A^{Θ} are well defined and reciprocally inverse requires the following compatibility results:

Lemma A.2 (Compatibility between ℓ and `lab`) *Given $A \subset \mathsf{N}$ finite:*

1. *For any $\theta \in \Theta_A$ and $\alpha \in \mathsf{L}_A$, if $\ell(\theta) = \alpha$ then* $(\texttt{lab}\ \ \varepsilon^{\Theta}_A(\theta)\ \ \varepsilon^{\mathsf{L}}_A(\alpha))$ *holds.*
2. *For any* $\texttt{t} \in \texttt{pr\_lab}_A$ *and* $\texttt{alpha} \in \texttt{labels}_A$*, if* $(\texttt{lab t alpha})$ *holds then* $\ell(\delta^{\Theta}_A(\texttt{t})) = \delta^{\mathsf{L}}_A(\texttt{alpha})$.

Proof. By induction on the structure of the given proof label or term of type `pr_lab`. □

Lemma A.3 (Compatibility between $\mathscr{k}$ and `key`) *Given $A \subset \mathsf{N}$ finite:*

1. *For any $\theta \in \Theta_A$ and $k \in \mathsf{K}$, if $\mathscr{k}(\theta) = k$ then* $(\texttt{key}\ \ \varepsilon^{\Theta}_A(\theta)\ \ \varepsilon^{k}(k))$ *holds.*
2. *For any* $\texttt{t} \in \texttt{pr\_lab}_A$ *and* $\texttt{k} \in \texttt{keys}$*, if* $(\texttt{key t k})$ *holds then* $\mathscr{k}(\delta^{\Theta}_A(\texttt{t})) = \delta^{k}(\texttt{k})$.

Proof. By induction on the structure of the given proof label or term of type `pr_lab`. □

Our encoding of proof labels is adequate in the sense given by the following theorem:

Theorem A.4 (Adequacy of proof labels) *For $A \subset \mathsf{N}$ finite:*

1. *ε^{Θ}_A is well defined. In particular, for any $\theta \in \Theta_A$,* $(\texttt{valid}\ \ \varepsilon^{\Theta}_A(\theta))$ *holds.*
2. *δ^{Θ}_A is well defined. In particular, for any* $\texttt{t:pr\_lab}$ *for which* $(\texttt{valid t})$ *holds,* $\delta^{\Theta}_A(\texttt{t})$ *is a valid proof label.*
3. $\delta^{\Theta}_A \circ \varepsilon^{\Theta}_A = id_{\Theta_A}$.
4. $\varepsilon^{\Theta}_A \circ \delta^{\Theta}_A = id_{\texttt{pr\_lab}_A}$.

Proof. By induction on the structure of the given proof label or term of type `pr_lab`. □

Before proving the adequacy of semantics, it is necessary to prove the existence of a correspondence between the type families `std`, `notin` and `neq`, present in the encoding of the LTS rules, and their mathematical counterpart. We state the related lemmas, all proved by structural induction.

Lemma A.5 *Given $A \subset \mathsf{N}$ finite:*

1. *For any $X \in \mathbb{X}_A$, if X is standard then* $(\texttt{std}\ \ \varepsilon^{\mathbb{X}}_A(X))$ *holds.*
2. *For any* $\texttt{X} \in \texttt{proc}_A$*, if* $(\texttt{std X})$ *holds then* $\delta^{\mathbb{X}}_A(\texttt{X})$ *is standard.*

Lemma A.6 *Given $A \subset \mathsf{N}$ finite:*

1. *For any $k \in \mathsf{K}$ and $X \in \mathbb{X}_A$, if $k \notin \mathsf{keys}(X)$ then* $(\texttt{notin}\ \ \varepsilon^{k}(k)\ \ \varepsilon^{\mathbb{X}}_A(X))$ *holds.*
2. *For any* $\texttt{k} \in \texttt{keys}$ *and* $\texttt{X} \in \texttt{proc}_A$*, if* $(\texttt{notin K X})$ *holds then* $\delta^{k}(\texttt{k}) \notin \mathsf{keys}(\delta^{\mathbb{X}}_A(\texttt{X}))$.

Lemma A.7

1. *For any $k, m \in \mathsf{K}$, if $k \neq m$ then* $(\texttt{neq}\ \ \varepsilon^{k}(k)\ \ \varepsilon^{k}(m))$ *holds.*
2. *For any* $\texttt{K},\texttt{M} \in \texttt{keys}$*, if* $(\texttt{neq K M})$ *holds then* $\delta^{k}(\texttt{K}) \neq \delta^{k}(\texttt{M})$.

Our encoding of the LTS in Fig. 2 is adequate in the sense given by the following theorem, which relates combined transitions and their encoding:

Theorem A.8 (Adequacy of semantics) *For $A \subset \mathsf{N}$ finite:*

1. *For all $X, Y \in \mathbb{X}$ and $\theta \in \Theta_A$, if $X \xmapsto{\theta} Y$ then* $(\texttt{fstep}\ \ \varepsilon^{\mathbb{X}}_A(X)\ \ \varepsilon^{\Theta}_A(\theta)\ \ \varepsilon^{\mathbb{X}}_A(Y))$ *holds.*
2. *For all $X, Y \in \mathbb{X}$ and $\theta \in \Theta_A$, if $X \overset{\theta}{\leadsto} Y$ then* $(\texttt{bstep}\ \ \varepsilon^{\mathbb{X}}_A(X)\ \ \varepsilon^{\Theta}_A(\theta)\ \ \varepsilon^{\mathbb{X}}_A(Y))$ *holds.*
3. *For all* $\texttt{X}$, $\texttt{Y} \in \texttt{proc}_A$ *and* $\texttt{T} \in \texttt{pr\_lab}_A$*, if* $(\texttt{fstep X T Y})$ *holds then* $\delta^{\mathbb{X}}_A(X) \xmapsto{\delta^{\Theta}_A(\theta)} \delta^{\mathbb{X}}_A(Y)$.
4. *For all* $\texttt{X}$, $\texttt{Y} \in \texttt{proc}_A$ *and* $\texttt{T} \in \texttt{pr\_lab}_A$*, if* $(\texttt{bstep X T Y})$ *holds then* $\delta^{\mathbb{X}}_A(X) \overset{\delta^{\Theta}_A(\theta)}{\leadsto} \delta^{\mathbb{X}}_A(Y)$.

Proof. By induction on the structure of the given transition. □