A Core Calculus for Equational Proofs of Distributed Cryptographic Protocols: Technical Report

Kristina Sojakova Mihai Codescu Joshua Gancher

April 28, 2023

Acknowledgement

This project was funded through the NGI Assure Fund, a fund established by NLnet with financial support from the European Commission's Next Generation Internet programme, under the aegis of DG Communications Networks, Content and Technology under grant agreement No. 957073.

1 Syntax of IPDL

IPDL is built from four layers: *protocols* are networks of mutually interacting *reactions*, which are simple monadic programs. Each reaction computes an *expression* probabilistically: *i.e.*, the computation may include sampling from *distributions*. In the context of a protocol, a reaction operates on a unique *channel* and may read from other channels, thereby utilizing computations coming from other reactions. The syntax and judgements of IPDL are outlined in Figures 1, 2, respectively, and are parameterized by a user-defined *signature* Σ:

Definition 1 (Signature)**.** *An IPDL signature Σ is a finite collection of:*

- *type symbols* t*;*
- *typed function symbols* $\mathsf{f} : \tau \to \sigma$*; and*
- *typed distribution symbols* $\mathsf{d} : \tau \twoheadrightarrow \sigma$*.*

We have a minimal set of data types, including the unit type $\mathsf{1}$, Booleans, products, as well as arbitrary type symbols t, drawn from the signature Σ. Expressions are used for non-probabilistic computations, and are standard. All values in IPDL are bitstrings of a length given by data types, so we annotate the operations $\mathsf{fst}_{\tau\times\sigma}$ and $\mathsf{snd}_{\tau\times\sigma}$ with the type of the pair to determine the index to split the pair into two; for readability we omit this subscript whenever appropriate. Function symbols f must be declared in the signature Σ, and for a constant $\mathsf{f} : \mathsf{1} \to \tau$, we write f in place of $\mathsf{f}\ \checkmark$. Substitutions $\theta : \Gamma_1 \to \Gamma_2$ between type contexts are standard.

Analogously to function symbols, distribution symbols d must be declared in the signature Σ, and for a constant $\mathsf{d} : \mathsf{1} \twoheadrightarrow \tau$, we write $\mathsf{samp}\ \mathsf{d}$ instead of $\mathsf{samp}\ (\mathsf{d}\ \checkmark)$. As mentioned above, reactions are monadic programs which may return expressions, sample from distributions, read from channels, branch on a value of type Bool, and sequentially compose. For readability, we often omit the type of the bound variable in a sequential composition, and write $x \leftarrow \mathsf{read}\ c;\ R$ and $x \leftarrow \mathsf{samp}\ d;\ R$ simply as $x \leftarrow c;\ R$ and $x \leftarrow d;\ R$ wherever appropriate. Protocols in IPDL are given by a simple but expressive syntax: channel assignment $o := R$ assigns the reaction R to channel o; parallel composition $P \mid\mid Q$ allows P and Q to freely interact concurrently; and channel generation $\mathsf{new}\ o : \tau\ \mathsf{in}\ P$ creates a new, internal channel for use in P. *Embeddings* $\phi : \Delta_1 \to \Delta_2$ between channel contexts are injective, type-preserving mappings specifying how to rename channels in Δ_2 to fit in the larger context Δ_1.

1.1 Typing

We restrict our attention to well-typed IPDL constructs. In addition to respecting data types, the typing judgments guarantee that all reads from channels in reactions are in scope, and that all channels are assigned at most one reaction in protocols. The typing $\Gamma \vdash e : \tau$ and $\Gamma \vdash d : \tau$ for expressions and distributions is standard, see Figures 3 and 4. Figure 5 shows the typing rules for reactions. Intuitively, $\Delta;\ \Gamma \vdash R : I \to \tau$ holds when R uses variables

Data Types	τ, σ	::=	$\mathsf{t} \mid \mathsf{1} \mid \mathsf{Bool} \mid \tau \times \tau$
Expressions	e	::=	$x \mid \checkmark \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{f}\ e \mid (e_1, e_2) \mid \mathsf{fst}_{\tau\times\sigma}\ e \mid \mathsf{snd}_{\tau\times\sigma}\ e$
Distributions	d	::=	$\mathsf{d}\ e$
Channels	i, o, c		
Reactions	R, S	::=	$\mathsf{ret}\ e \mid \mathsf{samp}\ d \mid \mathsf{read}\ c \mid \mathsf{if}\ e\ \mathsf{then}\ R_1\ \mathsf{else}\ R_2 \mid x : \sigma \leftarrow R;\ S$
Protocols	P, Q	::=	$0 \mid o := R \mid P \mid\mid Q \mid \mathsf{new}\ o : \tau\ \mathsf{in}\ P$
Channel Sets	I, O	::=	$\{c_1, \ldots, c_n\}$
Type Contexts	Γ	::=	$\cdot \mid \Gamma, x : \tau$
Channel Contexts	Δ	::=	$\cdot \mid \Delta, c : \tau$

Figure 1: Syntax of IPDL .

Expression Typing	$\Gamma \vdash e : \tau$
Distribution Typing	$\Gamma \vdash d : \tau$
Reaction Typing	$\Delta;\ \Gamma \vdash R : I \to \tau$
Protocol Typing	$\Delta \vdash P : I \to O$
Substitutions	$\theta : \Gamma_1 \to \Gamma_2$
Embeddings	$\phi : \Delta_1 \to \Delta_2$
Expression Equality	$\Gamma \vdash e_1 = e_2 : \tau$
Distribution Equality	$\Gamma \vdash d_1 = d_2 : \tau$
Reaction Equality	$\Delta;\ \Gamma \vdash R_1 = R_2 : I \to \tau$
Protocol Equality (Strict)	$\Delta \vdash P_1 = P_2 : I \to O$

Figure 2: Judgements of the exact fragment of ipdl.

in Γ, reads from channels in I typed according to Δ, and returns a value of type τ. Figure 6 gives the typing rules for protocols: $\Delta \vdash P : I \to O$ holds when P uses inputs in I to assign reactions to the channels in O, all typed according to Δ.

Channel assignment $o := R$ has the type $I \to \{o\}$ when R is well-typed with an empty variable context, making use of inputs from I as well as of o. We allow R to read from its own output o to express divergence: the protocol $o := \mathsf{read}\ o$ cannot reduce, which is useful for (conditionally) deactivating certain outputs. The typing rule for parallel composition $P \mid\mid Q$ states that P may use the outputs of Q as inputs while defining its own outputs, and vice versa. Importantly, the typing rules ensure that the outputs of P and Q are disjoint so that each channel carries a unique reaction. Finally, the rule for channel generation allows a protocol to select a fresh channel name o, assign it a type τ, and use it for internal computation and communication. Protocol typing plays a crucial role for modeling security. Simulation-based security in IPDL is modeled by the existence of a *simulator* with an appropriate typing judgment, $\Delta \vdash \mathsf{Sim} : I \to O$. Restricting the behavior of Sim to only use inputs along I is necessary to rule out trivial results (*e.g.*, Sim simply copies a secret from the specification).

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash \checkmark : 1} \qquad \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}} \qquad \frac{\mathsf{f} : \sigma \to \tau \in \Sigma \qquad \Gamma \vdash e : \sigma}{\Gamma \vdash \mathsf{f}\ e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \sigma \times \tau}{\Gamma \vdash \mathsf{fst}_{\sigma\times\tau}\ e : \sigma} \qquad \frac{\Gamma \vdash e : \sigma \times \tau}{\Gamma \vdash \mathsf{snd}_{\sigma\times\tau}\ e : \tau}$$

Figure 3: Typing for IPDL expressions.

$$\boxed{\Gamma \vdash d : \tau}$$

$$\frac{\mathsf{d} : \sigma \twoheadrightarrow \tau \in \Sigma \qquad \Gamma \vdash e : \sigma}{\Gamma \vdash \mathsf{d}\ e : \tau}$$

Figure 4: Typing for IPDL distributions.

$$\boxed{\Delta;\ \Gamma \vdash R : I \to \tau}$$

$$\frac{\Gamma \vdash e : \tau}{\Delta;\ \Gamma \vdash \mathsf{ret}\ e : I \to \tau} \qquad \frac{\Gamma \vdash d : \tau}{\Delta;\ \Gamma \vdash \mathsf{samp}\ d : I \to \tau} \qquad \frac{i : \tau \in \Delta \qquad i \in I}{\Delta;\ \Gamma \vdash \mathsf{read}\ i : I \to \tau}$$

$$\frac{\Gamma \vdash e : \mathsf{Bool} \qquad \Delta;\ \Gamma \vdash R_1 : I \to \tau \qquad \Delta;\ \Gamma \vdash R_2 : I \to \tau}{\Delta;\ \Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ R_1\ \mathsf{else}\ R_2 : I \to \tau} \qquad \frac{\Delta;\ \Gamma \vdash R : I \to \sigma \qquad \Delta;\ \Gamma, x : \sigma \vdash S : I \to \tau}{\Delta;\ \Gamma \vdash (x : \sigma \leftarrow R;\ S) : I \to \tau}$$

Figure 5: Typing for IPDL reactions.

1.2 Equational Logic

We now present the equational logic of IPDL . As mentioned above, the logic is divided into *exact* rules that establish semantic equivalences between protocols, and *approximate* rules that are used to discharge computational indistinguishability assumptions.

1.2.1 Exact Equality

The bulk of the reasoning in IPDL is done using exact equalities. At the expression level, we assume an ambient finite set of axioms of the form $\Gamma \vdash e_1 = e_2 : \tau$, where $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$. The rules for expression and distribution equality are standard, see Figures 7 and 8.

At the reaction level, we analogously assume an ambient finite set of axioms of the form $\Delta;\ \Gamma \vdash R_1 = R_2 : I \to \tau$, where $\Delta;\ \Gamma \vdash R_1 : I \to \tau$ and $\Delta;\ \Gamma \vdash R_2 : I \to \tau$. The rules for reaction equality, shown in Figures 9 and 10, ensure in particular that reactions form a *commutative monad*: we have

$$\big(x \leftarrow R_1;\ y \leftarrow R_2;\ S(x,y)\big) = \big(y \leftarrow R_2;\ x \leftarrow R_1;\ S(x,y)\big)$$

whenever R_2 does not depend on x. All expected equivalences for commutative monads hold for reactions, including the usual monad laws and congruence of equivalence under monadic bind. The SAMP-PURE rule allows us to drop an unused sampling, and the READ-DET rule allows us to replace two reads from the same channel by a single one. The rules IF-LEFT, IF-RIGHT, and IF-EXT allow us to manipulate conditionals.

At the protocol level, we similarly assume an ambient finite set of axioms of the form $\Delta \vdash P_1 = P_2 : I \to O$, where $\Delta \vdash P_1 : I \to O$ and $\Delta \vdash P_2 : I \to O$. We use these axioms to specify user-defined functional assumptions, *e.g.*, the correctness of decryption. Exact protocol equivalences allow us to reason about communication between subprotocols and functional correctness, and to simplify intermediate computations. We will see later that exact equivalence implies the existence of a *bisimulation* on protocols, which in turn implies perfect computational indistinguishability against an arbitrary distinguisher. The rules for the exact equality of protocols are in Figures 11, 12; we now describe them informally.

The COMP-NEW rule allows us to permute parallel composition and the creation of a new channel, and the same as *scope extrusion* in process calculi [**?**]. The ABSORB-LEFT rule allows us to discard a component in a parallel composition if it has no outputs; this allows us to eliminate internal channels once they are no longer used. The DIVERGE rule allows us to simplify diverging reactions: if a channel reads from itself and continues as an arbitrary reaction R, then we can safely discard R as we will never reach it in the first place. The three (un)folding rules FOLD-IF-LEFT, FOLD-IF-RIGHT, and FOLD-BIND allow us to simplify composite reactions by bringing their

$$\boxed{\Delta \vdash P : I \to O}$$

$$\frac{}{\Delta \vdash \mathsf{0} : I \to \varnothing} \qquad \frac{o : \tau \in \Delta \qquad o \notin I \qquad \Delta;\ \cdot \vdash R : I \cup \{o\} \to \tau}{\Delta \vdash (o := R) : I \to \{o\}}$$

$$\frac{\Delta \vdash P : I \cup O_2 \to O_1 \qquad \Delta \vdash Q : I \cup O_1 \to O_2}{\Delta \vdash P \parallel Q : I \to O_1 \cup O_2} \qquad \frac{\Delta, o : \tau \vdash P : I \to O \cup \{o\}}{\Delta \vdash (\mathsf{new}\ o : \tau\ \mathsf{in}\ P) : I \to O}$$

Figure 6: Typing for IPDL protocols.

components into the protocol level as separate internal channels. The rule SUBSUME states that channel dependency is transitive: if we depend on o_1, and o_1 in turn depends on o_0, then we also depend on o_0, and this dependency can be made explicit. The SUBST rule allows us to inline certain reactions into read commands. Inlining $o_1 := R_1$ into $o_2 := x \leftarrow \mathsf{read}\ o_1;\ R_2$ is sound provided R_1 is *duplicable*: observing two independent results of evaluating R_1 is equivalent to observing the same result twice. This side condition is easily discharged whenever R_1 does not contain probabilistic sampling. Finally, the DROP rule allows dropping unused reads from channels in certain situations. Due to timing dependencies among channels, we only allow dropping reads from the channel $o_1 := R_1$ in the context of $o_2 := \_ \leftarrow \mathsf{read}\ o_1;\ R_2$ when we have that $(\_ \leftarrow R_1;\ R_2) = R_2$. This side condition is met whenever all reads present in R_1 are also present in R_2.

$$\boxed{\Gamma \vdash e_1 = e_2 : \tau}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e = e : \tau}\ \text{REFL} \qquad \frac{\Gamma \vdash e_1 = e_2 : \tau}{\Gamma \vdash e_2 = e_1 : \tau}\ \text{SYM} \qquad \frac{\Gamma \vdash e_1 = e_2 : \tau \qquad \Gamma \vdash e_2 = e_3 : \tau}{\Gamma \vdash e_1 = e_3 : \tau}\ \text{TRANS}$$

$$\frac{\Gamma \vdash e_1 = e_2 : \tau\ \mathsf{axiom}}{\Gamma \vdash e_1 = e_2 : \tau}\ \text{AXIOM} \qquad \frac{\theta : \Gamma_1 \to \Gamma_2 \qquad \Gamma_2 \vdash e_1 = e_2 : \tau}{\Gamma_1 \vdash \theta^\star(e_1) = \theta^\star(e_2) : \tau}\ \text{SUBST}$$

$$\frac{\mathsf{f} : \sigma \to \tau \in \Sigma \qquad \Gamma \vdash e = e' : \sigma}{\Gamma \vdash \mathsf{f}\ e = \mathsf{f}\ e' : \tau}\ \text{APP-CONG} \qquad \frac{\Gamma \vdash e_1 = e_1' : \tau_1 \qquad \Gamma \vdash e_2 = e_2' : \tau_2}{\Gamma \vdash (e_1, e_2) = (e_1', e_2') : \tau_1 \times \tau_2}\ \text{PAIR-CONG}$$

$$\frac{\Gamma \vdash e = e' : \sigma \times \tau}{\Gamma \vdash \mathsf{fst}_{\sigma\times\tau}\ e = \mathsf{fst}_{\sigma\times\tau}\ e' : \sigma}\ \text{FST-CONG} \qquad \frac{\Gamma \vdash e = e' : \sigma \times \tau}{\Gamma \vdash \mathsf{snd}_{\sigma\times\tau}\ e = \mathsf{snd}_{\sigma\times\tau}\ e' : \tau}\ \text{SND-CONG}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{fst}_{\tau_1\times\tau_2}\ (e_1, e_2) = e_1 : \tau_1}\ \text{FST-PAIR} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{snd}_{\tau_1\times\tau_2}\ (e_1, e_2) = e_2 : \tau_2}\ \text{SND-PAIR}$$

$$\frac{\Gamma \vdash e : \sigma \times \tau}{\Gamma \vdash e = \left(\mathsf{fst}_{\sigma\times\tau}\ e,\ \mathsf{snd}_{\sigma\times\tau}\ e\right) : \sigma \times \tau}\ \text{PAIR-EXT} \qquad \frac{\Gamma \vdash e : 1}{\Gamma \vdash e = \checkmark : 1}\ \text{ONE-EXT}$$

Figure 7: Equality for IPDL expressions.

1.2.2 Approximate Equality

The equational theory for the approximate fragment of IPDL consists of two layers: one for the *approximate equality* of protocols, and one for the *asymptotic equality* of protocol families as functions of the security parameter $\lambda \in \mathbb{N}$. The approximate equality judgement $\Delta \vdash P \approx Q : I \to O\ \mathsf{width}\ k\ \mathsf{length}\ l$ equates two protocols $\Delta \vdash P : I \to O$ and $\Delta \vdash Q : I \to O$ with identical typing judgements. We think of these as corresponding to a specific security

$$\boxed{\Gamma \vdash d_1 = d_2 : \tau}$$

$$\frac{\mathsf{d} : \sigma \twoheadrightarrow \tau \in \Sigma \qquad \Gamma \vdash d = d' : \sigma}{\Gamma \vdash \mathsf{d}\ e = \mathsf{d}\ e' : \tau}\ \textsc{app-cong}$$

Figure 8: Equality for IPDL distributions.

$$\boxed{\Delta;\ \Gamma \vdash R_1 = R_2 : I \to \tau}$$

$$\frac{\Delta;\ \Gamma \vdash R : I \to \tau}{\Delta;\ \Gamma \vdash R = R : I \to \tau}\ \textsc{refl} \qquad \frac{\Delta;\ \Gamma \vdash R_1 = R_2 : I \to \tau}{\Delta;\ \Gamma \vdash R_2 = R_1 : I \to \tau}\ \textsc{sym}$$

$$\frac{\Delta;\ \Gamma \vdash R_1 = R_2 : I \to \tau \qquad \Delta;\ \Gamma \vdash R_2 = R_3 : I \to \tau}{\Delta;\ \Gamma \vdash R_1 = R_3 : I \to \tau}\ \textsc{trans} \qquad \frac{\Delta;\ \Gamma \vdash R_1 = R_2 : I \to \tau\ \mathsf{axiom}}{\Delta;\ \Gamma \vdash R_1 = R_2 : I \to \tau}\ \textsc{axiom}$$

$$\frac{i \notin I \qquad \Delta;\ \Gamma \vdash R_1 = R_2 : I \to \tau}{\Delta;\ \Gamma \vdash R_1 = R_2 : I \cup \{i\} \to \tau}\ \textsc{input-unused} \qquad \frac{\theta : \Gamma_1 \to \Gamma_2 \qquad \Delta;\ \Gamma_2 \vdash R_1 = R_2 : I \to \tau}{\Delta;\ \Gamma_1 \vdash \theta^\star(R_1) = \theta^\star(R_2) : I \to \tau}\ \textsc{subst}$$

$$\frac{\phi : \Delta_1 \to \Delta_2 \qquad \Delta_2;\ \Gamma \vdash R_1 = R_2 : I \to \tau}{\Delta_1;\ \Gamma \vdash \phi^\star(R_1) = \phi^\star(R_2) : \phi^\star(I) \to \tau}\ \textsc{embed} \qquad \frac{\Gamma \vdash e = e' : \tau}{\Delta;\ \Gamma \vdash \mathsf{ret}\ e = \mathsf{ret}\ e' : I \to \tau}\ \textsc{cong-ret}$$

$$\frac{\Gamma \vdash d = d' : \sigma}{\Delta;\ \Gamma \vdash \mathsf{samp}\ d = \mathsf{samp}\ d' : I \to \tau}\ \textsc{cong-samp}$$

$$\frac{\Gamma \vdash e = e' : \mathsf{Bool} \qquad \Delta;\ \Gamma \vdash R_1 = R_1' : I \to \tau \qquad \Delta;\ \Gamma \vdash R_2 = R_2' : I \to \tau}{\Delta;\ \Gamma \vdash \big(\mathsf{if}\ e\ \mathsf{then}\ R_1\ \mathsf{else}\ R_2\big) = \big(\mathsf{if}\ e'\ \mathsf{then}\ R_1'\ \mathsf{else}\ R_2'\big) : I \to \tau}\ \textsc{cong-if}$$

$$\frac{\Delta;\ \Gamma \vdash R = R' : I \to \sigma \qquad \Delta;\ \Gamma, x : \sigma \vdash S = S' : I \to \tau}{\Delta;\ \Gamma \vdash (x : \sigma \leftarrow R;\ S) = (x : \sigma \leftarrow R';\ S') : I \to \tau}\ \textsc{cong-bind}$$

Figure 9: Equality for IPDL reactions. Additional rules are given in Figure 10.

parameter λ. Analogously to exact protocol equality, we assume an ambient finite set of *approximate axioms* of the form $\Delta \vdash P \approx Q : I \to O$, where $\Delta \vdash P : I \to O$ and $\Delta \vdash Q : I \to O$. These axioms capture cryptographic assumptions on computational indistinguishability.

The parameters $k, l \in \mathbb{N}$ track the size of the derivation. The *width* parameter k simply counts the number of invocations of axioms applied during the proof: applying a single approximate axiom incurs $k = 1$, and we sum up the two values of k whenever we use transitivity. In the asymptotic equality judgement, k becomes a function of the security parameter λ and we require that it be bounded by a polynomial in λ: even though each individual axiom invocation introduces a negligible error, the sum of exponentially many negligible errors may not be negligible anymore.

Since most nontrivial reasoning in IPDL is done in the exact half, the approximate equality rules are used mostly to apply indistinguishability assumptions nested deeply inside protocols. The *length* parameter l tracks the largest size of such a nesting – also known as a *program context*. In the asymptotic equality judgement, we again require that l be bounded as a function of λ by a polynomial: exponentially large IPDL contexts could in principle be used to encode exponential-time probabilistic computations. An IPDL program context surrounding an indistinguishability assumption is formally a part of the adversary, and as such it must be resource-bounded for the indistinguishability assumption to apply.

In IPDL, the bound on resources is given by a *symbolic size* function $|\cdot|$ defined for expressions, reactions, and

$$\boxed{\Delta;\ \Gamma \vdash R_1 = R_2 : I \to \tau}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \Delta;\ \Gamma, x : \sigma \vdash R : I \to \tau}{\Delta;\ \Gamma \vdash (x : \sigma \leftarrow \mathsf{ret}\ e;\ R) = R[x := e] : I \to \tau}\ \text{RET-BIND} \qquad \frac{\Delta;\ \Gamma \vdash R : I \to \tau}{\Delta;\ \Gamma \vdash (x : \tau \leftarrow R;\ \mathsf{ret}\ x) = R : I \to \tau}\ \text{BIND-RET}$$

$$\frac{\Delta;\ \Gamma \vdash R_1 : I \to \sigma_1 \qquad \Delta;\ \Gamma, x_1 : \sigma_1 \vdash R_2 : I \to \sigma_2 \qquad \Delta;\ \Gamma, x_2 : \sigma_2 \vdash S : I \to \tau}{\Delta;\ \Gamma \vdash \big(x_2 : \sigma_2 \leftarrow (x_1 : \sigma_1 \leftarrow R_1;\ R_2);\ S\big) = \big(x_1 : \sigma_1 \leftarrow R_1;\ x_2 : \sigma_2 \leftarrow R_2;\ S\big) : I \to \tau}\ \text{BIND-BIND}$$

$$\frac{\Delta;\ \Gamma \vdash R_1 : I \to \sigma_1 \qquad \Delta;\ \Gamma \vdash R_2 : I \to \sigma_2 \qquad \Delta;\ \Gamma, x_1 : \sigma_1, x_2 : \sigma_2 \vdash S : I \to \tau}{\Delta;\ \Gamma \vdash \big(x_1 : \sigma_1 \leftarrow R_1;\ x_2 : \sigma_2 \leftarrow R_2;\ S\big) = \big(x_2 : \sigma_2 \leftarrow R_2;\ x_1 : \sigma_1 \leftarrow R_1;\ S\big) : I \to \tau}\ \text{EXCH}$$

$$\frac{\Gamma \vdash d : \sigma \qquad \Delta;\ \Gamma \vdash R : I \to \tau}{\Delta;\ \Gamma \vdash (x : \sigma \leftarrow \mathsf{samp}\ d;\ R) = R : I \to \tau}\ \text{SAMP-PURE}$$

$$\frac{i : \sigma \in \Delta \qquad i \in I \qquad \Delta;\ \Gamma, x : \sigma, y : \sigma \vdash R : I \to \tau}{\Delta;\ \Gamma \vdash \big(x : \sigma \leftarrow \mathsf{read}\ i;\ y : \sigma \leftarrow \mathsf{read}\ i;\ R\big) = \big(x : \sigma \leftarrow \mathsf{read}\ i;\ R[y := x]\big) : I \to \tau}\ \text{READ-DET}$$

$$\frac{\Delta;\ \Gamma \vdash R_1 : I \to \tau \qquad \Delta;\ \Gamma \vdash R_2 : I \to \tau}{\Delta;\ \Gamma \vdash \big(\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ R_1\ \mathsf{else}\ R_2\big) = R_1 : I \to \tau}\ \text{IF-LEFT}$$

$$\frac{\Delta;\ \Gamma \vdash R_1 : I \to \tau \qquad \Delta;\ \Gamma \vdash R_2 : I \to \tau}{\Delta;\ \Gamma \vdash \big(\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ R_1\ \mathsf{else}\ R_2\big) = R_2 : I \to \tau}\ \text{IF-RIGHT}$$

$$\frac{\Delta;\ \Gamma, x : \mathsf{Bool} \vdash R : I \to \tau \qquad \Gamma \vdash e : \mathsf{Bool}}{\Delta;\ \Gamma \vdash \big(\mathsf{if}\ e\ \mathsf{then}\ R[x := \mathsf{true}]\ \mathsf{else}\ R[x := \mathsf{false}]\big) = R[x := e] : I \to \tau}\ \text{IF-EXT}$$

Figure 10: Equality for IPDL reactions.

protocols. Since we assume that all function symbols will be interpreted by functions computable in poly-time, our symbolic size for expressions simply counts the number of variables and function applications present.

$$\begin{aligned}
|x| &:= 1 \\
|\checkmark| &:= 0 \\
|\mathsf{true}| &:= 1 \\
|\mathsf{false}| &:= 1 \\
|\mathsf{f}\ e| &:= |e| + 1 \\
|(e_1, e_2)| &:= |e_1| + |e_2| \\
|\mathsf{fst}_{\sigma\times\tau}\ e| &:= |e| \\
|\mathsf{snd}_{\sigma\times\tau}\ e| &:= |e|
\end{aligned}$$

For distributions, we follow a similar principle: we assume that all distribution symbols will be interpreted by probabilistic functions approximately computable in poly-time.

$$|\mathsf{d}\ e| := |e| + 1$$

For reactions, we sum up the symbolic sizes of all expressions and distributions occurring inside the reaction, with

the exception of the conditional: here we pick the size of the larger branch and add it to the size of the condition.

$$\begin{aligned} |\mathsf{ret}\ e| &:= |e| \\ |\mathsf{samp}\ d| &:= |d| \\ |\mathsf{read}\ c| &:= 1 \\ |\mathsf{if}\ e\ \mathsf{then}\ R_1\ \mathsf{else}\ R_2| &:= |e| + \mathsf{max}\,(|R_1|, |R_2|) \\ |x : \sigma \leftarrow R;\ S| &:= |R| + |S| \end{aligned}$$

Since protocols in IPDL are finite networks of channels that do not contain recursion, the size of a protocol is simply the sum of the symbolic sizes of all reactions occurring in the protocol.

$$\begin{aligned} |0| &:= 0 \\ |o := R| &:= |R| \\ |P \mid\mid Q| &:= |P| + |Q| \\ |\mathsf{new}\ o : \tau\ \mathsf{in}\ P| &:= |P| \end{aligned}$$

Figure 13 shows the rules for the approximate equality of IPDL protocols; crucially, rule STRICT allows us to descend to the exact half of the proof system. Whenever we need to make the ambient theory with approximate axioms $\Delta^1 \vdash P^1 \approx Q^1 : I^1 \to O^1, \ldots, \Delta^n \vdash P^n \approx Q^n : I^n \to O^n$ explicit, we write the approximate equality judgement as $\Delta^1 \vdash P^1 \approx Q^1 : I^1 \to O^1, \ldots, \Delta^n \vdash P^n \approx Q^n : I^n \to O^n \Rightarrow \Delta \vdash P \approx Q : I \to O$ width k length l.

For the asymptotic equality of IPDL protocols, we assume a finite set $\mathbb{T}_\approx$ of *axiom families* of the form $\{\Delta_\lambda \vdash P_\lambda \approx Q_\lambda : I_\lambda \to O_\lambda\}_{\lambda \in \mathbb{N}}$. In this setting, the asymptotic equivalence of two protocol families $\{\Delta_\lambda \vdash P_\lambda : I_\lambda \to O_\lambda\}_{\lambda \in \mathbb{N}}$ and $\{\Delta_\lambda \vdash Q_\lambda : I_\lambda \to O_\lambda\}_{\lambda \in \mathbb{N}}$ with pointwise-identical typing judgements takes the form of the judgement $\mathbb{T}_\approx \Rightarrow \{\Delta_\lambda \vdash P_\lambda \approx Q_\lambda : I_\lambda \to O_\lambda\}_{\lambda \in \mathbb{N}}$, see Figure 14.

Specifically, for any fixed λ we obtain an approximate theory by selecting from each axiom family in $\mathbb{T}_\approx$ the axiom corresponding to λ. Similarly, from each of the two protocol families we select the protocol corresponding to λ, which gives us two concrete protocols to equate approximately. We recall that an approximate equality judgement is tagged by a pair of parameters k and l. Letting $\lambda \in \mathbb{N}$ vary thus gives us two functions k_λ and l_λ, and we require that these be bounded by a polynomial. We can summarize the asymptotic judgement as saying that the protocol families are pointwise approximately equal, and both the width and length of the derivation, as well as the number of input and output channels are bounded by a polynomial in λ.

Whenever we need to make the underlying exact theory $\mathbb{T}$ explicit, we write the asymptotic equality judgement as $\mathbb{T}; \mathbb{T}_\approx \Rightarrow \{\Delta_\lambda \vdash P_\lambda \approx Q_\lambda : I_\lambda \to O_\lambda\}_{\lambda \in \mathbb{N}}$.

2 Maude Formalization

2.1 Maude

Maude [**?**] is a high-level declarative language and a high-performance logical framework supporting both equational and rewriting logic computation for a wide range of applications. Maude features several kinds of modules:

- *functional modules*, which are theories (with an initial model semantics) in membership equational logic that allow definitions of data types and operations on them, via multiple sorts, subsort relations between them, equations between terms, and assertions of membership of a term to a sort,
- *system modules*, which are theories in rewriting logic that extend functional modules with definitions of rewrite rules, representing transitions between states, and
- *strategy modules*, which control the way the rewriting rules are applied, by means of strategy combinators, such as concatenation, iterations and others.

We now present the features of the Maude language that we make use of in formalizing IPDL. Maude functional modules are introduced with the syntax `fmod NAME is ... endfm`. In a functional module we can declare sorts, using the keyword `sort`, state that two sorts are in the subsort relation, written `subsort s1 < s2`, declare operations on the sorts, using `op f : s1 ... sk -> s` for an operation `f` with argument sorts `s1 ... sk` and result sort

s. Moreover, operations may have attributes, written in square brackets after their declarations, like comm for commutativity or assoc for associativity. In Maude, terms are rewritten to a normal form modulo the declared attributes and the equations of defined operations. More precisely, equations are used as equational rules: instances of the left-hand side pattern that match subterms of a term are replaces with the corresponding instances of the right-hand side. The process is called term rewriting and the result of simplifying a term by complete application of equational rules is called its normal form. We can control which operations will appear in these ground forms by adding the attribute ctor to them. An operation that is not a constructor of a sort is regarded as defined. Equations are introduced by the syntax eq t1 = t2, where t1 and t2 are terms of sorts related via subsorting. We can assert sort membership using the syntax t : s where t is a term and s is a sort. Conditional equations are written ceq t = t' if C1 ∧ ... ∧ Cn where Ci is either an equation or a membership. We may declare variables using the keyword var. Functional modules are assumed to satisfy the executability requirements of confluence, termination, and sort-decreasingness, see details in [?]. The semantics of functional models is given in terms of the initial model whose elements are ground equivalence classes of terms modulo equations.

Rewriting logic extends equational logic by introducing the notion of rewrites corresponding to transitions between states. Unlike equations, rewrites are not symmetric. Maude system modules are introduced with the syntax mod NAME is ... endm. Rules are declared with the syntax rl [label] : t1 => t2. Conditional rules are written with the keyword crl [label] : t => t' if C1 ∧ ... ∧ Cn and their conditions Ci may be equations, memberships or rewrites. Rewrites are not expected to be terminating, confluent or deterministic. Rewrites denote transitions between the elements of the initial model of the functional part of a system module.

Maude strategy modules are introduced with the syntax smod NAME is ... endsm. In addition to declarations allowed in system modules, we can have strategy declarations and definitions. The main strategy combinators are ; for concatenation of strategy expressions, | for alternative, * for iteration of an expression zero or more times, idle for the strategy giving as result its argument, fail for the strategy that gives no result, s1 ? s2 : s3 for the strategy that attempts to run the strategy s1 then, if the run is successful, it runs s2, otherwise it runs s3. Several other derived constructions are also supported, *e.g.*, try s for s ? idle : idle and s1 or-else s2 for s1 ? idle : s2. The match and rewrite operator matchrew restricts the application of a strategy to a specific subterm of the subject term, see details in [?]. Strategies are declared as strat NAME : s1 ... sk @ s ., where s1 ... sk are the sorts of the arguments of the strategy and s is the subject sort to which the strategy is applied. The syntax for definitions is sd NAME(v1,..., vk) := Exp . where vi are variables of sort si and Exp is a strategy expression.

Maude supports module imports, using the keyword protecting, which means that no new elements of an imported sorts may be added and no identification between elements of an imported sorts via equations are allowed. Two more importation modes are supported, but we do not make use of them.

Maude provides several predefined data types. We will use Booleans, natural numbers, lists, sets and maps.

2.2 Syntax

We start with a sort Type for data types, together with constants unit and bool of sort Type and a binary product on the sort Type. Expressions are built over signatures, which are implemented as commutative lists of symbols, where a function or distribution symbol pairs the symbol name with its arity. Signatures are valid if they don't contain multiple occurences of same symbol name. Expressions are then implemented as a sort Expression that includes as a subsort the identifiers, which are provided by the default Maude sort Qid, such that we can use them for variable names. There are constructors for True, False and (). Application is represented as ap f e where ap is a constructor, f is an identifier standing for the name of the function symbol and e is an expression. Moreover we have constructors for pairs and projections on first and second component of a pair. Type contexts are implemented again as commutative lists of typed variables, written x : T, where x is an identifier and T is a type. Expression typing is implemented as a predicate typeOf : Signature TypeContext Expression -> Bool, while we let Maude handle expression equality by only adding the expression equality rules FAST-PAIR, SND-PAIR and PAIR-EXT as axioms, *e.g.*, eq fst pair(E1, E2) = E1 . where E1 E2 : Expression.

Channel sets are simply sets of identifiers, standing for channel names. Channel contexts are commutative lists of typed channel names, written c :: T.

Reactions are introduced by the following constructors of the sort Reaction, following the grammar for reactions. If e is an expression, return e is a reaction. If d is an identifier, standing for the name of a distribution symbol, and e is an expression, samp d < e > is a reaction. If c is a channel name, read c is a reaction. Moreover, we write if e

`then R1 else R2` for branching, if `e` is an expression and `R1, R2` are reactions, and `x : T <- R1 ; R2` for binding, when `x` is an identifier, `T` is a type and `R1, R2` are reactions. Typing of reactions $\Delta;\ \Gamma \vdash R : I \to \tau$ is given by a function `typeOf : Signature ChannelContext TypeContext Set{ChannelName} Reaction -> Type` , with the meaning that we compute the type of a reaction in the context given by a signature, a channel context, a type context and a set of inputs, i.e. `Delta; Gamma ⊢ R : I → T` if and only if `typeOf(Sigma, Delta, Gamma, I, R) = T`, where `Sigma` is current signature. Maude allows us to write the typing judgements in a very similar way to their original formulation, *e.g.*, the typing rule for binding is written as

```
ceq typeOf(Sigma, Delta, Gamma, I, x : T1 <- R1 ; R2) =
    typeOf(Sigma, Delta, Gamma (x : T1), I, R2)
    if typeOf(Sigma, Delta, Gamma, I, R1) == T1 .
```

Protocols also follow the grammar for protocols, using the following constructors for the sort `Protocol`. For the empty protocol we write `emptyProtocol`. If `c` is a channel name and `R` is a reaction, `c ::= R` is a protocol. If `P1, P2` are protocols, so is `P1 || P2`. Finally. `new c : T in P` is a protocol, if `c` is a channel name, `T` is a type and `P` is a protocol. Typing of protocols $\Delta \vdash P : I \to O$ is implemented as a predicate `typeOf : Signature ChannelContext SetChannelName Protocol -> Bool`, with the meaning that the protocol typechecks in the context given by a signature, a channel context and a set of inputs. Note that since the set of outputs can be computed from a protocol, we do not add it as a parameter of the type checking predicate, so we will have that `Delta ⊢ P : I → getOutputs(P)` if and only if `typeOf(Sigma, Delta, I, P)`, where `Sigma` is the current signature and `getOutputs` computes the outputs of `P`. The implementation splits the typechecking into a check that the inputs are valid w.r.t. `Delta` and a recursive function that does the rest of typechecking:

```
eq typeOf(Sigma, Delta, I, A, P) =
    validChanSet I Delta A
    and
    typeOfAux(Sigma, Delta, I, A, P)
```

For example, the typing rule for `new` checks that `c` is new and that `P` typechecks when extending `Delta` with the typed channel `c :: T`:

```
eq typeOfAux(Sigma, Delta, I, new c : T in P) =
       not occurs c Delta
    and typeOfAux(Sigma, Delta (c :: T), I, P) .
```

2.3 Exact equality

At the reaction level, exact equality is given with axioms of the form $\Delta;\ \Gamma \vdash R_1 = R_2 : I \to \tau$. Let us consider the following example:

`(A :: bool) (B :: bool); empty ⊢`
`x : bool <- return True ; if x then read A else read B = read A : {A, B} → bool.`

The proof of this is obtained by applying the TRANS axiom to

`(A :: bool) (B :: bool); empty ⊢`
`x : bool <- return True ; if x then read A else read B =`
`if True then read A else read B : {A, B} → bool`

that we prove by RET-BIND and

`(A :: bool) (B :: bool); empty ⊢ if True then read A else read B = read A : {A, B} → bool`

that we prove by IF-LEFT.

From a practical point of view, it is inconvenient to write this proof in this way, because we have to make explicit all intermediate steps, and this is tedious and error-prone. Instead, we will work with a transition system. Its states are *configurations* containing the context, i.e. the current signature `Sigma, Delta, Gamma, I, T`, and the current reaction: `rConfig(Sigma, Delta, Gamma, R, I, T)`. The transitions in the system are determined by rewrite rules, which are obtained by orienting the axioms of the exact equality calculus from left to right. Since we can apply the SYM axiom, the choice of direction is not important. For example, the IF-LEFT axiom becomes

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

We also employ the Maude strategy language to conveniently write application of the TRANS axiom as rule composition, denoted ;. The proof in the example above becomes

```
srew
 rConfig(emptySig, (A :: bool) (B :: bool), emptyTypeCtx,
        x : bool <- return True ; if x then read A else read B,
        (A, B), bool)
using ret-bind ; if-left .
```

and Maude returns the following result

```
Solution 1
rewrites: 28 in 0ms cpu (0ms real) (~ rewrites/second)
result ReactionConfig:
 rConfig(emptySig, (A :: bool) (B :: bool), emptyTypeCtx,
        read A, (A , B), bool)
```

If the condition of a rule is a rewrite, we will need to explicitly provide a sub-proof for that step as well. For example, the rule CONG-BIND is

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

and we can apply it to rewrite the reaction `x : bool <- if True then read A else read B ; return x` to `x : bool <- read A ; return x` by writing `cong-bind{if-left, idle}`.

The IF-EXT axiom has the particularity that it establishes an equality between reactions where a variable has been substituted with a term. Maude cannot apply this rule, because it cannot do the matching. For this reason, we have omitted this rule and replaced it with several rules that we can prove using IF-EXT. We have also introduced an alpha-renaming rule for convenience, as we can also derive it from the exact equality axioms.

The same principle is applied for exact equality of protocols. This time we rewrite protocol configurations, of the form `pConfig(Sigma, Delta, P, I, O)`. The rules of exact equality for protocols may make use of exact equality of reactions. For example, the CONG-REACT rule is

```
 crl [CONG-REACT] :
    pConfig(Sigma, Delta (c :: T), c ::= R, I, c)
    =>
    pConfig(Sigma, Delta (c :: T), c ::= R', I, c)
    if
```

```
rConfig(Sigma, Delta (c :: T), emptyTypeCtx, R,
       insert(c, I), T)
=>
rConfig(Sigma, Delta (c :: T), emptyTypeCtx, R', I', T)
/\ I' == insert(c, I)
/\ not c in I .
```

2.4 Normal Forms

We work with protocols that start with a list of declarations of internal channels, using `new`, followed by a parallel compositions of channel assignments. The reactions in these assignments can be transformed into a list of binds of the form `x : T <- read c`, called bind-read reactions, followed by a reaction without binds. The list of binds can be regarded as commutative, as two reactions with the same list of binds in different order are equivalent due to the reaction equivalence rule EXCH. Similarly, different order of declarations of internal channels gives equivalent protocols, by using the protocol equivalence rule NEW-EXCH. When writing equivalence proofs, we do not want to make the use of these rules explicit. Instead, we want to be able to apply the rules as though we could freely consider a certain declaration of an internal channel or a certain bind read reaction as the first.

Therefore, we introduce normal forms of reactions and protocols. For reactions, normal forms `nf(L, R, O)` consist of a commutative list `L` of bind-read reactions, a bind-free reaction `R` and a chosen order `O` of the names of the variables occuring in the binds in `L`, given as a list of names. The latter will be used to determine how to turn the normal form of a reaction into a regular reaction. For example, the normal form of

```
'd : bool <- read 'ce ;
'm0 : bool <- read 'in0 ;
'm1 : bool <- read 'in1 ;
'k0 : bool <- read 'key0 ;
'k1 : bool <- read 'key1 ;
 if 'd then return 'k0 else return 'k1
```

is

```
nf(
 ('d : bool <- read 'ce)
 ('m0 : bool <- read 'in0)
 ('m1 : bool <- read 'in1)
 ('k0 : bool <- read 'key0)
 ('k1 : bool <- read 'key1),
 if 'd then return 'k0 else return 'k1,
 'd :: 'm0 :: 'm1 :: 'k0 :: 'k1
)
```

During equivalence proofs, we may obtain in a normal form `nf(L, R, O)` either arbitrary binds in `L` (*e.g.*, by substituting a read from a channel with the reaction assigned to that channel) or reactions `R` that are not bind-free. This will be represented as a pre-normal-form, written `preNF(L, R, O)`, which is a normal form without restrictions on the occuring reactions. If `L` contains a bind that is not a read bind, we will write it as `x1 : T1 <~ R1`. The general strategy will be to transform pre-normal-forms `preNF(L, R, O)` to normal forms using the following steps:

- if `x1 : T1 <~ R1` is in `L` and `R1` is of the form `nf(L2, R2, O2)`, move the inner binds from `L1` at the level of `L`, and simplify the reaction of `x1` to `R2`.
- if `x1 : T1 <~ R1` is in `L` and `R1` is bind-free, rewrite the entire reaction as `preNF(L', x1 : T1 <- R1 ; R, O')`, where `L'` and `O'` are obtained by removing `x1 : T1 <~ R1` and `x1` from `L` and `O`, respectively.
- apply reaction-level axioms to `R` to bring it in the form `L' ; R'`, where `L'` is a list of bind reads and `R'` is bind-free, then move `L'` at the outer level of `L`.

At the level of protocols, normal forms `newNf(L, P, O)` consist of a commutative list `L` of declarations of internal channels, a protocol `P` that does not start with internal channel declarations and again a designated order `O` for the names of internal channels occuring in the declarations in `L`. For example, the normal form of

```
new 'ce : bool in
new 'key0 : bool in
new 'key1 : bool in
new 'flip : bool in
new 'choice : bool in
 (
 ('ce ::= 'f : bool <- read 'flip ;
         'c : bool <- read 'choice ;
         if 'f then
          (if 'c then return False else return True)
         else
          (if 'c then return True else return False)
 )
 || ('msgenc0 ::= 'd : bool <- read 'ce ;
                 'm0 : bool <- read 'in0 ;
                 'm1 : bool <- read 'in1 ;
                 'k0 : bool <- read 'key0 ;
                 'k1 : bool <- read 'key1 ;
                 if 'd then return 'k0 else return 'k1)
  || ('key0 ::= return True)
  || ('key1 ::= return False)
  || ('flip ::= return True)
  || ('choice ::= return False)
)
```

is

```
newNF(
 ('ce : bool) ('key0 : bool) ('key1 : bool)
 ('flip : bool) ('choice : bool),

 ('ce ::= 'f : bool <- read 'flip ;
         'c : bool <- read 'choice ;
         if 'f then
          (if 'c then return False else return True)
         else
          (if 'c then return True else return False)
 )
 || ('msgenc0 ::= 'd : bool <- read 'ce ;
                 'm0 : bool <- read 'in0 ;
                 'm1 : bool <- read 'in1 ;
                 'k0 : bool <- read 'key0 ;
                 'k1 : bool <- read 'key1 ;
                 if 'd then return 'k0 else return 'k1)
  || ('key0 ::= return True)
  || ('key1 ::= return False)
  || ('flip ::= return True)
  || ('choice ::= return False),

  'ce :: 'key0 :: 'key1 :: 'flip :: 'choice
)
```

2.5 Families of protocols

Families of protocols provide a convenient abbreviation for semantically related protocols `P[0]...P[n]`, where the value of `n` is typically not known. The semantical relation translates in the protocols being assigned similar reactions. We illustrate the syntax with the help of an example:

```
(family 'SumCommit 'i (bound (n + 2)) ::=
       (when ('i =T= 0) --> nf(emptyBRList, return False, emptyCNameList)) ;;
       (when ('i =T= (n + 2)) -->
             nf( ('x : bool <- read ('SumCommit [ n + 1 ]))
                 ('f : bool <- read 'LastCommit) ,
                 return (ap 'xor pair('x, 'f)) ,
                 'x :: 'f :: emptyCNameList )
       ) ;;
       (otherwise --> nf(('x : bool <- read ('SumCommit ['i -- 1]))
                         ('c : bool <- read ('Commit ['i -- 1])),
                         return (ap 'xor pair('x, 'c)),
                         'x :: 'c :: emptyCNameList )
       )
   )
```

Here `'i` is an index variable ranging between `0` and `n + 2`. The reaction assigned to the protocol `'SumCommit['i]` is given with alternatives. We allow the bound to be a natural number, an identifier denoting a natural number or an expression involving natural numbers and identifiers. We represent this as a sort `NatTerm` that is a super-sort of `Qid` and `Nat` together with addition, deletion (written `--`) and multiplication on that sort, extending in the expected way the corresponding operations on natural numbers. The conditions occuring in the alternatives are of sort `BoolTerm`, and can be comparisons between `NatTerms` (`=T=, <T, <=T`), user-defined predicates (`apply 'p t`) where `'p` is the name of the predicate and `t` is a `NatTerm` or negation of a `BoolTerm`.

In this new setting, we may introduce a channel directly or via a family of protocols. Channel names, which were identifiers so far, must be extended to indexed identifiers. We implement them as a sort `ChannelName` which includes as a sub-sort the sort of identifiers and has a constructor `_[_] : Qid List{NatTerm} -> ChannelName`, and thus `'c['i 'j]` is an example of a channel name. Taking this into account, we have also extended channel sets and channel contexts to keep track of the bound of a family of protocols. Channel sets are implemented as sets of bounded channel names, which are written `c @ l` , where `c` is an identifier and `l` is a list of bounds for the family named `c`. We use an empty list for a regular protocol, with no indices. Channel contexts are commutative lists of typed bounded channel names, written `c @ l :: T`, where `T` is a type.

We allow families of protocols with two indices as well. We write `family 'F ('i 'j) ((bound m) (uniformBound n))` for a family indexed by `'i` ranging from `0` to `m` and by `'j` ranging from `0` to `n`. We also allow the second bound to vary for each `'i`, but we did not use this in the case studies so far.

The equality calculus must be adapted to the new notation. We have introduced rules that apply the core equality rules over the new notation, with the meaning that the rules are applied in parallel for each index. We need to record the assumptions made about indices, so we extend the protocol configuration with a new component of sort `Set{BoolTerm}`. Moreover, we have a rule for induction proofs. We present here the variant for one index, as the one for two indices is similar. The goal is to rewrite `P || family C q (bound nt1) ::= cases` to `P || family C q (bound nt1) ::= cases'` by induction on the index

```
crl [INDUCTION-when-one] :
    pConfig(Sigma, Delta,
      P || (family C q (bound nt1) ::= cases), I, O, A)
    =>
    pConfig(Sigma, Delta ,
      P || (family C q (bound nt1) ::= cases'), I, O, A)
```

We start with the base case and we must provide a proof that if we assign `C[0]` its corresponding case from `cases` with `q = 0`, we can rewrite the resulting protocol to the protocol that we get by assigning `C[0]` its corresponding case from `cases'` with `q = 0`. We record the assumption `q = 0` in the set of index assumptions `A`. We also need to update the current outputs, by removing the outputs of the family `C` and adding `C[0]`:

```
if
pConfig(Sigma, Delta ,
  P || (projectIndex (family C q (bound nt1) ::= cases) 0 A empty ), I,
  insert( C[0] @ nil, O \ (C @ nt1)),
  insert(q =T= 0, A)
  )
=>
pConfig(Sigma, Delta , P2 , I, O', A')
/\
O' == insert( C[0] @ nil, O \ (C @ nt1))
/\
A' == insert(q =T= 0, A)
/\
P2 == P || (projectIndex (family C q (bound nt1) ::= cases') 0 A empty)
```

The induction step assumes that we have successfully proven the property up to index `'k`, so now we can make use of `family C q (bound 'k) ::= cases'` when proving the property for index `'k + 1`, where `'k` is arbitrary. We record in A the assumption that `'k + 1` must be in bounds. We also need to update the current outputs, by removing the outputs of the family `C` and adding `C[k + 1]` and the outputs of the family `C` with the new bound `k`:

```
/\
pConfig(Sigma, Delta ,
  P || (family C q (bound 'k) ::= cases') ||
  (projectIndex (family C q (bound nt1) ::= cases) ('k ++ 1) A empty), I,
  insert(C @ 'k, insert(C['k ++ 1] @ nil, O \ (C @ nt1))),
  insert('k ++ 1 <=T nt1, A)
  )
  =>
pConfig(Sigma, Delta, P3, I, O'', A'')
/\
O'' = insert(C @ 'k, insert(C['k ++ 1] @ nil, O \ (C @ nt1)))
/\
A'' == insert('k ++ 1 <=T nt1, A)
/\
P3 == (
       P || (family C q (bound 'k) ::= cases') ||
       (projectIndex (family C q (bound nt1) ::= cases') ('k ++ 1) A empty)
     )
  [nonexec] .
```

A strategy will call the induction rule using

```
INDUCTION-when-one[
       C:Qid <- Q,
       cases':Cases <- 'the cases that we want to get after the induction proof'
      ]
      { 'proof of induction base',
        'proof of induction step'
      }
```

2.6 Strategies

We now come to the strategies that will appear in proofs. It is possible that some of them will make use of other substrategies, but as these will not be in use, we refrain from including them here. To ease presentation, we group strategies by the main core rule that is applied. They may have several forms to be applied in different contexts *e.g.*, channels, families, groups of families, cases.

The rules REFL, TRANS, AXIOM and EMBED are not explicitly applied, as they are implied by the properties of rewrite relation in Maude. The rule SYM does not require the use of a strategy, and in order to apply it we must specify explicitly the protocol that we rewrite from. More precisely, if the current protocol is `P`, we write `SYM[P1:Protocol <- P']{proof}` where `proof` is an exact equality proof rewriting `P'` to `P`. The rule INPUT-UNUSED is embedded in the application of other rules, in the way the conditions on inputs are given. The rules CONG-REACT, CONG-NEW and CONG-COMP are applied inside the strategy definition, and their usage is not visible to the user. The rules CONG-COMM and CONG-ASSOC are not applied explicitly, as it suffices to specify the parallel composition in Maude as a commutative and associative operation.

2.6.1 SUBST

Here we have the largest number of variations, because we need rules for substituting a channel in a family, a family in a family taking into account whether they have one or two indices and so on. Since the number of parameters varies, we cannot have a meta-strategy that tries all possible variants. In the future we plan to generate the extra arguments from the context where the rule applies, and thus reduce the arguments of all strategies to the name of the channel/family that gets substituted and the the name of the channel/family where the substitution takes place. Thus, we will be able to introduce a meta-strategy that greatly simplifies substitutions.

We have the following substitution strategies:

- `substNF(C1, C2)` substitutes the channel `C1` in `C2`. Both channels must be in normal form. The strategy also gets the pre-normal form resulting from the substitution to a normal form, as described above.
- `substNFRead(C1,C2)` is a simpler particular case of substitution when the channel `C` that we substitute reads from another channel. Both channels must be in normal form.
- `smart-subst-nf(C1, C2)` tries to apply `substNFRead` and if that fails, applies `substNF`. In the future we plan to plug all substitution strategies under this meta-strategy.
- `substNFFamiliesOne(C1, C2, R)` substitutes in the family `C2` with one index a read from `C1[i]` for some index `i` with the corresponding reaction `R` assigned in the family `C1` to the index `i`. The family `C2` must be in normal form.
- `applyCaseDistSubst(q1, q2, q3, q4, pr)` works under the assumption that we have two groups, `q1, q2`, and `q1` is defined with cases. The rule does a `substNFRead` equivalent for the families `q3, q4` with two indices that come from the first branch of `q1` and from `q2`. The protocol `pr` is then used in a SYM proof to redo the grouping.
- `substChannelFamilyOne(C1, C2)` substitute a channel in a family with one index.
- `applySubstChannelBranch(C1, q2)` substitutes the channel `C1` in the family `C2`, in the first branch of a group defined with cases.
- `applyCaseDistBranch2(q1, q2)` substitutes a channel in a family on the left branch of the right branch of a group.
- `applyBranch2SubstRev(q1, q2, nt, x, T, R)` applies a reverse substitution on the left branch of a family. The parameters `nt, x, T, R` are the index, the name of the bind variable, the type and the reaction of the channel that is reversely substituted.
- `applySubstRevFamily(Q, C2, T)` does a reverse substitution on a branch of a family with cases. The parameter `T` is the type of the reaction that is reversely substituted.
- `substNFReadFamilyOneChannel(C1, C2)` is a `substNFRead` equivalent for a family with one index and a channel.
- `substNFReadFamilyTwoChannel(C1, C2)` is a `substNFRead` equivalent for a family with two indices and a channel.

- `substRevFamilyChannel(Q, C, nt, T)` does the reverse substitution of a family `Q` in a channel `C`. The parameters `nt, T` are the index and the type of the channel that is reversely substituted.
- `substNFFamilyOneChannel(C1, C2, R)` is the `substNF` equivalent for a family with one index and a channel.
- `applySubstNFLeft(q1, q2, R)` applies `substNFFamiliesOne` on the left branch of a family defined with cases.

2.6.2 DROP

- `applyDropNF(C1, C2)` applies the normal form version of DROP.
- `applyDropPreNF(C1, C2)` applies the pre-normal form version of DROP.

2.6.3 ABSORB

- `absorbChannel(C)` applies the new-normal-form version of ABSORB for the channel `C`.
- `absorbFamily(Q)` applies the new-normal-form version of ABSORB for the family `Q`.
- `applyAbsorbReverse(P)` applies the reverse of the ABSORB rule for the protocol `P`.
- `addNewFamilyToGroup(P, Q1, Q2)` adds the family `Q2`, introduced by the protocol `P`, to the group `Q1`.
- `applyCaseDistAbsorb(q1, q2, q3, pr)` operates under the assumption that the current protocol is of the form `family q2 ::= P || family q1 ::= when cond1 --> P1 ;; otherwise --> P2` and applies the ABSORB rule on the protocol `P || P1` then it reconstructs the original shape of the current protocol.

2.6.4 FOLD

- `foldNF(C1, C2)` applies the normal form version of the FOLD rule, when the channel `C1` gets folded in the channel `C2`.
- `foldNFPre(C1, C2)` applies the pre-normal form version of the FOLD rule, when the channel `C1` gets folded in the channel `C2`.
- `foldNFFamily(Q1, Q2)` applies the normal form version of the FOLD rule, when the family `Q1` gets folded in the family `Q2`.

2.6.5 READ-INSIDE-IF

This is a rule derived from IF-EXT and allows us to rewrite `x :  T1 <- read i ; if M then R1 else R2` to `if M then x :  T1 <- read i ; R1 else x :  T1 <- read i ; R2`.

- `applyReadInsideIfPre(C)` applies READ-INSIDE-IF to a protocol in new-normal-form.

2.6.6 Purely syntactic transformations

Under this heading we group a number of strategy that change only the shape of a protocol. The rules that apply are derivable from the core rules.

- `applyAddToGroup(Q1, Q2)` moves the family `Q2` inside the group `Q1`.
- `changeOrder(C, ql)` changes the specified order of reads in a normal or pre-normal form. `C` is the name of the channel that is assigned the (pre-)normal form and `ql` is the new order, given as a list of variable names.
- `applyReorderNF(Q, ql)`: on the first branch of the family `Q`, changes the order in the normal form as specified in `ql`.
- `nf2PreNF(C)` turn the normal form assigned to the channel `C` to a pre-normal form

- `applyGroupFamilies(Q1, Q2)` composes the families `Q1, Q2` to a new family `'Comp[Q1 Q2]` that assigns to each index `i` the protocol `(Q1[i] ::= R1) || (Q2[i] ::= R2` where `R1, R2` are the reactions assigned to the index `i` by `Q1, Q2`. This transformation is needed for some induction proofs.
- `applyUngroupFamilies(Q1, Q2)` is the reverse transformation of the previous strategy.
- `moveProtocolUnderNewNF` if the current protocol is the parallel composition of a protocol `P` with a new-normal-form, move `P` inside the new-normal-form.
- `applyDeleteEmptyNF(Q)` if the new-normal-form assigned to the group `Q` has no new declarations, keeps only its protocol.
- `applyDropName(Q)` removes the group name `Q`.
- `applyCombine(Q)` if the group `Q` is defined using cases, removes the group name and moves the cases inside the families of the group.
- `applyAlphaNFPr(C, QL)` does an alpha-renaming of the bind variables of a normal form assigned to the channel `C`. `QL` specifies the renaming.
- `applyBranch2Alpha(q1, QL)` does an alpha-renaming on the otherwise branch of a family `q1`. `QL` specifies the renaming.
- `moveBindInPre(C, Q)` if the channel `C` is assigned a normal form, move the bind assigned to the variable `Q` in the reaction of the normal form, and turn the normal form to a pre-normal form.
- `applyBranch2MoveReads(q1, ql)` moves the reads specified by the list `ql` from bind list of a normal form to the reaction of the normal form on the left branch of a family `q1`.
- `moveReadsToRFamily(C,cnl)` move the reads specified in `cnl` from the bind list of a normal form to the reaction of the normal form assigned to the family `C`.

2.7 A simple example

In our simplest example, the IPDL *Hello World* analogue, Alice receives a Boolean message, encodes it by xor-ing it with a randomly generated Boolean, and leaks the encoding to the adversary. We show that this is equal to leaking a randomly generated ciphertext.

Formally, our signature consists of two symbols: $\oplus : \mathsf{Bool} \times \mathsf{Bool} \to \mathsf{Bool}$ for the Boolean sum, and $\mathsf{flip} : 1 \twoheadrightarrow \mathsf{Bool}$ for the uniform distribution on Booleans. We write $x \oplus y$ in place of $\oplus\,(x, y)$.

2.7.1 The Assumptions

Our single axiom states that flip is invariant under xor-ing with a fixed Boolean:

- $\cdot;\ x : \mathsf{Bool} \vdash (y \leftarrow \mathsf{flip};\ \mathsf{ret}\ x \oplus y) = \mathsf{samp}\ \mathsf{flip} : \varnothing \to \mathsf{Bool}$

This is indeed the case if (and only if) flip is uniform.

2.7.2 The Ideal Functionality

Upon receiving the input message, the ideal functionality generates a random ciphertext on an internal channel Ctxt and leaks its value to the adversary:

- $\mathsf{Ctxt} := m \leftarrow \mathsf{In}; \mathsf{samp}\ \mathsf{flip}$
- $\mathsf{LeakCtxt}^{\mathsf{id}}_{\mathsf{adv}} := \mathsf{read}\ \mathsf{Ctxt}$

2.7.3 The Real Protocol

In the real protocol, Alice generates a random Boolean key on an internal channel Key, constructs the ciphertext by xor-ing the input message with the key, and leaks the resulting ciphertext to the adversary:

- $\mathsf{Key} := \mathsf{samp}\ \mathsf{flip}$
- $\mathsf{Ctxt} := m \leftarrow \mathsf{In};\ k \leftarrow \mathsf{Key};\ \mathsf{ret}\ m \oplus k$
- $\mathsf{LeakCtxt}^{\mathsf{Alice}}_{\mathsf{adv}} := \mathsf{read}\ \mathsf{Ctxt}$

2.7.4 The Simulator

The simulator mediates between the two leakage channels $\mathsf{LeakCtxt}^{\mathsf{id}}_{\mathsf{adv}}$ and $\mathsf{LeakCtxt}^{\mathsf{Alice}}_{\mathsf{adv}}$ by forwarding the former to the latter:

- $\mathsf{LeakCtxt}^{\mathsf{Alice}}_{\mathsf{adv}} := \mathsf{read}\ \mathsf{LeakCtxt}^{\mathsf{id}}_{\mathsf{adv}}$

2.7.5 Real = Ideal + Sim

On the left-hand side of the above equality we have the real protocol. On the right-hand side, we have the composition of the ideal functionality with the simulator, followed by the hiding of the channel $\mathsf{LeakCtxt}^{\mathsf{id}}_{\mathsf{adv}}$. The two protocols now have identical inputs (the channel In) as well as outputs (the channel $\mathsf{LeakCtxt}^{\mathsf{Alice}}_{\mathsf{adv}}$).

We now simplify both protocols so that they have the same internal structure. On the left-hand side, we fold the internal channel

- $\mathsf{Key} := \mathsf{samp}\ \mathsf{flip}$

into the channel

- $\mathsf{Ctxt} := m \leftarrow \mathsf{In};\ k \leftarrow \mathsf{Key};\ \mathsf{ret}\ m \oplus k$,

yielding

- $\mathsf{Ctxt} := m \leftarrow \mathsf{In};\ k \leftarrow \mathsf{flip};\ \mathsf{ret}\ m \oplus k$

and on the right-hand side we fold the internal channel

- $\mathsf{LeakCtxt}^{\mathsf{id}}_{\mathsf{adv}} := \mathsf{read}\ \mathsf{Ctxt}$

into the channel

- $\mathsf{LeakCtxt}^{\mathsf{Alice}}_{\mathsf{adv}} := \mathsf{read}\ \mathsf{LeakCtxt}^{\mathsf{id}}_{\mathsf{adv}}$,

yielding

- $\mathsf{LeakCtxt}^{\mathsf{Alice}}_{\mathsf{adv}} := \mathsf{read}\ \mathsf{Ctxt}$.

The two protocols now both have an internal channel Ctxt and an output channel $\mathsf{LeakCtxt}^{\mathsf{Alice}}_{\mathsf{adv}}$.

To finish the proof, we fold the internal channel into the output channel in both protocols, yielding the two single-reaction protocols

- $\mathsf{LeakCtxt}^{\mathsf{Alice}}_{\mathsf{adv}} := m \leftarrow \mathsf{In};\ k \leftarrow \mathsf{flip};\ \mathsf{ret}\ m \oplus k$, and
- $\mathsf{LeakCtxt}^{\mathsf{Alice}}_{\mathsf{adv}} := m \leftarrow \mathsf{In};\ \mathsf{samp}\ \mathsf{flip}$

The equality between these two now follows immediately from our axiom, and we are done.

2.7.6 Maude implementation

Assume we work in the file `helloWorld.maude` placed in the `lib` folder of the IPDL-Maude repository. We start by importing the strategies and starting a new module that extends `APPROX-EQUALITY`, which provides both exact and approximate equality

```
load ../src/strategies

mod HELLO-WORLD is
 protecting APPROX-EQUALITY .
```

We have to define the signature. Our example works with booleans, so we will not introduce new datatypes. When needed, we define them as new constants of sort `Type`.

We must introduce a function symbol for $\oplus$ and a distribution symbol for flip:

```
op xorF : -> SigElem .
eq xorF = 'xor : (bool * bool) ~> bool .

op flipF : -> SigElem .
eq flipF = 'flip : unit ~>> bool .

op sig : -> Signature .
eq sig = xorF flipF .
```

We then write the protocol resulting from composing the ideal functionality with the simulator followed by hiding the channel `'LeakCtxt_id_adv` and the protocol `real`:

```
 op idealPlusSim : -> Protocol .
 eq idealPlusSim =
    new 'Ctxt : bool in
    new 'LeakCtxt_id_adv : bool in
    (
    ('LeakCtxt_id_adv ::=
       nf(('c : bool <- read 'Ctxt),
          return 'c,
          'c :: emptyCNameList )
    )
    ||
    ('LeakCtxt_Alice_adv ::=
       nf('c : bool <- read 'LeakCtxt_id_adv,
          return 'c,
          'c :: emptyCNameList)
    )
    ||
    ('Ctxt ::=
       nf( 'm : bool <- read 'In,
           samp ('flip < () >),
           'm :: emptyCNameList)
    )
    )
 .

  op real : -> Protocol .
  eq real =
    new 'Key : bool in
    new 'Ctxt : bool in
    (
```

```
    ('Key ::= samp ('flip < () >))
    ||
    ('Ctxt ::=
      nf( ('m : bool <- read 'In)
          'k : bool <- read 'Key,
          return (ap 'xor pair('m, 'k)),
         'm :: 'k :: emptyCNameList
            )
    )
    ||
    ('LeakCtxt_Alice_adv ::=
      nf( 'c : bool <- read 'Ctxt ,
          return 'c,
         'c :: emptyCNameList )
    )
   )

  .
```

We then close the `HELLO-WORLD` module and open a new module, `EXECUTE`, importing `HELLO-WORLD` and `STRATEGIES`, where we add the assumptions and typically strategies for using them in proofs

```
smod EXECUTE is
  protecting STRATEGIES .
  protecting HELLO-WORLD .
```

The assumption is introduced at the level of reactions

```
rl [assumption] :
    rConfig(Sigma, Delta, Gamma (x : bool),
           y : bool <- samp flip ;
           return (ap 'xor pair(x, y)), I, A, bool
           )
    =>
    rConfig(Sigma, Delta , Gamma (x : bool),
           samp flip, I, A, bool)
  .
```

and we will want to apply it in the main reaction of a pre-normal form, so we need the following strategy:

```
strat applyAssumption : ChannelName @ ProtocolConfig .
  sd applyAssumption(cn) :=
   match pConf s.t. startsWithNew pConf
    ? CONG-NEW-NF{applyAssumption(cn)}
    : matchrew pConf s.t. pConfig(Sigma, Delta, P, I, O, A) := pConf by pConf
      using CONG-REACT[o:ChannelName <- cn]
              { cong-pre-nf{assumption} ;
                try (pre2Nf)}
  .
```

which states, read in reverse order, that we apply CONG-PRE-NF with the assumption as a sub-proof to the reaction `R` that is assigned to the channel `cn` (thus requiring to apply CONG-REACT) that is inside a new normal form, and then we must apply `cong-new-nf`. Moreover, `pre2Nf` converts the pre-normal form to a normal form, as now we no longer have binds in the reaction of the pre-normal form.

We now get to writing the proof. The initial configuration is

```
pConfig(sig,
       ('In @ nil :: bool)
```

```
        ('LeakCtxt_Alice_adv @ nil :: bool),
        idealPlusSim,
        'In @ nil,
        getOutputs(idealPlusSim),
        empty)
```

where

- `sig` is the signature defined above,
- the channel context contains the input channel `'In` and the output channel `'LeakCtxt_Alice_adv`. Both are of type `bool` and they have no indices,
- the protocol that we want to rewrite,
- the set of its inputs, here we have just one input channel,
- the set of its outputs, which can be computed, so it is more convenient to call the function `getOutputs` than to enumerate all inputs,
- since we have no indices, the set of assumptions about them is empty.

We first turn the protocol in new normal form, then we do the two folds:

```
srew [1]
pConfig(sig,
        ('In @ nil :: bool)
        ('LeakCtxt_Alice_adv @ nil :: bool),
        idealPlusSim,
        'In @ nil,
        getOutputs(idealPlusSim),
        empty)
using
      sugar-newNF
    ; foldNF('LeakCtxt_id_adv, 'LeakCtxt_Alice_adv)
    ; foldNF('Ctxt, 'LeakCtxt_Alice_adv)
.
```

Similarly, for `real`, we turn the protocol in new normal form, we do the folds and then we apply the assumption:

```
srew [1]
 pConfig(sig,
        ('In @ nil :: bool)
        ('LeakCtxt_Alice_adv @ nil :: bool),
        real,
        'In @ nil,
        getOutputs(real),
        empty)
using
      sugar-newNF
    ; foldNF('Key, 'Ctxt)
    ; foldNF('Ctxt, 'LeakCtxt_Alice_adv)
    ; applyAssumption('LeakCtxt_Alice_adv)
.
```

In both cases we get the same result, so we now can combine the two proofs in one, using SYM:

```
srew [1]
 pConfig(sig,
        ('In @ nil :: bool)
```

```
        ('LeakCtxt_Alice_adv @ nil :: bool),
        real,
        'In @ nil,
        getOutputs(real),
        empty)
using
      sugar-newNF
    ; foldNF('Key, 'Ctxt)
    ; foldNF('Ctxt, 'LeakCtxt_Alice_adv)
    ; applyAssumption('LeakCtxt_Alice_adv)
    ; SYM[P1:Protocol <- idealPlusSim]{
      sugar-newNF
    ; foldNF('LeakCtxt_id_adv, 'LeakCtxt_Alice_adv)
    ; foldNF('Ctxt, 'LeakCtxt_Alice_adv)
    }
.
```

When running the proof in Maude, we get `idealPlusSim` in the protocol in the resulting `pConfig`, as expected. Maude also reports on the number of rewrites and the time spent doing the proof:´

```
$ maude lib/helloWorld.maude
                    \||||||||||||||||||/
                  --- Welcome to Maude ---
                    /||||||||||||||||||\
           Maude 3.2.1 built: Feb 21 2022 18:21:17
            Copyright 1997-2022 SRI International
                 Tue Mar 14 04:52:45 2023
==========================================
srewrite [1] in EXECUTE : pConfig(sig, ('In @ nil :: bool)
    'LeakCtxt_Alice_adv @ nil :: bool, real, 'In @ nil, getOutputs(real),
    empty) using sugar-newNF ; foldNF('Key, 'Ctxt) ; foldNF('Ctxt,
    'LeakCtxt_Alice_adv) ; applyAssumption('LeakCtxt_Alice_adv) ; SYM[
    P1:Protocol <- idealPlusSim]{sugar-newNF ; foldNF('LeakCtxt_id_adv,
    'LeakCtxt_Alice_adv) ; foldNF('Ctxt, 'LeakCtxt_Alice_adv)} .

Solution 1
rewrites: 516 in 0ms cpu (1ms real) (~ rewrites/second)
result ProtocolConfig: pConfig(('xor : bool * bool ~> bool) 'flip : unit ~>>
    bool, ('In @ nil :: bool) 'LeakCtxt_Alice_adv @ nil :: bool, new 'Ctxt :
    bool in new 'LeakCtxt_id_adv : bool in ('Ctxt ::= nf('m : bool <- read
    'In, samp ('flip < () >), 'm :: emptyCNameList)) || ('LeakCtxt_Alice_adv
    ::= nf('c : bool <- read 'LeakCtxt_id_adv, return 'c, 'c ::
    emptyCNameList)) || 'LeakCtxt_id_adv ::= nf('c : bool <- read 'Ctxt,
    return 'c, 'c :: emptyCNameList), 'In @ nil, 'LeakCtxt_Alice_adv @ nil,
    empty)
```

2.8 Approximate equality

We start by defining wrappers for width and length

```
  sort Width .
  sort Length .

  op width_ : Nat -> Width [ctor] .
  op length_ : Nat -> Length [ctor] .
```

and the measure functions

```
op |_| : Protocol -> Nat .
op |_| : Reaction -> Nat .
op |_| : Expression -> Nat .
```

The configurations `aConfig(Sigma, Delta, P, I, O, A, w, l)` for approximate equality extend protocol configurations with fields for width and length. The main idea is that we set these at 0 at the start of a proof, and we keep track of their modifications with the help of the approximate equality rules. The STRICT rule allows us to apply exact equality calculus without modifying the width and the length

```
crl [STRICT] :
  aConfig(Sigma, Delta, P, I, O, A, w, l)
  =>
  aConfig(Sigma, Delta, Q, I, O, A, w, l)
if
 pConfig(Sigma, Delta, P, I, O, A)
 =>
 pConfig(Sigma, Delta, Q, I, O, A)
.
```

The TRANS rule

```
crl [TRANS] :
  aConfig(Sigma, Delta, P, I, O, A, width nw, length nl)
  =>
  aConfig(Sigma, Delta, P2, I, O, A, width (nw + nw1 + nw2), length (nl + defMax(nl1, nl2)))
if
  aConfig(Sigma, Delta, P, I, O, A, width 0, length 0)
  =>
  aConfig(Sigma, Delta, P1, I, O, A, width nw1, length nl1)
  /\
  aConfig(Sigma, Delta, P1, I, O, A, width 0, length 0)
  =>
  aConfig(Sigma, Delta, P2, I, O, A, width nw2, length nl2)
.
```

adds to the current value of width the values computed in the sub-proofs and to the current value of length the maximum of the two lengths computed in the subproofs.

Approximate equality axioms combine the rules AXIOM and INPUT-UNUSED: if we rewrite a `aConfig` whose width is `w` and whose length is `l`, we get a `aConfig` whose width is `w + 1` and whose length is `l + | I  I'|` where `I` is the set of inputs of the configuration and `I'` is the set of inputs used by the protocol we want to rewrite with the axiom.

Strategies for applying an approximate equality axiom are of the form

```
strat S : Param @ ApproxEqConfig .
sd S(X) :=
   match aConf s.t. aConfStartsWithNew aConf
   ? CONG-NEW-NF-APPROX{S(X)}
   : matchrew aConf s.t.
      aConfig(Sigma, Delta, P, I, O, A, width w, length l) := aConf
     by aConf
     using
     CONG-COMP-APPROX{
        axiom(X)
     }
   .
```

Proofs will typically consist of a composition, using TRANS rule, of several STRICT steps with a number of applications of the axioms, using their corresponding strategies.

$$\boxed{\Delta \vdash P = Q : I \to O}$$

$$\frac{\Delta \vdash P : I \to O}{\Delta \vdash P = P : I \to O}\ \text{REFL} \qquad \frac{\Delta \vdash P_1 = P_2 : I \to O}{\Delta \vdash P_2 = P_1 : I \to O}\ \text{SYM}$$

$$\frac{\Delta \vdash P_1 = P_2 : I \to O \qquad \Delta \vdash P_2 = P_3 : I \to O}{\Delta \vdash P_1 = P_3 : I \to O}\ \text{TRANS} \qquad \frac{\Delta \vdash P = Q : I \to O\ \mathsf{axiom}}{\Delta \vdash P = Q : I \to O}\ \text{AXIOM}$$

$$\frac{i \notin I \cup O \qquad \Delta \vdash P_1 = P_2 : I \to O}{\Delta \vdash P_1 = P_2 : I \cup \{i\} \to O}\ \text{INPUT-UNUSED} \qquad \frac{\phi : \Delta_1 \to \Delta_2 \qquad \Delta_2 \vdash P = Q : I \to O}{\Delta_1 \vdash \phi^\star(P) = \phi^\star(Q) : \phi^\star(I) \to \phi^\star(O)}\ \text{EMBED}$$

$$\frac{o : \tau \in \Delta \qquad o \notin I \qquad \Delta;\ \cdot \vdash R = R' : I \cup \{o\} \to \tau}{\Delta \vdash (o := R) = (o := R') : I \to \{o\}}\ \text{CONG-REACT}$$

$$\frac{\Delta \vdash P = P' : I \cup O_2 \to O_1 \qquad \Delta \vdash Q : I \cup O_1 \to O_2}{\Delta \vdash P \mid\mid Q = P' \mid\mid Q : I \to O_1 \cup O_2}\ \text{CONG-COMP-LEFT}$$

$$\frac{\Delta, o : \tau \vdash P = P' : I \to O \cup \{o\}}{\Delta \vdash (\mathsf{new}\ o : \tau\ \mathsf{in}\ P) = (\mathsf{new}\ o : \tau\ \mathsf{in}\ P') : I \to O}\ \text{CONG-NEW}$$

$$\frac{\Delta \vdash P_1 : I \cup O_2 \to O_1 \qquad \Delta \vdash P_2 : I \cup O_1 \to O_2}{\Delta \vdash P_1 \mid\mid P_2 = P_2 \mid\mid P_1 : I \to O_1 \cup O_2}\ \text{COMP-COMM}$$

$$\frac{\Delta \vdash P_1 : I \cup O_2 \cup O_3 \to O_1 \qquad \Delta \vdash P_2 : I \cup O_1 \cup O_3 \to O_2 \qquad \Delta \vdash P_3 : I \cup O_1 \cup O_2 \to O_3}{\Delta \vdash (P_1 \mid\mid P_2) \mid\mid P_3 = P_1 \mid\mid (P_2 \mid\mid P_3) : I \to O_1 \cup O_2 \cup O_3}\ \text{COMP-ASSOC}$$

$$\frac{\Delta, o_1 : \tau_1, o_2 : \tau_2 \vdash P : I \to O \cup \{o_1, o_2\}}{\Delta \vdash \big(\mathsf{new}\ o_1 : \tau_1\ \mathsf{in}\ \mathsf{new}\ o_2 : \tau_2\ \mathsf{in}\ P\big) = \big(\mathsf{new}\ o_2 : \tau_2\ \mathsf{in}\ \mathsf{new}\ o_1 : \tau_1\ \mathsf{in}\ P\big) : I \to O}\ \text{NEW-EXCH}$$

$$\frac{\Delta \vdash P : I \cup O_2 \to O_1 \qquad \Delta, o : \tau \vdash Q : I \cup O_1 \to O_2 \cup \{o\}}{\Delta \vdash P \mid\mid (\mathsf{new}\ o : \tau\ \mathsf{in}\ Q) = \mathsf{new}\ o : \tau\ \mathsf{in}\ (P \mid\mid Q) : I \to O_1 \cup O_2}\ \text{COMP-NEW}$$

$$\frac{\Delta \vdash P : I \to O \qquad \Delta \vdash Q : I \cup O \to \varnothing}{\Delta \vdash P \mid\mid Q = P : I \to O}\ \text{ABSORB-LEFT}$$

Figure 11: Exact equality for IPDL protocols. Additional rules are given in Figure 12.

$$\boxed{\Delta \vdash P = Q : I \to O}$$

$$\frac{o : \tau \in \Delta \qquad o \notin I \qquad \Delta;\ \cdot \vdash R : I \cup \{o\} \to \tau}{\Delta \vdash (o := x : \tau \leftarrow \mathsf{read}\ o;\ R) = (o := \mathsf{read}\ o) : I \to \{o\}}\ \textsc{diverge}$$

$$\frac{o \notin I \qquad b \in I \qquad b : \mathsf{Bool}, o : \tau \in \Delta \qquad \Delta;\ \cdot \vdash S_1 : I \cup \{o\} \to \tau \qquad \Delta;\ \cdot \vdash S_2 : I \cup \{o\} \to \tau}{\begin{array}{c}\Delta \vdash \big(\mathsf{new}\ l : \tau\ \mathsf{in}\ o := x : \mathsf{Bool} \leftarrow \mathsf{read}\ b;\ \mathsf{if}\ x\ \mathsf{then}\ \mathsf{read}\ l\ \mathsf{else}\ S_2 \mid\mid l := S_1\big) = \\ \big(o := x : \mathsf{Bool} \leftarrow \mathsf{read}\ b;\ \mathsf{if}\ x\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\big) : I \to \{o\}\end{array}}\ \textsc{fold-if-left}$$

$$\frac{o \notin I \qquad b \in I \qquad b : \mathsf{Bool}, o : \tau \in \Delta \qquad \Delta;\ \cdot \vdash S_1 : I \cup \{o\} \to \tau \qquad \Delta;\ \cdot \vdash S_2 : I \cup \{o\} \to \tau}{\begin{array}{c}\Delta \vdash \big(\mathsf{new}\ r : \tau\ \mathsf{in}\ o := x : \mathsf{Bool} \leftarrow \mathsf{read}\ b;\ \mathsf{if}\ x\ \mathsf{then}\ S_1\ \mathsf{else}\ \mathsf{read}\ r \mid\mid r := S_2\big) = \\ \big(o := x : \mathsf{Bool} \leftarrow \mathsf{read}\ b;\ \mathsf{if}\ x\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\big) : I \to \{o\}\end{array}}\ \textsc{fold-if-right}$$

$$\frac{o \notin I \qquad o : \tau_2 \in \Delta \qquad \Delta;\ \cdot \vdash R_1 : I \cup \{o\} \to \tau_1 \qquad \Delta;\ x : \tau_1 \vdash R_2 : I \cup \{o\} \to \tau_2}{\Delta \vdash \big(\mathsf{new}\ c : \sigma\ \mathsf{in}\ o := x : \tau_1 \leftarrow \mathsf{read}\ c;\ R_2 \mid\mid c := R_1\big) = (o := x : \tau_1 \leftarrow R_1;\ R_2) : I \to \{o\}}\ \textsc{fold-bind}$$

$$\frac{\begin{array}{c}o_1 \neq o_2 \\ o_1, o_2 \notin I \qquad o_1 : \tau_1, o_2 : \tau_2 \in \Delta \qquad \Delta;\ \cdot \vdash R_1 : I \cup \{o_1, o_2\} \to \tau_1 \qquad \Delta;\ x_1 : \tau_1 \vdash R_2 : I \cup \{o_1, o_2\} \to \tau_2 \\ \Delta;\ \cdot \vdash \big(x_1 : \tau_1 \leftarrow R_1;\ x_1' : \tau_1 \leftarrow R_1;\ \mathsf{ret}\ (x_1, x_1')\big) = \big(x_1 : \tau_1 \leftarrow R_1;\ \mathsf{ret}\ (x_1, x_1)\big) : I \cup \{o_1, o_2\} \to \tau_1 \times \tau_1\end{array}}{\Delta \vdash \big(o_1 := R_1 \mid\mid o_2 := x_1 : \tau_1 \leftarrow \mathsf{read}\ o_1;\ R_2\big) = \big(o_1 := R_1 \mid\mid o_2 := x_1 : \tau_1 \leftarrow R_1;\ R_2\big) : I \to \{o_1, o_2\}}\ \textsc{subst}$$

$$\frac{\begin{array}{c}o_1 \neq o_2 \qquad o_1, o_2 \notin I \qquad o_1 : \tau_1, o_2 : \tau_2 \in \Delta \qquad \Delta;\ \cdot \vdash R_1 : I \cup \{o_1, o_2\} \to \tau_1 \\ \Delta;\ \cdot \vdash R_2 : I \cup \{o_1, o_2\} \to \tau_2 \qquad \Delta;\ \cdot \vdash (x_1 : \tau_1 \leftarrow R_1;\ R_2) = R_2 : I \cup \{o_1, o_2\} \to \tau_2\end{array}}{\Delta \vdash \big(o_1 := R_1 \mid\mid o_2 := x_1 \leftarrow \mathsf{read}\ o_1;\ R_2\big) = (o_1 := R_1 \mid\mid o_2 := R_2) : I \to \{o_1, o_2\}}\ \textsc{drop}$$

Figure 12: Additional rules for exact equality of IPDL protocols. Distinguishing changes of equalities are highlighed in red.

$$\boxed{\Delta \vdash P \approx Q : I \to O \text{ width } k \text{ length } l}$$

$$\frac{\Delta \vdash P = Q : I \to O}{\Delta \vdash P \approx Q : I \to O \text{ width } 0 \text{ length } 0}\ \text{STRICT} \qquad \frac{\Delta \vdash P_1 \approx P_2 : I \to O \text{ width } k \text{ length } l}{\Delta \vdash P_2 \approx P_1 : I \to O \text{ width } k \text{ length } l}\ \text{SYM}$$

$$\frac{\Delta \vdash P_1 \approx P_2 : I \to O \text{ width } k_1 \text{ length } l_1 \qquad \Delta \vdash P_2 \approx P_3 : I \to O \text{ width } k_2 \text{ length } l_2}{\Delta \vdash P_1 \approx P_3 : I \to O \text{ width } k_1 + k_2 \text{ length } \max(l_1, l_2)}\ \text{TRANS}$$

$$\frac{\Delta \vdash P \approx Q : I \to O \text{ axiom}}{\Delta \vdash P \approx Q : I \to O \text{ width } 1 \text{ length } 0}\ \text{AXIOM} \qquad \frac{i \notin I \cup O \qquad \Delta \vdash P \approx Q : I \to O \text{ width } k \text{ length } l}{\Delta \vdash P \approx Q : I \cup \{i\} \to O \text{ width } k \text{ length } l + 1}\ \text{INPUT-UNUSED}$$

$$\frac{\theta : \Delta_1 \to \Delta_2 \qquad \Delta_1 \vdash P \approx Q : I \to O \text{ width } k \text{ length } l}{\Delta_2 \vdash \theta^\star(P) \approx \theta^\star(Q) : \theta^\star(I) \to \theta^\star(O) \text{ width } k \text{ length } l}\ \text{EMBED}$$

$$\frac{\Delta \vdash P \approx P' : I \cup O_2 \to O_1 \text{ width } k \text{ length } l \qquad \Delta \vdash_\Sigma Q : I \cup O_1 \to O_2}{\Delta \vdash P \mid\mid Q \approx P' \mid\mid Q : I \to O_1 \cup O_2 \text{ width } k \text{ length } l + |Q|}\ \text{CONG-COMP-LEFT}$$

$$\frac{\Delta, o : A \vdash P \approx P' : I \to O \cup \{o\} \text{ width } k \text{ length } l}{\Delta \vdash (\text{new } o : A \text{ in } P) \approx (\text{new } o : A \text{ in } P') : I \to O \text{ width } k \text{ length } l}\ \text{CONG-NEW}$$

Figure 13: Approximate equality for IPDL protocols.

$$\boxed{\{\Delta^1_\lambda \vdash P^1_\lambda \approx Q^1_\lambda : I^1_\lambda \to O^1_\lambda\}_{\lambda\in\mathbb{N}}, \ldots, \{\Delta^n_\lambda \vdash P^n_\lambda \approx Q^n_\lambda : I^n_\lambda \to O^n_\lambda\}_{\lambda\in\mathbb{N}} \Rightarrow \{\Delta_\lambda \vdash P_\lambda \approx Q_\lambda : I_\lambda \to O_\lambda\}_{\lambda\in\mathbb{N}}}$$

$$\frac{\begin{array}{c}\forall \lambda, \Delta^1_\lambda \vdash P^1_\lambda \approx Q^1_\lambda : I^1_\lambda \to O^1_\lambda, \ldots, \Delta^n_\lambda \vdash P^n_\lambda \approx Q^n_\lambda : I^n_\lambda \to O^n_\lambda \Rightarrow \Delta_\lambda \vdash P_\lambda \approx Q_\lambda : I_\lambda \to O_\lambda \text{ width } k_\lambda \text{ length } l_\lambda \\ k_\lambda = \mathsf{O}(\mathsf{poly}(\lambda)) \qquad l_\lambda = \mathsf{O}(\mathsf{poly}(\lambda)) \qquad |I_\lambda| = \mathsf{O}(\mathsf{poly}(\lambda)) \qquad |O_\lambda| = \mathsf{O}(\mathsf{poly}(\lambda))\end{array}}{\{\Delta^1_\lambda \vdash P^1_\lambda \approx Q^1_\lambda : I^1_\lambda \to O^1_\lambda\}_{\lambda\in\mathbb{N}}, \ldots, \{\Delta^n_\lambda \vdash P^n_\lambda \approx Q^n_\lambda : I^n_\lambda \to O^n_\lambda\}_{\lambda\in\mathbb{N}} \Rightarrow \{\Delta_\lambda \vdash P_\lambda \approx Q_\lambda : I_\lambda \to O_\lambda\}_{\lambda\in\mathbb{N}}}$$

Figure 14: Asymptotic equivalence for IPDL protocol families.