# Using session types as an effect system

Dominic Orchard and Nobuko Yoshida

Imperial College London

**Abstract**

Side effects are a core part of practical programming. However, they are often hard to reason about, particularly in a concurrent setting. We propose a foundation for reasoning about concurrent side effects using *sessions*. We show that *session types* are expressive enough to encode an *effect system* for stateful processes. This is formalised via an effect-preserving encoding of a simple imperative language with an effect system into the $\pi$-calculus with sessions and session types (into which we encode effect specifications). We demonstrate how this technique can be used to reason about, specify, and control the scope of concurrent side effects.

## 1 Introduction

*Side effects* such as input-output and mutation of memory are important features of practical programming. However, effects are often difficult to reason about due to their implicit impact. Reasoning about effects is even more difficult in a concurrent setting, where interference may cause unintended non-determinism. For example, consider a parallel program: $\mathbf{put}\, x\, ((\mathbf{get}\, x) + 2) \mid \mathbf{put}\, x\, ((\mathbf{get}\, x) + 1)$ where $x$ is a mutable memory cell. Given an initial assignment $x \mapsto 0$, the final value stored at $x$ may be any of 3, 2, or 1 since calls to **get** and **put** may be interleaved.

Many approaches to reasoning, specifying, and controlling the scope of effects have therefore been proposed. Seemingly orthogonally, various approaches for specifying and reasoning about concurrent interactions have also been developed. In this paper, we show that two particular approaches for reasoning about effects and concurrency are in fact *non*-orthogonal; one can be embedded into the other. We show that *session types* [9] for concurrent processes are expressive enough to encode *effect systems* [3] for state. We formalise this ability by embedding/encoding a simple imperative language with an effect system into the $\pi$-calculus with session types: sessions simulate state and session types become effect annotations. Formally, our embedding maps type-and-effect judgements to session type judgements:

$$\Gamma \vdash M : \tau, F \quad \xrightarrow{\textit{embedding}} \quad [\![\Gamma]\!];\, \mathit{res} : ![\![\tau]\!].\mathbf{end},\, \mathit{eff} : [\![F]\!] \vdash [\![M]\!] \tag{1}$$

That is, an expression $M$ of type $\tau$ in context $\Gamma$, performing effects $F$ is mapped to a process $[\![M]\!]$ which sends its result over session channel *res* and simulates effects by interactions $[\![F]\!]$ (defined by an interpretation of the effect annotation) over session channel *eff*.

We start with the traditional encoding of a mutable store into the $\pi$-calculus (Section 2) and show how its session types provide a kind of effect system. Section 2 introduces a simple imperative language, which we embed into the $\pi$-calculus with sessions (sometimes called the *session calculus*) (Section 3). The embedding is shown sound with respect to an equational theory for effects. Section 4 demonstrates how to build upon the encoding to guard against effect interference in a concurrent setting and to safely introduce implicit parallelism.

Our embedding has been formalised in Agda to account for all details and is available at `https://github.com/dorchard/effects-as-sessions` (Appendix B gives a brief description).

The main result of this paper is technical, about the expressive power of the $\pi$-calculus with session primitives and session types. This technical result has a number of possible uses:

[0]Author's version. To appear in the pre-proceedings of PLACES 2015.

- *Effects systems for the $\pi$-calculus*: rather than adding an additional effect system on top of the $\pi$-calculus, we show that existing work on session types can be reused for this purpose.
- *Semantics of concurrency and effects*: our approach provides an intermediate language for the semantics of effects in a concurrent setting. We demonstrate this in Section 4, with a semantics for parallel composition in the source language which avoids race conditions.
- *Compilation*: Related to the above, the session calculus can be used as a typed intermediate language for compilation, where our embedding provides the translation. Section 4 demonstrates an optimisation step where safe implicit parallelism is introduced based on effect information and soundness results of our embedding.

Effect systems have been used before to reason about effects in concurrent programs. For example, Deterministic Parallel Java uses an effect system to check that parallel processes can safely commute without memory races, and otherwise schedules processes to ensure determinism [1]. Our approach allows state effects to be incorporated directly into concurrent protocol descriptions, reusing session types, without requiring interaction between two distinct systems.

# 2 Simulating state with sessions

**2.1 State via processes** A well-known way to implement state in a process algebra is to represent a mutable store as a server-like process (often called a *variable agent*) that offers two modes of interaction ($\mathsf{get}$ and $\mathsf{put}$). In the $\mathsf{get}$ mode, the agent waits to receive a value on its channel which is then "stored"; in the $\mathsf{put}$ it sends the stored value. This can be implemented in the $\pi$-calculus with branching and recursive definitions as follows (Figure 1 describes the syntax), where *Store* is parameterised by a session channel $c$ and the stored value $x$:

$$\textbf{def } \mathit{Store}(c,x) = c \rhd \{\mathsf{get} : c!\langle x\rangle.\mathit{Store}\langle c,x\rangle,\ \mathsf{put} : c?(y).\mathit{Store}\langle c,y\rangle,\ \mathsf{stop} : \mathbf{0}\} \textbf{ in } \mathit{Store}\langle \mathit{eff}, i\rangle \quad (2)$$

That is, *Store* provides a choice (by $\rhd$) over channel $c$ between three behaviours labelled $\mathsf{get}$, $\mathsf{put}$, and $\mathsf{stop}$. The $\mathsf{get}$ branch sends the state $x$ on $c$ and then recurses with the same parameters, preserving the stored value. The $\mathsf{put}$ branch receives $y$ which then becomes the new state by continuing with recursive call $\mathit{Store}\langle c,y\rangle$. The $\mathsf{stop}$ branch provides finite interaction by terminating the agent. The store agent is initialised with channel *eff* and initial value $i$.

The following parameterised operations *get* and *put* then provide interaction with the store:

$$\mathit{get}(c)(x).P = \overline{c} \lhd \mathsf{get}\,.\,\overline{c}?(x).P \qquad \mathit{put}(c)\langle V\rangle.P = \overline{c} \lhd \mathsf{put}\,.\,\overline{c}!\langle V\rangle.P \quad (3)$$

where $\overline{c}$ is the opposite endpoint of a channel, and *get* selects (by the $\lhd$ operator) the $\mathsf{get}$ branch then receives a value which is bound to $x$ in the scope of $P$, and *put* selects its relevant branch then sends a value $V$ before continuing as $P$.

A process can then use *get* and *put* for stateful computation by parallel composition with *Store*, *e.g.* $\mathit{get}(\mathit{eff})(x).\mathit{put}(\mathit{eff})\langle x+1\rangle.\textbf{end} \mid \mathit{Store}\langle \mathit{eff}, i\rangle$ increments the initial value.

**2.2 Session types** Session types provide descriptions (and restrictions) of the interactions that take place over channels [9]. Session types record sequences of typed *send* ($![\tau]$) and *receive* ($?[\tau]$) interactions, terminated by the **end** marker, branched by *select* ($\oplus$) and *choice* ($\&$) interactions, with cycles provided by a fixed point $\mu\alpha$ and session variables $\alpha$:

$$S,T ::= ![\tau].S \mid ?[\tau].S \mid \oplus[l_1 : S_1, \ldots, l_n : S_n] \mid \&[l_1 : S_1, \ldots, l_n : S_n] \mid \mu\alpha.S \mid \alpha \mid \textbf{end}$$

where $\tau$ ranges over value types $\textbf{nat}, \textbf{unit}$ and session channels $S$, and $l$ ranges over labels.

$$
\begin{array}{rrll}
\textit{(value variables)} & v ::= x, y, z & \textit{(session channel variables)} & c, d, \overline{c}, \overline{d} \\
\textit{(values)} & V ::= C \mid v & & \textit{constants / variables} \\
\textit{(processes)} & P, Q ::= c?(x).P & \mid c!\langle V\rangle.P & \textit{receive / send} \\
& \mid c?(d).P & \mid c!\langle d\rangle.P & \textit{channel receive / send} \\
& \mid c \rhd \{\tilde{l} : \tilde{P}\} & \mid c \lhd l.P & \textit{branching / selection} \\
& \mid \mathbf{def}\, X(\tilde{x}, \tilde{c}) = P \,\mathbf{in}\, Q & \mid X\langle \tilde{V}, \tilde{c}\rangle & \textit{recursive definition / use} \\
& \mid \nu c.P & & \textit{channel restriction} \\
& \mid (P \mid Q) & & \textit{parallel composition} \\
& \mid \mathbf{0} & & \textit{nil process} \\
\textit{(value-types)} & \tau ::= \mathbf{unit} \mid \mathbf{nat} \mid S & \textit{(contexts)} & \Gamma ::= \emptyset \mid \Gamma, x : \tau \mid \Gamma, X : (\tilde{\tau}, \tilde{S})
\end{array}
$$

($l$ ranges over labels, $\tilde{l} : \tilde{P}$ over sequences of label-process pairs, and $\tilde{e}$ over syntax sequences)

Figure 1: Syntax of $\pi$-calculus with recursion and sessions

Figure 4 (p. 10) gives the rules of the session typing system (based on that in [9]). Session typing judgements for processes have the form $\Gamma; \Delta \vdash P$ meaning a process $P$ has value variables $\Gamma = x_1 : \tau_1 \ldots x_n : \tau_n$ and session-typed channels $\Delta = c_1 : S_1, \ldots c_n : S_n$.

For some state type $\tau$ and initial value $i : \tau$, the *Store* process (2) has session judgement:

$$\Gamma; \mathit{eff} : \mu\alpha.\, \&[\mathsf{get} : ![\tau].\alpha, \mathsf{put} : ?[\tau].\alpha, \mathsf{stop} : \mathbf{end}] \vdash \mathit{Store}\langle \mathit{eff}, i\rangle$$

That is, *eff* is a channel over which there is an sequence of offered choice between the get branch, which sends a value, put branch which receives a value, and terminated by the stop branch. The session judgements for *get* and *put* (3) are then:

$$\frac{\Gamma, x : \tau;\ \Delta, \overline{\mathit{eff}} : S \vdash P}{\Gamma; \Delta, \overline{\mathit{eff}} : \oplus[\mathsf{get} : ?[\tau].S] \vdash \mathit{get}(\mathit{eff})(x).P} \qquad \frac{\Gamma; \emptyset \vdash V : \tau \quad \Gamma; \Delta, \overline{\mathit{eff}} : S \vdash P}{\Gamma; \Delta, \overline{\mathit{eff}} : \oplus[\mathsf{put} : ![\tau].S] \vdash \mathit{put}(\mathit{eff})\langle V\rangle.P} \quad (4)$$

We use a variant of session typing where selection $\rhd$, used by *get* and *put*, has a selection session type $\oplus$ with only the selected label (seen above), and not the full range of labels offered by its dual branching process, which would be $\oplus[\mathsf{get} : ?[\tau].S, \mathsf{put} : ![\tau].S, \mathsf{stop} : \mathbf{end}]$ for both. Duality of select and branch types is achieved by using session subtyping to extend select types with extra labels [2] (see Appendix A.1 for details).

**2.3 Effect systems** Effect systems are a class of static analyses for effects, such as state [3]. Traditionally, effect systems are described as syntax-directed analyses by augmenting typing rules with effect judgements, *i.e.*, $\Gamma \vdash M : \tau, F$ where $F$ describes the effects of $M$ – usually a set of effect tokens. We define the *effect calculus*, a simple imperative language with effectful operations and a type-and-effect system defined in terms of an abstract effect algebra.

Terms comprise variables, **let**-binding, operations, and constants, and types comprise value types for natural numbers and unit:

$$M, N ::= x \mid \mathbf{let}\, x \leftarrow M \,\mathbf{in}\, N \mid \mathit{op}\, M \mid c \qquad \tau, \sigma ::= \mathbf{unit} \mid \mathbf{nat}$$

where $x$ ranges over variables, *op* over unary operations, and $c$ over constants. We do not include function types as there is no abstraction (higher-order calculi are discussed in Section 5). Constants and operations can be effectful and are instantiated to provide application-specific effectful operations in the calculus. As defaults we add zero and unit constants $0, \mathsf{unit} \in c$ and a pure successor operation for natural numbers $\mathsf{suc} \in \mathit{op}$.

$$\text{(var)}\frac{x:\tau \in \Gamma}{\Gamma \vdash x : \tau, I} \quad \text{(let)}\frac{\Gamma \vdash M : \sigma, F \quad \Gamma, x:\sigma \vdash N : \tau, G}{\Gamma \vdash \mathbf{let}\, x \leftarrow M\, \mathbf{in}\, N : \tau, F \bullet G} \quad \text{(const)}\frac{}{\Gamma \vdash c : C_\tau, C_F} \quad \text{(op)}\frac{\Gamma \vdash M : Op_\sigma, I}{\Gamma \vdash op\, M : Op_\tau, Op_F}$$

Figure 2: Type-and-effect system for the effect calculus

**Definition 1** (Effect system)**.** Let $F$ be a set of effect annotations with a monoid structure $(F, \bullet, I)$ where $\bullet$ combines effects (corresponding to sequential composition) and $I$ is the trivial effect (for pure computation). Throughout $F, G, H$ will range over effect annotations.

Figure 2 defines the type-and-effect relation. The (var) rule marks variable use as pure (with $I$). In (let), the left-to-right evaluation order of **let**-binding is exposed by the composition order of the effect $F$ of the bound term $M$ followed by effect $G$ of the **let**-body $N$. The (const) rule introduces a constant of type $C_\tau$ with effects $C_F$, and (op) applies an operation to its pure argument of type $Op_\sigma$, returning a result of type $Op_\tau$ with effect $Op_F$.

**2.4 State effects** The effect calculus can be instantiated with different notions of effect. For state, we use the effect monoid $(\mathsf{List}\,\{\mathbf{G}\,\tau, \mathbf{P}\,\tau\}, ++, [\,])$ of lists of effect tokens, where $\mathbf{G}$ and $\mathbf{P}$ represent *get* and *put* effects parameterised by a type $\tau$, $++$ concatenates lists and $[\,]$ is the empty list. Many early effect systems annotated terms with sets of effects. Here we use lists to give a more precise account of state which includes the order in which effects occur.

Terms are extended with constant $\mathsf{get}$ and unary operation $\mathsf{put}$ where $\emptyset \vdash \mathsf{get} : \tau, [\mathbf{G}\,\tau]$ and $\Gamma \vdash \mathsf{put}\, M : \mathbf{unit}, [\mathbf{P}\,\tau]$ for $\Gamma \vdash M : \tau, I$. For example, the following is a valid judgement:

$$\emptyset \vdash \mathbf{let}\, x \leftarrow \mathsf{get}\ \mathbf{in}\ \mathsf{put}\,(\mathsf{suc}\ x)\ : \mathbf{nat}, [\mathbf{G}\,\mathbf{nat}, \mathbf{P}\,\mathbf{nat}] \tag{5}$$

Type safety of the store is enforced by requiring that any *get* effects must have the same type as their nearest preceding *put* effect. We implicitly apply this condition throughout.

**2.5 Sessions as effects** The session types of processes interacting with *Store* provide the same information as the state effect system. Indeed, we can define a bijection between state effect annotations and the session types of *get* and *put* (4):

$$[\![[\,]]\!] = \mathbf{end} \qquad [\![(\mathbf{G}\,\tau) :: F]\!] = \oplus[\mathsf{get} : ?[\tau].[\![F]\!]] \qquad [\![(\mathbf{P}\,\tau) :: F]\!] = \oplus[\mathsf{put} : ![\tau].[\![F]\!]] \tag{6}$$

where :: is the *cons* operator for lists. Thus processes interacting with *Store* have session types corresponding to effect annotations. For example, the following has the same state semantics as (5) and isomorphic session types:

$$\emptyset; \overline{eff} : [\![[\mathbf{G}\,\mathbf{nat}, \mathbf{P}\,\mathbf{nat}]]\!] \vdash get(eff)(x).put(eff)\langle \mathbf{suc}\, x\rangle \tag{7}$$

# 3 Embedding the effect calculus into the $\pi$-calculus

Our embedding is based on the embedding of the call-by-value $\lambda$-calculus (without effects) into the $\pi$-calculus [5, 8] taking $\mathbf{let}\, x \leftarrow M\, \mathbf{in}\, N = (\lambda x.N)\, M$. Since effect calculus terms return a result and $\pi$-calculus processes do not, the embedding is parameterised by a *result channel* $r$ over which the return value is sent, written $[\![-]\!]_r$. Variables and pure **let**-binding are embedded:

$$[\![\mathbf{let}\, x \leftarrow M\, \mathbf{in}\, N]\!]_r = \ \nu q.\ ([\![M]\!]_q \mid \overline{q}?(x).[\![N]\!]_r) \qquad [\![x]\!]_r = \ r!\langle x\rangle \tag{8}$$

Variables are simply sent over the result channel. For **let**, an intermediate channel $q$ is created over which the result of the bound term $M$ is sent by the left-hand parallel process $[\![M]\!]_q$ and received and bound to $x$ by the right-hand process before continuing with $[\![N]\!]_r$. This enforces a left-to-right, CBV evaluation order (despite the parallel composition).

Pure constants and unary operations can be embedded similarly to variables and **let** given suitable value operations in the $\pi$-calculus. For example, successor and zero are embedded as:

$$[\![\mathsf{suc}\, M]\!]_r = \nu q.\ ([\![M]\!]_q \mid \overline{q}?(x).r!\langle \mathbf{suc}\ x\rangle) \qquad [\![\mathsf{zero}]\!]_r = r!\langle \mathbf{zero}\rangle \tag{9}$$

Given a mapping $[\![-]\!]$ from effect calculus types to corresponding value types in the $\pi$-calculus, the above embedding of terms (8),(9) can be extended to typing judgements as follows (where $[\![\Gamma]\!]$ interprets the type of each free-variable assumption, preserving the structure of $\Gamma$):

$$[\![\Gamma \vdash M : \tau]\!]_r \ = \ [\![\Gamma]\!]; r : ![\![\tau]\!].\mathbf{end} \vdash [\![M]\!]_r \tag{10}$$

**With effects** Effectful computations are embedded by interactions with a side-effect handling agent over a session channel. The embedding, written $[\![-]\!]_r^{\mathit{eff}}$, maps a judgement $\Gamma \vdash M : \tau,\ F$ to a session type judgement with channels $\Delta = (r : ![\![\tau]\!].\mathbf{end},\ \mathit{eff} : [\![F]\!])$ *i.e.*, the effect annotation $F$ is interpreted as the session type of channel $\mathit{eff}$. For state, this interpretation is defined as in eq. (6). The embedding first requires an intermediate step, written $(\!|-|\!)_r^{\mathsf{ei},\mathsf{eo}}$

$$(\!|\, \Gamma \vdash M : \tau, F \,|\!)_r^{\mathsf{ei},\mathsf{eo}} \ = \ \forall g. \quad [\![\Gamma]\!];\ r : ![\![\tau]\!],\ \mathsf{ei} : ?[\![F \bullet g]\!],\ \overline{\mathsf{eo}} : ![\![g]\!] \ \vdash (\!|\, M \,|\!)_r^{\mathsf{ei},\mathsf{eo}} \tag{11}$$

where $\mathsf{ei}$ and $\mathsf{eo}$ are session channels over which session channels for simulating effects are communicated: $\mathsf{ei}$ receives a channel of session type $[\![F \bullet g]\!]$ (*i.e.*, capable of carrying out effects $F \bullet g$) and $\overline{\mathsf{eo}}$ sends a channel of session type $[\![g]\!]$ (capable of carrying out effects $g$). Here the effect $g$ is universally quantified at the meta level. This provides a way to thread a channel for effect interactions through a computation, such as in the case of **let**-binding.

The interpretation is then defined:

$$\begin{aligned}
(\!|\, \mathbf{let}\, x \leftarrow M\, \mathbf{in}\, N \,|\!)_r^{\mathsf{ei},\mathsf{eo}} &= \nu q, \mathsf{ea}\,.\, ((\!|\, M \,|\!)_q^{\mathsf{ei},\mathsf{ea}} \mid \overline{q}?(x).(\!|\, N \,|\!)_r^{\mathsf{ea},\mathsf{eo}}) \\
(\!|\, x \,|\!)_r^{\mathsf{ei},\mathsf{eo}} &= \mathsf{ei}?(c).r!\langle x\rangle.\overline{\mathsf{eo}}!\langle c\rangle \\
(\!|\, C \,|\!)_r^{\mathsf{ei},\mathsf{eo}} &= \mathsf{ei}?(c).[\![C]\!]_r.\overline{\mathsf{eo}}!\langle c\rangle \qquad \textit{(when } C \textit{ is pure)} \\
(\!|\, \mathit{op}\, M \,|\!)_r^{\mathsf{ei},\mathsf{eo}} &= \mathsf{ei}?(c).[\![\mathit{op}\, M]\!]_r.\overline{\mathsf{eo}}!\langle c\rangle \qquad \textit{(when op is pure)}
\end{aligned} \tag{12}$$

The embedding of variables is straightforward, where the channel for simulating effects is received on $\mathsf{ei}$ and then sent without use on $\overline{\mathsf{eo}}$. Embedding pure operations/constants is similar, reusing the pure embedding defined in equation (9).

The **let** case resembles the pure embedding of **let** but threads through an effect-carrying session channel. Intermediate channel $\mathsf{ea}$ is introduced over which the effect channel is passed from the embedding of $M$ to $N$. Let $\Gamma \vdash M : \tau, F$ and $\Gamma, x : \sigma \vdash N : \tau, G$ then the universally quantified effect variable $\forall g$ for $(\!|\, M \,|\!)_q^{\mathsf{ei},\mathsf{ea}}$ is instantiated to $G \bullet g$. The following partial session-type derivation for the **let** encoding shows the propagation of effects via session types:

$$\dfrac{\dfrac{q : ![\![\sigma]\!], \mathsf{ei} : ?[\![F \bullet G \bullet g]\!], \overline{\mathsf{ea}} : ![\![G \bullet g]\!] \vdash (\!|\, M \,|\!)_q^{\mathsf{ei},\mathsf{ea}} \qquad \overline{q} : ?[\![\sigma]\!], r : ![\![\tau]\!], \mathsf{ea} : ?[\![G \bullet g]\!], \overline{\mathsf{eo}} : ![\![g]\!] \vdash \overline{q}?(x).(\!|\, N \,|\!)_r^{\mathsf{ea},\mathsf{eo}}}{r : ![\![\tau]\!], \mathsf{ei} : ?[\![F \bullet G \bullet g]\!], \overline{\mathsf{eo}} : ![\![g]\!], q : ![\![\sigma]\!], \overline{q} : ?[\![\sigma]\!], \overline{\mathsf{ea}} : ![\![G \bullet g]\!], \mathsf{ea} : ?[\![G \bullet g]\!] \vdash (\!|\, M \,|\!)_q^{\mathsf{ei},\mathsf{ea}} \mid \overline{q}?(x).(\!|\, N \,|\!)_r^{\mathsf{ea},\mathsf{eo}}}}{r : ![\![\tau]\!], \mathsf{ei} : ?[\![F \bullet G \bullet g]\!], \overline{\mathsf{eo}} : ![\![g]\!] \vdash \nu q, \mathsf{ea}.\ ((\!|\, M \,|\!)_q^{\mathsf{ei},\mathsf{ea}} \mid \overline{q}?(x).(\!|\, N \,|\!)_r^{\mathsf{ea},\mathsf{eo}})}$$

The *get* and *put* operations of our state effects are embedded similarly to equation (3) (page 2), but with the receiving and sending of the session channel which interacts with the store:

$$\begin{aligned}(\!|\, \mathsf{get} \,|\!)_r^{\mathsf{ei},\mathsf{eo}} &= \mathsf{ei}?(c).c \lhd \mathsf{get} \,.\, c?(x).r!\langle x\rangle.\overline{\mathsf{eo}}!\langle c\rangle \\ (\!|\, \mathsf{put}\ M \,|\!)_r^{\mathsf{ei},\mathsf{eo}} &= \nu q.\ ([\![M]\!]_q \mid\ \mathsf{ei}?(c).\overline{q}?(x).c \lhd \mathsf{put} \,.\, c!\langle x\rangle.r!\langle \mathbf{unit}\rangle.\overline{\mathsf{eo}}!\langle c\rangle) \end{aligned} \tag{13}$$

The embedding of get receives channel $c$ over which it performs its effect by selecting the get branch and receiving $x$ which is sent as the result on $r$ before sending $c$ on $\overline{\mathsf{eo}}$. The put embedding is similar to get and **let**, but using the pure embedding $[\![M]\!]_q$ since $M$ is pure.

The full embedding is then defined in terms of the intermediate as follows:

$$[\![\Gamma \vdash M : \tau, F]\!]_r^{\mathit{eff}} = [\![\Gamma]\!]; r : ![\![\tau]\!], \mathit{eff} : [\![F]\!] \vdash \nu \mathsf{ei}, \mathsf{eo}.\ ((\!|\, \Gamma \vdash M : \tau, F \,|\!)_r^{\mathsf{ei},\mathsf{eo}} \mid \overline{\mathsf{ei}}!\langle \mathit{eff}\rangle.\mathsf{eo}?(c)) \tag{14}$$

where *eff* is the free session channel over which effects are performed.

Finally, the embedded program is composed in parallel with the variable agent, for example:

$$\mathbf{def}\ \mathit{Store}(c, x) = \ldots (\text{see } (2)) \ \mathbf{in}\ \mathit{Store}\langle \overline{\mathit{eff}}, 0\rangle \mid [\![\mathbf{let}\ x \leftarrow \mathsf{get}\ \mathbf{in}\ \mathsf{put}\ (\mathsf{suc}\ x)]\!]_r^{\mathit{eff}} \tag{15}$$

**3.1 Soundness** The effect calculus exhibits the equational theory defined by the relation $\equiv$ in Figure 3, which enforces monoidal properties on effects and the effect algebra (assoc),(unitL),(unitR), and which allows pure computations to commute with effectful ones (comm). Our embedding is sound with respect to these equations and the weak bisimulation relation of the $\pi$-calculus with sessions (see [4] for more on weak bisimulation).

**Theorem 1** (Soundness)**.** *If* $\Gamma \vdash M \equiv N : \tau, F$ *then* $[\![\Gamma]\!]; (r :![\![\tau]\!].\mathbf{end}, e : [\![F]\!]) \vdash [\![M]\!]_r^e \approx [\![N]\!]_r^e$

Appendix C provides the proof. The proof of soundness for (comm) uses the following lemma on the encoding of pure terms (those annotated with $I$), which requires an additional restriction on the effect algebra.

**Lemma 1** (Pure encoding)**.** *An effect system where* $\forall F, G.(F \bullet G \equiv I) \Rightarrow (F \equiv G \equiv I)$ *has the following property of the intermediate encoding, for all* $M$, $\Gamma$, $\tau$*:*

$$(\!|\, \Gamma \vdash M : \tau, I \,|\!)_r^{\mathsf{ei},\mathsf{eo}} \ \approx\ \mathsf{ei}?(c).\overline{\mathsf{eo}}!\langle c\rangle.[\![M]\!]_r$$

That is, the intermediate encoding of a pure term is bisimilar to a pure encoding (without effect simulation) prefixed by receiving an effect-simulating channel $c$ on ei and sending it on $\overline{\mathsf{eo}}$ without any use. Appendix C provides the proof. The state effect system described here satisfies this additional condition on the effect algebra since any two lists whose concatenation is the empty list implies that both lists are themselves the empty list.

$$\text{(assoc)}\ \frac{\Gamma \vdash M : \sigma, F \quad \Gamma, x : \sigma \vdash N : \tau', G \quad \Gamma, y : \tau' \vdash P : \tau, H \quad x \notin FV(P)}{(\mathbf{let}\ y \leftarrow (\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N)\ \mathbf{in}\ P) \equiv (\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ (\mathbf{let}\ y \leftarrow N\ \mathbf{in}\ P)) : \tau, F \bullet G \bullet H}$$

$$\text{(unitL)}\ \frac{\Gamma \vdash x : \sigma, I \quad \Gamma, y : \sigma \vdash M : \tau, F}{\Gamma \vdash (\mathbf{let}\ y \leftarrow x\ \mathbf{in}\ M) \equiv M[x/y] : \tau, F} \qquad \text{(unitR)}\ \frac{\Gamma \vdash M : \tau, F}{\Gamma \vdash (\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ x) \equiv M : \tau, F}$$

$$\text{(comm)}\ \frac{\Gamma \vdash M : \tau_1,\ I \qquad \Gamma \vdash N : \tau_2,\ F \qquad \Gamma, x : \tau_1, y : \tau_2 \vdash P : \tau,\ G \quad x \notin FV(N) \quad y \notin FV(M)}{\Gamma \vdash \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ (\mathbf{let}\ y \leftarrow N\ \mathbf{in}\ P) \equiv \mathbf{let}\ y \leftarrow N\ \mathbf{in}\ (\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ P) : \tau,\ F \bullet G}$$

Figure 3: Equations of the effect calculus

# 4 Discussion

**Concurrent effects** In a concurrent setting, side effects can lead to non-determinism and race conditions. For example, the program in the introduction $\mathbf{put}\, x\, ((\mathbf{get}\, x)+2) \,|\, \mathbf{put}\, x\, ((\mathbf{get}\, x)+1)$ has four possible final values for $x$ due to arbitrarily interleaved **get** and **put** operations.

Consider an extension to the source language which adds a binary operator for parallel composition | (we elide details of the type-and-effect rule, but an additional effect operator, describing parallel effects, might be included). We might then consider the following encoding using the parallel composition of the $\pi$-calculus:

$$[\![M \mid N]\!]_r^{\mathit{eff}} = \nu q_1, q_2\,.\,([\![M]\!]_{q_1}^{\mathit{eff}} \mid [\![N]\!]_{q_2}^{\mathit{eff}} \mid \overline{q_1}?(x).\overline{q_2}?(y).r!\langle(x,y)\rangle)$$

where $q_1$ and $q_2$ are the result channels for each term, from which the results are paired and sent over $r$. This encoding is not well-typed under the session typing scheme: the (par) rule (see Figure 4, p. 10) requires that the session channel environments of each process be disjoint but *eff* appears on both sides. Thus, session types naturally prevent effect interference.

Concurrent effects can be allowed by introducing *shared channels*, over which sessions can be initiated [9]. One possible semantics for parallel composition in the source language might be that the store is "locked" by each process, providing atomicity. This can be described by the following redefinition of *Store* and the encoding of parallel composition:

$$\mathbf{def}\ \mathit{Store}(c,x) = \ldots(\mathit{see}(2))\ \mathbf{in}\ \mathbf{accept}\ k(c).\mathit{Store}\langle c,0\rangle$$

$$[\![M \mid N]\!]_r^k = \nu q_1, q_2.\ (\mathbf{request}\ k(c).[\![M]\!]_{q_1}^c \mid \mathbf{request}\ k(d).[\![N]\!]_{q_2}^d \mid \overline{q_1}?(x).\overline{q_2}?(y).r!\langle(x,y)\rangle)$$

where $k$ is a shared channel and **request**/**accept** initiate two separate binary sessions between the store process and the client processes, ensuring atomicity of side effects within each process.

Various other kinds of concurrent effect interaction can be described using the rich language of the session calculus and variations of our embedding.

**Compiling to the session calculus** One use for our embedding is as a typed intermediate language for a compiler since the $\pi$-calculus with sessions provides an expressive language for concurrency. For example, even without explicit concurrency in the source language our encoding can be used to introduce implicit parallelism as part of a compilation step via the session calculus. In the case of compiling a term which matches either side of the (comm) rule above, a pure term $M$ can be computed in parallel with $N$, *i.e.*, given terms $\Gamma \vdash M : \tau_1$, $I$ and $\Gamma \vdash N : \tau_2$, $F$ and $\Gamma, x : \tau_1, y : \tau_2 \vdash P : \tau$, $G$ where $x \notin FV(N), y \notin FV(M)$ then the following specialised encodings can be given:

$$(\!|\, \mathbf{let}\, y \leftarrow N\, \mathbf{in}\, (\mathbf{let}\, x \leftarrow M\, \mathbf{in}\, P)\,|\!)_r^{\mathsf{ei},\mathsf{eo}} =$$

$$(\!|\, \mathbf{let}\, x \leftarrow M\, \mathbf{in}\, (\mathbf{let}\, y \leftarrow N\, \mathbf{in}\, P)\,|\!)_r^{\mathsf{ei},\mathsf{eo}} = \nu\, q, s, \mathsf{eb}.\, ([\![M]\!]_q \mid (\!|\, N\,|\!)_s^{\mathsf{ei},\mathsf{eb}} \mid \overline{q}?(x).\overline{s}?(y).(\!|\, P\,|\!)_r^{\mathsf{eb},\mathsf{eo}})$$

This alternate encoding introduces the opportunity for parallel evaluation of $M$ and $N$. It is enabled by the effect system (which annotates $M$ with $I$) and it is sound: it is weakly bisimilar to the usual encoding (which follows from the soundness proof of (comm) in Appendix C and the pure encoding lemma 1).

# 5 Summary and further work

This paper showed that sessions and session types are expressive enough to encode stateful computations with an effect system. We formalised this via a sound embedding of a simple, and

general, effect calculus into the session calculus. Whilst we have focussed on causal state effects, our effect calculus and embedding can also be instantiated for I/O effects, where *input/output* operations and effects have a similar form to *get/put*. We considered only state effects on a single store, but traditional effect systems account for multiple stores via *regions*. Our approach could be extended with a store and session channel per region. Other instantiations of our effect calculus/embedding are further work, for example, for set-based effects.

Effect reasoning is difficult in higher-order settings as the effects of abstracted computations are locally unknown. Effect systems account for this by annotating function types with the *latent effects* of a function which are delayed till application. A possible encoding of a function type with latent effects into a session type could be following:

$$[\![\sigma \xrightarrow{F} \tau]\!] = ![\![\sigma]\!] \,.\, ![?[\![F \bullet G]\!]] \,.\, ![![\![G]\!]] \,.\, ![![\![\tau]\!]]$$

*i.e.*, a channel over which four things can be sent: a $[\![\sigma]\!]$ value for the function argument, a channel which can receive a further channel capable of simulating effects $F \bullet G$, a channel which can send a channel capable of simulating effects $G$, and a channel which can send a $[\![\tau]\!]$ for the result. Thus, the encoding of a function receives effect handling channels which have the same form as the effect channels for first-order term encodings. A full, formal treatment of effects in a higher-order setting is forthcoming work.

Effects systems also commonly include a (partial) ordering on effects, which describes how effects can be overapproximated [3]. For example, causal state effects are ordered by prefix inclusion, thus an expression $M$ with judgement $\Gamma \vdash M : \tau, [\mathbf{G}\,\tau]$ might have its effects over-approximated (via a subsumption rule) to $\Gamma \vdash M : \tau, [\mathbf{G}\,\tau, \mathbf{P}\,\tau']$. It is possible to account for (some) subeffecting using subtyping of sessions. Formalising this is further work.

Whilst we have embedded effects into sessions, the converse seems possible: to embed sessions into effects. Nielson and Nielson previously defined an effect system for higher-order concurrent programs which resembles some aspects of session types [6]. Future work is to explore mutually inverse embeddings of sessions and effects. Relatedly, further work is to explore whether various kinds of *coeffect system* (which dualise effect systems, analysing context and resource use [7]) such as bounded linear logics, can also be embedded into session types.

**Acknowledgements** Thanks to Tiago Cogumbreiro and the anonymous reviewers for their feedback. The work has been partially sponsored by EPSRC EP/K011715/1, EP/K034413/1, and EP/L00058X/1, and EU project FP7-612985 UpScale.

# References

[1] R. L. Bocchino Jr, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. *In Proocedings of OOPSLA 2009*, pages 97–116, 2009.

[2] T.-C. Chen, M. Dezani-Ciancaglini, and N. Yoshida. On the preciseness of subtyping in session types. In *PPDP 2014*, pages 146–135. ACM Press, 2014.

[3] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *Proceedings of Conference on LISP and func. prog.*, LFP '86, 1986.

[4] D. Kouzapas, N. Yoshida, R. Hu, and K. Honda. On asynchronous eventful session semantics. *MSCS*. To appear (2015).

[5] R. Milner. Functions as processes. *MSCS*, 2(2):119–141, 1992.

[6] H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In *Proceedings of the symposium on Principles of programming languages*, pages 84–97. ACM, 1994.

[7] T. Petricek, D. A. Orchard, and A. Mycroft. Coeffects: a calculus of context-dependent computation. In *Proceedings of ICFP*, pages 123–135, 2014.

[8] D. Sangiorgi and D. Walker. *The $\pi$-Calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[9] N. Yoshida and V. T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electr. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.

# A Session types

Figure 4 gives the full session typing system used in this work which is close to that of Yoshida and Vasconcelos [9]. For a session $S$, its *dual* $\overline{S}$ is defined in the usual way [9]. Throughout we used the usual convention of eliding a trailing $\mathbf{0}$, *e.g.*, writing $r!\langle x\rangle$ instead of $r!\langle x\rangle.\mathbf{0}$, and likewise for session types, *e.g.*, $![\tau]$ instead of $![\tau].\mathbf{end}$.

$\boxed{\Gamma;\Delta \vdash V : \tau}$ (*value typing*) (const) $\dfrac{C : C_\tau}{\Gamma;\emptyset \vdash C : C_\tau}$ (var) $\dfrac{v : \tau \in \Gamma}{\Gamma;\emptyset \vdash v : \tau}$ (suc) $\dfrac{\Gamma \vdash V : \mathbf{nat}}{\Gamma \vdash \mathbf{suc}\, V : \mathbf{nat}}$

$\boxed{\Gamma;\Delta \vdash P}$ (*process typing*)

(end) $\Gamma;\tilde{c} : \mathbf{end} \vdash \mathbf{0}$ (par) $\dfrac{\Gamma;\Delta_1 \vdash P \quad \Gamma;\Delta_2 \vdash Q}{\Gamma;\Delta_1,\Delta_2 \vdash P \mid Q}$ (restrict) $\dfrac{\Gamma;\Delta, c : S, \overline{c} : \overline{S} \vdash P}{\Gamma;\Delta \vdash \nu c.P}$

(def) $\dfrac{\Gamma, X : (\tilde{\tau},\tilde{S}), \tilde{x} : \tilde{\tau};\ \tilde{c} : \tilde{S} \vdash P \quad \Gamma, X : (\tilde{\tau},\tilde{S});\Delta \vdash Q}{\Gamma;\Delta \vdash \mathbf{def}\, X(\tilde{x},\tilde{c}) = P\ \mathbf{in}\ Q}$ (dvar) $\dfrac{\Gamma;\tilde{d} : \mathbf{end} \vdash \tilde{e} : \tilde{\tau}}{\Gamma, X : (\tilde{\tau},\tilde{S});\tilde{c} : \tilde{S}, \tilde{d} : \mathbf{end} \vdash X\langle \tilde{e},\tilde{c}\rangle}$

(chan-recv) $\dfrac{\Gamma;\Delta, c : T, d : S \vdash P}{\Gamma;\Delta, c :?[S].T \vdash c?(d).P}$ (chan-send) $\dfrac{\Gamma;\Delta, c : T \vdash P}{\Gamma;\Delta, c :![S].T, d : S \vdash c!\langle d\rangle.P}$

(recv) $\dfrac{\Gamma, x : \tau;\Delta, c : S \vdash P}{\Gamma;\Delta, c :?[\tau].S \vdash c?(x).P}$ (send) $\dfrac{\Gamma;\emptyset \vdash e : \tau \quad \Gamma;\Delta, c : S \vdash P}{\Gamma;\Delta, c :![\tau].S \vdash c!\langle e\rangle.P}$

(branch) $\dfrac{\Gamma;\Delta, c : S_i \vdash P_i}{\Gamma;\Delta, c : \&[\tilde{l} : \tilde{S}] \vdash c \rhd \{\tilde{l} : \tilde{P}\}}$ (select) $\dfrac{\Gamma;\Delta, c : S\ \vdash P}{\Gamma;\Delta, c : \oplus[l : S] \vdash c \lhd l.P}$

where $\tilde{x} : \tilde{\tau}$ is shorthand for a sequence of variable-type pairs, and similarly $\tilde{c} : \tilde{S}$ for channels, $\tilde{l} : \tilde{S}$ for labels and sessions, and $\tilde{e}$ for a sequence of expressions.

Figure 4: Session typing relation over the $\pi$-calculus with recursion and sessions [9].

## A.1 Subtyping and selection

Our session typing system assigns selection types that include only the label $l$ being selected ((select) in Figure 4). Duality with branch types is provided by subtyping on selection types:

$$\text{(sel)} \qquad \oplus[\tilde{l} : \tilde{S}] \prec \oplus[\tilde{l} : \tilde{S}, \tilde{l}' : \tilde{S}']$$

(this is a special case of the usual full subtyping rule for selection, see [2, [SUB-SEL], Table 5, p. 4]). Therefore, for example, the *get* process could be typed:

$$\text{(sub)}\ \frac{\dfrac{\Gamma, x:\tau; \Delta; \overline{c}:S \vdash P}{\Gamma; \Delta, \overline{c}:\oplus[\mathsf{get}:?[\tau].S] \vdash get(c)(x).P} \quad \text{(sel)}\ \dfrac{}{\oplus[\mathsf{get}:?[\tau].S] \prec \oplus[\mathsf{get}:?[\tau].S, \mathsf{put}:![\tau].S]}}{\Gamma; \Delta, \overline{c}:\oplus[\mathsf{get}:?[\tau].S, \mathsf{put}:![\tau].S] \vdash get(c)(x).P}$$

However, such subtyping need only be applied when duality is being checked, that is, when opposing endpoints of a channel are bound by channel restriction, $\nu c.P$. We take this approach, thus subtyping is only used with channel restriction such that, prior to restriction, session types can be interpreted as effect annotations with selection types identifying effectful operations.

# B Agda encoding

The Agda formalisation of our embedding defines data types of typed terms for the effect calculus `_,_⊢_,_` and session calculus `_*_⊢_`, indexed by the terms' effects, types, and contexts:

```
data_,_⊢_,_ (eff : Effect) : (Gam : Context Type) -> Type -> (Carrier eff) -> Set where ...
data _*_⊢_: (Γ : Context VType) -> (Σ : Context SType) -> (t : PType) -> Set where ...
```

These type constructors are multi-arity infix operators. For the effect calculus type, the first index `eff : Effect` is a record providing the effect algebra, operations, and constants, of which the `Carrier` field holds the type for effect annotations. The embedding is then a function:

```
embed : forall {Γ τ F} -> (e : stEff , Γ ⊢ τ , F)
                       -> (map interpT Γ) * ((Em , [ interpT τ ]!· end) , interpEff F) ⊢ proc
```

where `interpT : Type -> VType` maps types of the effect calculus to value types for sessions, and `interpEff : List StateEff -> SType` maps state effect annotations to session types `SType`. Here the constructor `[_]!·_` is a binary data constructor representing the session type for send. The intermediate embedding has the type (which also uses the receive session type `[_]?·_`):

```
embedInterm : forall {Γ τ F G}
  -> (M : stEff , Γ ⊢ τ , F)
  -> (map interpT Γ * (((Em , [ interpT τ ]!· end),
                           , [ sess (interpEff (F ++ G)) ]?· end)
                           , [ sess (interpEff G) ]!·end) ⊢ proc
```

# C Soundness proof of embedding, wrt. Figure 3 equations

**Theorem** (Soundness). If $\Gamma \vdash M \equiv N : \tau, F$ then $[\![\Gamma]\!]; (r : ![\![\tau]\!].\mathbf{end}, e : [\![F]\!]) \vdash [\![M]\!]^e_r \approx [\![N]\!]^e_r$

**Proof** Since $[\![M]\!]^e_r = \nu \mathsf{ei}, \mathsf{eo}.\ (\!|\, M \,|\!)^{\mathsf{ei},\mathsf{eo}}_r \mid \overline{\mathsf{ei}}!\langle e\rangle.\mathsf{eo}?(c))$ and $[\![N]\!]^e_r = \nu \mathsf{ei}, \mathsf{eo}.\ (\!|\, N \,|\!)^{\mathsf{ei},\mathsf{eo}}_r \mid \overline{\mathsf{ei}}!\langle e\rangle.\mathsf{eo}?(c))$ (eq. 14) we need only consider $(\!|\, M \,|\!)^{\mathsf{ei},\mathsf{eo}}_r \approx (\!|\, N \,|\!)^{\mathsf{ei},\mathsf{eo}}_r$, *i.e.*, bisimilarity of the intermediate embeddings. We address each equation in turn. The relation $\overset{\text{def}}{=}$ denotes definitional equality based on $(\!|-|\!)^{\mathsf{ei},\mathsf{eo}}_r$.

(unitR)

$$\begin{aligned}
&(\!|\, \mathbf{let}\, x \leftarrow M\, \mathbf{in}\, x \,|\!)^{\mathsf{ei},\mathsf{eo}}_r \\
&\overset{\text{def}}{=} \nu\, q, \mathsf{ea}.\, ((\!|\, M \,|\!)^{\mathsf{ei},\mathsf{ea}}_q \mid \overline{q}?(x).\mathsf{ea}?(c).r!\langle x\rangle.\overline{\mathsf{eo}}!\langle c\rangle) \\
&\approx \nu\, \mathsf{ea}.\, ((\!|\, M \,|\!)^{\mathsf{ei},\mathsf{ea}}_r \mid \mathsf{ea}?(c).\overline{\mathsf{eo}}!\langle c\rangle) && \{\text{forwarding } q \to r,\ \beta\text{-reduction}\} \\
&\approx (\!|\, M \,|\!)^{\mathsf{ei},\mathsf{eo}}_r \quad \square && \{\text{forwarding } \mathsf{ea} \to \mathsf{eo},\ \beta\text{-reduction}\}
\end{aligned}$$

(unitL)

$$
\begin{aligned}
&(\!| \mathbf{let}\, y \leftarrow x \,\mathbf{in}\, M |\!)^{\mathsf{ei},\mathsf{eo}}_r \\
&\stackrel{\text{def}}{=} \nu\, q, \mathsf{ea}.((\!| x |\!)^{\mathsf{ei},\mathsf{ea}}_q \mid \overline{q}?(y).(\!| M |\!)^{\mathsf{ea},\mathsf{eo}}_r) \\
&\stackrel{\text{def}}{=} \nu\, q, \mathsf{ea}.(\mathsf{ei}?(c).q!\langle x\rangle.\overline{\mathsf{ea}}!\langle c\rangle \mid \overline{q}?(y).(\!| M |\!)^{\mathsf{ea},\mathsf{eo}}_r) \\
&\approx \nu\, \mathsf{ea}.(\mathsf{ei}?(c).\overline{\mathsf{ea}}!\langle c\rangle \mid (\!| M |\!)^{\mathsf{ea},\mathsf{eo}}_r[x/y]) && \{\beta, \text{structural congruence}\} \\
&\approx (\!| M |\!)^{\mathsf{ei},\mathsf{eo}}_r[x/y] && \{\text{forwarding } \mathsf{ei} \to \mathsf{ea}\} \\
&\approx (\!| M[x/y] |\!)^{\mathsf{ei},\mathsf{eo}}_r \quad \square && \{\text{var substitution preserved by } (\!|-|\!)\}
\end{aligned}
$$

(assoc)

$$
\begin{aligned}
&(\!| \mathbf{let}\, y \leftarrow (\mathbf{let}\, x \leftarrow M \,\mathbf{in}\, N) \,\mathbf{in}\, P |\!)^{\mathsf{ei},\mathsf{eo}}_r \\
&\stackrel{\text{def}}{=} \nu\, q, \mathsf{ea}.\, ((\!| \mathbf{let}\, x \leftarrow M \,\mathbf{in}\, N |\!)^{\mathsf{ei},\mathsf{ea}}_q \mid \overline{q}?(y).(\!| P |\!)^{\mathsf{ea},\mathsf{eo}}_r) \\
&\stackrel{\text{def}}{=} \nu\, q, \mathsf{ea}.\, (\nu\, q1, \mathsf{eb}.\, ((\!| M |\!)^{\mathsf{ei},\mathsf{eb}}_{q1} \mid \overline{q1}?(x).(\!| N |\!)^{\mathsf{eb},\mathsf{ea}}_q) \mid \overline{q}?(y).(\!| P |\!)^{\mathsf{ea},\mathsf{eo}}_r) \\
(*) &\approx \nu\, q, \mathsf{ea}, q1, \mathsf{eb}.\, ((\!| M |\!)^{\mathsf{ei},\mathsf{eb}}_{q1} \mid \overline{q1}?(x).(\!| N |\!)^{\mathsf{eb},\mathsf{ea}}_q \mid \overline{q}?(y).(\!| P |\!)^{\mathsf{ea},\mathsf{eo}}_r) && \{\text{structural congruence}\}
\end{aligned}
$$

$$
\begin{aligned}
&(\!| \mathbf{let}\, x \leftarrow M \,\mathbf{in}\, (\mathbf{let}\, y \leftarrow N \,\mathbf{in}\, P) |\!)^{\mathsf{ei},\mathsf{eo}}_r \\
&\stackrel{\text{def}}{=} \nu\, q, \mathsf{ea}.\, ((\!| M |\!)^{\mathsf{ei},\mathsf{ea}}_q \mid \overline{q}?(x).(\!| \mathbf{let}\, y \leftarrow N \,\mathbf{in}\, P |\!)^{\mathsf{ea},\mathsf{eo}}_r) \\
&\stackrel{\text{def}}{=} \nu\, q, \mathsf{ea}.\, ((\!| M |\!)^{\mathsf{ei},\mathsf{ea}}_q \mid \overline{q}?(x).\nu\, q1, \mathsf{eb}.\, ((\!| N |\!)^{\mathsf{ea},\mathsf{eb}}_{q1} \mid \overline{q1}?(y).(\!| P |\!)^{\mathsf{eb},\mathsf{eo}}_r)) \\
&\approx \nu\, q, \mathsf{ea}, q1, \mathsf{eb}.\, ((\!| M |\!)^{\mathsf{ei},\mathsf{ea}}_q \mid \overline{q}?(x).(\!| N |\!)^{\mathsf{ea},\mathsf{eb}}_{q1} \mid \overline{q1}?(y).(\!| P |\!)^{\mathsf{eb},\mathsf{eo}}_r) && \{\text{sequentiality, } x \notin fv(P)\} \\
&\approx \nu\, q, \mathsf{ea}, q1, \mathsf{eb}.\, ((\!| M |\!)^{\mathsf{ei},\mathsf{eb}}_{q1} \mid \overline{q1}?(x).(\!| N |\!)^{\mathsf{eb},\mathsf{ea}}_q \mid \overline{q}?(y).(\!| P |\!)^{\mathsf{ea},\mathsf{eo}}_r) && \{\alpha, \mathsf{ea} \leftrightarrow \mathsf{eb}, q \leftrightarrow q1\} \\
&\approx (*) \quad \square
\end{aligned}
$$

(comm)

$$
\begin{aligned}
&(\!| \mathbf{let}\, x \leftarrow M \,\mathbf{in}\, (\mathbf{let}\, y \leftarrow N \,\mathbf{in}\, P) |\!)^{\mathsf{ei},\mathsf{eo}}_r \\
&\stackrel{\text{def}}{=} \nu\, q, \mathsf{ea}.\, ((\!| M |\!)^{\mathsf{ei},\mathsf{ea}}_q \mid \overline{q}?(x).(\!| \mathbf{let}\, y \leftarrow N \,\mathbf{in}\, P |\!)^{\mathsf{ea},\mathsf{eo}}_r) \\
&\stackrel{\text{def}}{=} \nu\, q, \mathsf{ea}.\, ((\!| M |\!)^{\mathsf{ei},\mathsf{ea}}_q \mid \overline{q}?(x).\nu\, q1, \mathsf{eb}.\, ((\!| N |\!)^{\mathsf{ea},\mathsf{eb}}_{q1} \mid \overline{q1}?(y).(\!| P |\!)^{\mathsf{eb},\mathsf{eo}}_r)) \\
&\approx \nu\, q, \mathsf{ea}, q1, \mathsf{eb}.\, ((\!| M |\!)^{\mathsf{ei},\mathsf{ea}}_q \mid (\!| N |\!)^{\mathsf{ea},\mathsf{eb}}_{q1} \mid \overline{q}?(x).\overline{q1}?(y).(\!| P |\!)^{\mathsf{eb},\mathsf{eo}}_r) && \{\text{sequentiality, } x \notin fv(N)\} \\
&\approx \nu\, q, \mathsf{ea}, q1, \mathsf{eb}.\, (\mathsf{ei}?(c).\overline{\mathsf{ea}}!\langle c\rangle.[\![M]\!]_q \mid (\!| N |\!)^{\mathsf{ea},\mathsf{eb}}_{q1} \mid \overline{q}?(x).\overline{q1}?(y).(\!| P |\!)^{\mathsf{eb},\mathsf{eo}}_r) && \{\text{purity lemma on } M\} \\
(*) &\approx \nu\, q, q1, \mathsf{eb}.\, ([\![M]\!]_q \mid (\!| N |\!)^{\mathsf{ei},\mathsf{eb}}_{q1} \mid \overline{q}?(x).\overline{q1}?(y).(\!| P |\!)^{\mathsf{eb},\mathsf{eo}}_r) && \{\text{forwarding } \mathsf{ei} \to \mathsf{ea}\}
\end{aligned}
$$

$$
\begin{aligned}
&(\!| \mathbf{let}\, y \leftarrow N \,\mathbf{in}\, (\mathbf{let}\, x \leftarrow M \,\mathbf{in}\, P) |\!)^{\mathsf{ei},\mathsf{eo}}_r \\
&\stackrel{\text{def}}{=} \nu\, q, \mathsf{ea}.\, ((\!| N |\!)^{\mathsf{ei},\mathsf{ea}}_q \mid \overline{q}?(y).(\!| \mathbf{let}\, x \leftarrow M \,\mathbf{in}\, P |\!)^{\mathsf{ea},\mathsf{eo}}_r) \\
&\stackrel{\text{def}}{=} \nu\, q, \mathsf{ea}.\, ((\!| N |\!)^{\mathsf{ei},\mathsf{ea}}_q \mid \overline{q}?(y).\nu\, q1, \mathsf{eb}.\, ((\!| M |\!)^{\mathsf{ea},\mathsf{eb}}_{q1} \mid \overline{q1}?(x).(\!| P |\!)^{\mathsf{eb},\mathsf{eo}}_r)) \\
&\approx \nu\, q, \mathsf{ea}, q1, \mathsf{eb}.\, ((\!| N |\!)^{\mathsf{ei},\mathsf{ea}}_q \mid (\!| M |\!)^{\mathsf{ea},\mathsf{eb}}_{q1} \mid \overline{q}?(y).\overline{q1}?(x).(\!| P |\!)^{\mathsf{eb},\mathsf{eo}}_r) && \{\text{sequentiality, } y \notin fv(M)\} \\
&\approx \nu\, q, \mathsf{ea}, q1, \mathsf{eb}.\, ((\!| N |\!)^{\mathsf{ei},\mathsf{ea}}_q \mid \mathsf{ea}?(c).\overline{\mathsf{eb}}!\langle c\rangle.[\![M]\!]_{q1} \mid \overline{q}?(y).\overline{q1}?(x).(\!| P |\!)^{\mathsf{eb},\mathsf{eo}}_r) && \{\text{purity lemma on } M\} \\
&\approx \nu\, q, q1, \mathsf{ea}.\, ((\!| N |\!)^{\mathsf{ei},\mathsf{ea}}_q \mid [\![M]\!]_{q1} \mid \overline{q}?(y).\overline{q1}?(x).(\!| P |\!)^{\mathsf{ea},\mathsf{eo}}_r) && \{\text{forwarding } \mathsf{ea} \to \mathsf{eb}\} \\
&\approx \nu\, q, q1, \mathsf{ea}.\, ([\![M]\!]_{q1} \mid (\!| N |\!)^{\mathsf{ei},\mathsf{ea}}_q \mid \overline{q}?(y).\overline{q1}?(x).(\!| P |\!)^{\mathsf{ea},\mathsf{eo}}_r) && \{\text{structural congruence}\} \\
&\approx \nu\, q, q1, \mathsf{eb}.\, ([\![M]\!]_q \mid (\!| N |\!)^{\mathsf{ei},\mathsf{eb}}_{q1} \mid \overline{q1}?(y).\overline{q}?(x).(\!| P |\!)^{\mathsf{eb},\mathsf{eo}}_r) && \{\alpha, q \leftrightarrow q1, \mathsf{ea} \leftrightarrow \mathsf{eb}\} \\
&\approx \nu\, q, q1, \mathsf{eb}.\, ([\![M]\!]_q \mid (\!| N |\!)^{\mathsf{ei},\mathsf{eb}}_{q1} \mid \overline{q}?(x).\overline{q1}?(y).(\!| P |\!)^{\mathsf{eb},\mathsf{eo}}_r) && \{\text{reorder recv.}\} \\
&\approx (*) \quad \square
\end{aligned}
$$

**Lemma** (Pure encoding) If an effect system has the property that $\forall F, G.(F \bullet G \equiv I) \Rightarrow (F \equiv G \equiv I)$ then, for all $M$, $\Gamma$, $\tau$ it follows that:

$$(\!|\, \Gamma \vdash M : \tau, I \,|\!)_r^{\mathsf{ei},\mathsf{eo}} \;\approx\; \mathsf{ei}?(c).\overline{\mathsf{eo}}!\langle c\rangle.[\![M]\!]_r$$

*Proof.* By induction on the derivation of type-and-effect judgments with a pure effect.

- (var) $\Gamma \vdash v : I$, trivial by the definition of $(\!|-|\!)$.
- (let) $\Gamma \vdash \mathbf{let}\, x \leftarrow M\, \mathbf{in}\, N : \tau, I$, the embedding is:

$$(\!|\, \mathbf{let}\, x \leftarrow M\, \mathbf{in}\, N \,|\!)_r^{\mathsf{ei},\mathsf{eo}} = \nu q, \mathsf{ea}.((\!|\, M \,|\!)_q^{\mathsf{ei},\mathsf{ea}} \mid \overline{q}?(x).(\!|\, N \,|\!)_r^{\mathsf{ea},\mathsf{eo}})$$

  The condition of the lemma on effect systems means $\Gamma \vdash M : \sigma, I$ and $\Gamma, x : \sigma \vdash N : \tau, I$, therefore the inductive hypotheses are that:

$$(\!|\, M \,|\!)_q^{\mathsf{ei},\mathsf{ea}} \approx \mathsf{ei}?(c).\overline{\mathsf{ea}}!\langle c\rangle.[\![M]\!]_q \qquad (\!|\, N \,|\!)_r^{\mathsf{ea},\mathsf{eo}} \approx \mathsf{ea}?(c).\overline{\mathsf{eo}}!\langle c\rangle.[\![N]\!]_r$$

  Therefore:

$$\begin{aligned}
(\!|\, \mathbf{let}\, x \leftarrow M\, \mathbf{in}\, N \,|\!)_r^{\mathsf{ei},\mathsf{eo}} &= \nu q, \mathsf{ea}.((\!|\, M \,|\!)_q^{\mathsf{ei},\mathsf{ea}} \mid \overline{q}?(x).(\!|\, N \,|\!)_r^{\mathsf{ea},\mathsf{eo}}) \\
&= \nu q, \mathsf{ea}.(\mathsf{ei}?(c).\overline{\mathsf{ea}}!\langle c\rangle.[\![M]\!]_q \mid \overline{q}?(x).\mathsf{ea}?(c).\overline{\mathsf{eo}}!\langle c\rangle.[\![N]\!]_r) \\
&\approx \nu q.(\mathsf{ei}?(c).([\![M]\!]_q \mid \overline{q}?(x).\overline{\mathsf{eo}}!\langle c\rangle.[\![N]\!]_r)) \\
&\approx \nu q.(\mathsf{ei}?(c).\overline{\mathsf{eo}}!\langle c\rangle.([\![M]\!]_q \mid \overline{q}?(x).[\![N]\!]_r)) \\
&\approx \mathsf{ei}?(c).\overline{\mathsf{eo}}!\langle c\rangle.[\![\mathbf{let}\, x \leftarrow M\, \mathbf{in}\, N]\!]_r
\end{aligned}$$

- (constant) $\Gamma \vdash C : \tau, I$, trivial by definition of $(\!|-|\!)$.
- (op) $\Gamma \vdash op\, M : \tau, I$, where $\Gamma \vdash M : \sigma, I$, trivial by definition of $(\!|-|\!)$.

□