Effects as Sessions, Sessions as Effects

Dominic Orchard
Imperial College London, UK[1]
d.orchard@imperial.ac.uk

Nobuko Yoshida
Imperial College London, UK
n.yoshida@imperial.ac.uk

Abstract

Effect and session type systems are two expressive behavioural type systems. The former is usually developed in the context of the λ-calculus and its variants, the latter for the π-calculus. In this paper we explore their relative expressive power. Firstly, we give an embedding from PCF, augmented with a parameterised effect system, into a session-typed π-calculus (session calculus), showing that session types are powerful enough to express effects. Secondly, we give a reverse embedding, from the session calculus back into PCF, by instantiating PCF with concurrency primitives and its effect system with a session-like effect algebra; effect systems are powerful enough to express sessions. The embedding of session types into an effect system is leveraged to give a new implementation of session types in Haskell, via an effect system encoding. The correctness of this implementation follows from the second embedding result. We also discuss various extensions to our embeddings.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

Keywords session types, π-calculus, effect systems, PCF, encoding, type systems, Concurrent Haskell

1. Introduction

The simple type theory for the λ-calculus classifies the range of input and output values required by, and provided by, a computation. Various other kinds of type system specify further, describing not just *what* is computed, but *how* values are computed. These might be informally described as *behavioural* type systems, *i.e.* the intensional behaviour of computation is described. In this paper we study the relative expressive power of two such behavioural type systems for two fundamental calculi: *effect types* in the λ-calculus and *session types* in the π-calculus.

Effect types augment standard value-typing to describe side-effect behaviour. They are the type system representation of *effect systems*, a general class of static program analysis for collecting information on effects such as state, exceptions, or resource use [12, 20, 50]. Classes of effect analyses are often defined abstractly via a system parameterised by an algebra of effects such as semi-lattices in early work [20] or semiring and Kleene algebra-like structures later [36, 38]. On the other hand, session types describe and restrict concurrent interactions over channels in the π-calculus, with types for sequencing (via prefixing), branching, and recursion.

The π-calculus and the λ-calculus are simple yet powerful prototypes of computation. But they do not stand apart. The π-calculus subsumes the power of the λ-calculus, and is universal with respect to sequential computation. This result was first proved by Milner [33], opening up the use of λ-calculus encodability to show the expressiveness of various type disciplines for the π-calculi. Such embeddings are applied for type-based analyses of programming languages for, *e.g.*, concurrent abstract data types [45], a tail-call optimisation of functions [31], secrecy [24] and termination [56]. Recently Toninho *et al.* [51] showed that simply typed λ-terms are encodable more tightly into session-typed processes via a Curry-Howard interpretation, providing a new logical explanation of sharing and copying parallel λ-evaluation strategies.

This paper goes further: we show that the session-typed π-calculus is expressive enough to systematically encode various classes of effect system for the λ-calculus (PCF). This is notable since, compared to effect systems, session types are more restrictive, in the sense that they rule out many computations, whereas effect systems tend to be more descriptive.

But this correspondence is not just in one direction. We show that a general effect system can be instantiated to capture session types. Using this encoding, we implement session-typed communications in Concurrent Haskell via a type-level effect system.

Processes as effect handlers, session types as effects The core idea behind the embedding of effects into sessions is the simulation of effectful computations using processes as *effect handlers* (inspired by work on effect semantics via handlers, *e.g.*, [3, 46]). Interactions with these processes, via session-typed channels, induces a description of the effects of a process as a session type.

Consider the following simple recursive process, often called the *variable agent*, used to simulate a single mutable memory cell:

$$\textbf{def}\ \mathit{Var}(c,x) = c \rhd \{\mathsf{get} : c!\langle x\rangle.\mathit{Var}\langle c,x\rangle,\ \mathsf{put} : c?(y).\mathit{Var}\langle c,y\rangle\}$$

A process generated by $\mathit{Var}\langle c,i\rangle$ repeatedly offers on channel c a choice ($c \rhd \{...\}$) of two interaction modes: get and put. If the get branch is chosen, x is sent on c ($c!\langle x\rangle$) then the process recurses with the same parameters. In the put branch, a value y is received on c ($c?(y)$) which becomes the new stored value via the recursive call. The initial stored value is i. Thus, $\mathit{Var}\langle c,i\rangle$ handles *get* and *put* operations performed by a client process.

For example, by interaction with the handler, the following process increments the stored value:

$$c \lhd \mathsf{get}\,.\,c?(x).c \lhd \mathsf{put}\,.\,c!\langle x+1\rangle \ \mid\ \mathit{Var}\langle \overline{c},i\rangle \qquad (1)$$

On channel c the get branch is selected (via $\lhd$), a value x is received, the put branch is selected, then $x+1$ is sent. The session-typing discipline restricts channels to rule-out various unsafe concurrent behaviour. For the left-hand process in (1), a standard session typing system (*e.g.* [22, 57]) might assign to channel c the session type $c : \oplus[\mathsf{get} : ?[\mathbb{Z}].\oplus[\mathsf{put} : ![\mathbb{Z}]]]$. This complements the session type of the variable agent and together their interaction is

[1] At the University of Cambridge, Computer Laboratory from January 2016.

POPL'16, January 20-22, 2016, St. Petersburg, FL, USA.

http://dx.doi.org/10.1145/2837614.2837634

proven type and communication safe. Considering the intention behind the variable agent (simulating state), session types here act as an effect system. The above session type describes the process having side-effects of reading and then writing an integer to memory.

Paper structure and contributions We conjecture that session types and effect systems are equally expressive. We make the following contributions to elucidate their relationship:

- We embed a variant of PCF with a general effect system into a session-typed variant of the π-calculus (*session calculus* for short) (Section 3). We prove this embedding is type-preserving and sound (both operationally and axiomatically) (Section 4).
- The embedding is parametric in various effect-dependent structure. These are instantiated for effects with linear control-flow, including variations on state (list or set-based) and resource usage effects in (Section 3.5 and Section 9.1).
- We extend the encoding to a parallel variant of PCF where effects may interfere (Section 5). This requires only a small extension to the duality predicate of session typing.
- We instantiate PCF with concurrency primitives and an effect system for sessions. We reverse the embedding of Section 3, giving an embedding of the session calculus into this instantiation of PCF (Section 7), showing that effect systems are powerful enough to encode session types. This is leveraged to give a new implementation of session types in Haskell (Section 8), enlarging the typability from previous implementations. This is our *Artifact*, available at `http://dorchard.co.uk/popl16`.

Section 2 introduces the two calculi. Section 9 discusses extensions to our encodings and related work. A companion technical report contains proofs and additional definitions [41].

2. Background: two typed calculi

2.1 PCF with effects

Our source language is an effectful, call-by-value variant of PCF (simply-typed λ-calculus with natural numbers, conditionals, and recursion) which we call *FPCF* (*ef*fectful PCF) with syntax:

$$M, N ::= V \mid M\,N \mid \mathbf{case}\,M\,\mathbf{of}\,0 \mapsto N, (\mathsf{suc}\,x) \mapsto N' \mid C$$
$$V ::= x \mid \lambda x.M \mid \mathbf{rec}\,(\lambda f.\lambda x.M) \mid C_V$$

V denotes *values*, M, N *computations*, and x ranges over variables. C ranges over constants which can be instantiated to give application-specific (possibly effectful) operations. C_V is the subset of C for value constants, which includes the pure zero, successor, and unit constructors $0, \mathsf{suc}, \mathsf{unit} \in C_V$. The **case** syntax pattern matches on natural number constructors.

We define a type-and-effect system for *FPCF* similar in style to the rich effect systems of Nielson and Nielson [38], which differ from traditional effect systems (*e.g.*, [20]) by distinguishing sequential control flow from branching control flow (alternation).

Definition 1 (Effect algebra)**.** Let $\mathcal{F}$ be a set, where F, G, H range over its elements, with partial order $\sqsubseteq$ and with structure:

- Monoid $(\mathcal{F}, \bullet, I)$ where $\bullet$ corresponds to sequential composition and I is the trivial effect for pure computation.
- Commutative semigroup $(\mathcal{F}, \oplus)$ where $\oplus$ corresponds to branching, with distributivity $(F \oplus G) \bullet H = (F \bullet H) \oplus (G \bullet H)$.
- Closure operation $(\mathcal{F}, -^*)$ for effect fixed-points with axioms $F^* = 1 \oplus (F^* \bullet F) = 1 \oplus (F \bullet F^*)$.

For some systems, $\oplus$ is the least-upper bound with respect to $\sqsubseteq$.

Definition 2 (Types and effects)**.** The type-and-effect system for *FPCF* has judgements of the form $\Gamma \vdash M : \tau, F$ meaning that term M has type τ in the context Γ of free-variable typing assumptions and performs effect F. The syntax of types is defined:

$$\sigma, \tau ::= \sigma \xrightarrow{F} \tau \mid \mathbf{nat} \mid \mathbf{unit}$$

where F is an effect annotation for the *latent effect* of a function.

$$\text{var}\ \frac{}{\Gamma \vdash x : \tau, I} \quad \text{abs}\ \frac{\Gamma, x : \sigma \vdash M : \tau, F}{\Gamma \vdash \lambda x.M : \sigma \xrightarrow{F} \tau, I}$$

$$\text{app}\ \frac{\Gamma \vdash M : \sigma \xrightarrow{H} \tau, F \quad \Gamma \vdash N : \sigma, G}{\Gamma \vdash M\,N : \tau, F \bullet G \bullet H} \quad \text{const}\ \frac{}{\emptyset \vdash C : C_\tau, C_F}$$

$$\text{case}\ \frac{\Gamma \vdash M : \mathbf{nat}, F \quad \Gamma \vdash N_1 : \tau, G \quad \Gamma, x : \mathbf{nat} \vdash N_2 : \tau, H}{\Gamma \vdash \mathbf{case}\,M\,\mathbf{of}\,0 \mapsto N_1,\ (\mathsf{suc}\,x) \mapsto N_2 : \tau, F \bullet (G \oplus H)}$$

$$\text{sub}\ \frac{\Gamma \vdash M : \tau, F \quad F \sqsubseteq G}{\Gamma \vdash M : \tau, G} \quad \text{rec}\ \frac{\Gamma, f : \tau \xrightarrow{F} \tau, x : \tau \vdash M : \tau, F}{\Gamma \vdash \mathbf{rec}\,(\lambda f.\lambda x.M) : \tau \xrightarrow{F} \tau, I}$$

Figure 1. Type-and-effect system for *FPCF*

Figure 1 gives the type-and-effect rules. The (var) and (abs) rules describe variable use and abstraction as pure (with I). For (abs), the effect of the function body F becomes a latent effect. The (sub) rule allows effects to be overapproximated (with respect to the partial order $\sqsubseteq$). The (app) rule exposes the left-to-right evaluation order of (call-by-value) application by the composition order of the effect F of the function term M followed by effect G of the argument term and then H of the function body. The (const) rule introduces a constant of type C_τ with effects C_F. The effect of (case) sequentially composes the effect F of the matched expression M with the branching composition $G \oplus H$. The effect of a recursive binding (rec) is the same as that of its body and recursive call. The closure operator $(-)^*$ is often required to provide a valid typing for a recursive definition.

We include *let*-binding as syntactic sugar $(\mathbf{let}\,x = M\,\mathbf{in}\,N) := (\lambda x.N)\,M$, with the (let) rule typing.

$$\text{let}\ \frac{\Gamma \vdash M : \sigma, F \quad \Gamma, x : \sigma \vdash N : \tau, G}{\Gamma \vdash \mathbf{let}\,x = M\,\mathbf{in}\,N : \tau, F \bullet G}$$

Example 1 (Simple causal state)**.** Let $E = \{\mathsf{rd}\,\tau, \mathsf{wr}\,\tau \mid \forall \tau\}$ be a set of symbols tagged by types and $[E]$ lists of E (with $[\,]$ nil and $::$ cons operators). Let $\mathcal{F}$ be the set generated by $\mathcal{F} := [E] \mid \mathcal{F} + \mathcal{F}$ with an effect algebra where $\oplus = +$, $I = [\,]$, and $\bullet$ as list concatenation which distributes with $+$, defined:

$$[\,] \bullet f = f \quad (e :: f) \bullet g = e :: (f \bullet g) \quad (f + g) \bullet h = (f \bullet h) + (g \bullet h)$$

We ignore fixed-points for now and assume the effect algebra axioms up to isomorphism (*e.g.* $F + (G + H) \cong (F + G) + H$). Constants C are extended with get and put, typed:

$$\emptyset \vdash get : \tau, [\mathsf{rd}\,\tau] \qquad \emptyset \vdash put : \tau \xrightarrow{[\mathsf{wr}\,\tau]} \mathbf{unit}, [\,] \qquad (2)$$

For example, $\emptyset \vdash (\lambda x.put\,(\mathsf{suc}\,x))\,get : \mathbf{nat}, [\mathsf{rd}\,\mathbf{nat}, \mathsf{wr}\,\mathbf{nat}]$ is a valid judgement. Type safety of the store requires that *read* effects must have the same type as their nearest preceding *write* effect.

Definition 3 ($\beta\eta$-equality)**.** Let $\equiv$ be an equivalence relation over terms with (call-by-value) β-equations:

$$\begin{array}{rl}
(\beta) & (\lambda x.M)\,V \equiv M[V/x] \\
(\mathbf{rec}\beta) & \mathbf{rec}\,(\lambda f.\lambda x.M) \equiv \lambda x.M[\mathbf{rec}\,(\lambda f.\lambda x.M)/f] \\
(\mathbf{case}\beta 1) & (\mathbf{case}\,0\,\mathbf{of}\,0 \mapsto M, (\mathsf{suc}\,x) \mapsto N) \equiv M \\
(\mathbf{case}\beta 2) & (\mathbf{case}\,(\mathsf{suc}\,V)\,\mathbf{of}\,0 \mapsto M, (\mathsf{suc}\,x) \mapsto N) \equiv N[V/x]
\end{array}$$

We add two further equations on **let**:

$$\begin{array}{rl}
\text{(let-id)} & (\mathbf{let}\,x = M\,\mathbf{in}\,x) \equiv M \\
\text{(let-assoc)} & \mathbf{let}\,y = (\mathbf{let}\,x = M\,\mathbf{in}\,N)\,\mathbf{in}\,N' \\
& \equiv \mathbf{let}\,x = M\,\mathbf{in}\,(\mathbf{let}\,y = N\,\mathbf{in}\,N') \quad (\mathit{if}\ x \notin \mathsf{fv}(N'))
\end{array}$$

Note, (β) above is equivalent to ($\mathbf{let}\, x = V\, \mathbf{in}\, M) \equiv M[V/x]$. Equality extends to type-and-effect judgements $\Gamma \vdash M \equiv N : \tau, F$ where $\Gamma \vdash M : \tau, F$ and $\Gamma \vdash N : \tau, F$. In the usual way, η-equality is type-dependent: $M \equiv \lambda x.M\, x$ holds only when M is a function type. Further, in *FPCF* η-equality holds only when M is also pure. That is, $\Gamma \vdash M \equiv (\lambda x.M\, x) : \sigma \xrightarrow{F} \tau, I$.

Definition 4 (Operational semantics)**.** A parameterised operational semantics is defined by a relation $\rightarrow$ of *reductions* between effect-specific configurations $\mathbb{C}$. Maps from terms to configurations are written $\mathcal{C}, \mathcal{D} : M \rightarrow \mathbb{C}$.

The four β-equality laws above are oriented left-to-right into pure reductions where a configuration is unchanged, *e.g.* $\forall \mathcal{C}\ \mathcal{C}[(\lambda x.M)\, V] \rightarrow \mathcal{C}[M[V/x]]$. The usual inductive rules provide the rest of the CBV operational semantics.

Proposition 1 (Subject reduction on *FPCF*)**.**

$$\emptyset \vdash M : \tau, F \ \wedge\ \mathcal{C}[M] \rightarrow \mathcal{D}[N] \Rightarrow\ \emptyset \vdash N : \tau, G \ \wedge\ G \sqsubseteq F$$

2.2 Session calculus

We consider the π-calculus with session primitives, which we call the *session calculus*. Figure 2 shows the syntax, where l ranges over labels, $\tilde{l} : \tilde{P}$ over sequences of label-process pairs. The calculus is based on the second system in [57] using the dual channels from [35] instead of polarity. We define the dual operation over channels c as $\overline{c}$ with $\overline{\overline{c}} = c$. Intuitively, names c and $\overline{c}$ are two dual *endpoints*. Throughout we elide trailing occurrences of $\mathbf{0}$ and **end**, *e.g.*, writing $r!\langle x\rangle$ instead of $r!\langle x\rangle.\mathbf{0}$.

Definition 5 (Operational semantics)**.** The reduction relation $\rightarrow$ contains β-laws for send/receive, branch/select, and $\mathbf{if}$:

$$\begin{array}{rl}
(\beta) & c?(x).P \mid \overline{c}!\langle V\rangle.Q \rightarrow P[V/x] \mid Q \\
(\beta\text{-chan}) & c?(d).P \mid \overline{c}!\langle c_0\rangle.Q \rightarrow P[c_0/d] \mid Q \\
(\beta\text{-}*) & *c?(d).P \mid \overline{c}!\langle c_0\rangle.Q \rightarrow *c?(d).P \mid P[c_0/d] \mid Q \\
(\beta\text{-}\lhd\rhd) & c \rhd l_i.P \mid \overline{c} \lhd \{\tilde{l} : \tilde{Q}\} \rightarrow P \mid Q_i \quad (l_i \in \tilde{l}) \\
(\text{res}) & P \rightarrow P' \ \Rightarrow\ \nu c.P \rightarrow \nu c.P' \\
(\text{par}) & P \rightarrow P' \ \Rightarrow\ P \mid Q \rightarrow P' \mid Q \\
(\text{str}) & Q \equiv P \rightarrow P' \equiv Q' \ \Rightarrow\ Q \rightarrow Q' \\
(\text{if1}) & \mathbf{if}[0 = 0]\mathbf{then}\, P\, \mathbf{else}\, Q \rightarrow P \\
(\text{if2}) & \mathbf{if}[\mathsf{suc}\, V = 0]\mathbf{then}\, P\, \mathbf{else}\, Q \rightarrow Q
\end{array}$$

Equation (β) reduces complementary send/receive actions of a value V over a linear channel c; similarly equation (β-chan) gives the interaction of sending/receiving a linear channel d over a linear channel c. Equation (β-$*$) resembles equation (β-chan) but involves *replicated input* over channel c, that is, c is not linear in the usual sense, but can repeatedly receive values via the persistence of $*c?(d).P$. Equation (β-$\lhd\rhd$) gives the reduction of complementary branching/selection. Reduction is congruent with respect to parallel composition (par) and restriction (res). Furthermore, (str) extends $\rightarrow$ along the *structural congruence* relation $\equiv$ [57], which provides that $\mid$ is a commutative monoid with $\mathbf{0}$ unit, amongst other things.

Definition 6 (Session types [4, 57])**.** Session types record sequences of typed *send* ($![\tau]$) and *receive* ($?[\tau]$) interactions, terminated by **end**, branched by *select* ($\oplus$) and *choice* ($\&$) interactions, *output* ($*![\tau]$), and with recursive types $\mu\alpha.S$:

$$S ::= ![\tau].S \mid ?[\tau].S \mid *![\tau].S \mid \mathbf{end} \mid \oplus[\tilde{l} : \tilde{S}] \mid \&[\tilde{l} : \tilde{S}] \mid \mu\alpha.S \mid \alpha$$

where τ ranges over value types, including session types (Figure 2), l ranges over labels, $\tilde{l} : \tilde{S}$ is a sequence of label-session type pairs, and recursive types obey the fixed-point law $\mu\alpha.S = S[\mu\alpha.S/\alpha]$. We assume recursive types are guarded and *carried types* (*e.g.*, τ in $![\tau].S$) are closed; **end** is often omitted. We define the alias $*?[\tau] := \mu\alpha.?[\tau].\alpha$ for *replicated input*.

$$\begin{array}{lll}
\text{(channels)} & c, \overline{c}, d, \overline{d}, p, \overline{p}, q, \overline{q}, r, \overline{r} & \text{(variables)}\quad v ::= x, y, z \\
\text{(values)} & V ::= 0 \mid \mathsf{suc}\, V \mid \mathsf{pred}\, V \mid \mathsf{unit} \mid v & \text{constants / variables} \\
\text{(processes)} & P, Q ::= c?(x).P \mid c!\langle V\rangle.P & \text{receive / send} \\
& \mid c?(d).P \mid c!\langle d\rangle.P & \text{channel receive / send} \\
& \mid c \rhd \{\tilde{l} : \tilde{P}\} \mid c \lhd l.P & \text{branch / select} \\
& \mid *c?(d).P \mid \mathbf{0} & \text{replicated input / nil} \\
& \mid \mathbf{if}\ [V = 0]\ \mathbf{then}\ P\ \mathbf{else}\ Q & \text{conditionals} \\
& \mid \nu c.P \mid (P \mid Q) & \text{restrict / parallel} \\
\text{(value types)} & \tau ::= \mathbf{unit} \mid \mathbf{nat} \mid S & \text{(contexts)}\quad \Gamma ::= \emptyset \mid \Gamma, x : \tau
\end{array}$$

Figure 2. Syntax of the session calculus

The output type $*![\tau]$ differs from the send type $![\tau]$ since the former interacts only with replicated input: messages on a channel typed $*![\tau]$ may occur in multiple processes. On the other hand, $![\tau]$ is a *linear* send (appears exactly once) [4, 15]. This distinction is essential to encode *FPCF* as well as its parallel extension.

For a session S, its *dual* $\overline{S}$ is defined:

$$\begin{array}{c}
\overline{![\tau].S} = ?[\tau].\overline{S} \quad \overline{?[\tau].S} = ![\tau].\overline{S} \quad \overline{*![\tau]} = *?[\tau] \quad \overline{*?[\tau]} = *![\tau] \\
\overline{\oplus[l_1 : S_1, ..., l_n : S_n]} = \&[l_1 : \overline{S_1}, ..., l_n : \overline{S_n}] \quad \overline{\mu\alpha.S} = \mu\alpha.\overline{S} \\
\overline{\&[l_1 : S_1, ..., l_n : S_n]} = \oplus[l_1 : \overline{S_1}, ..., l_n : \overline{S_n}] \quad \overline{\mathbf{end}} = \mathbf{end}
\end{array}$$

The predicate $*\mathsf{able}(\Delta)$ classifies session types in the environment which comprise only replicated outputs or **end**.

$$\begin{array}{l}
\mathsf{able}(\Delta) \Leftrightarrow \forall (c : S) \in \Delta.\mathsf{able}(S) \\
*\mathsf{able}(S) \Leftrightarrow (\exists T.\ S = *![\tau].T \wedge *\mathsf{able}(T)) \vee (S = \mathbf{end})
\end{array}$$

Figure 3 gives the full session typing system used in this work which is a combination of [4, 57]. Judgements are of the form $\Gamma; \Delta \vdash P$ where Γ is a mapping from variables to base types and Δ is a mapping from channels to session types. In (par), we check two processes are composable or not by $\odot$ (see below); (chan-send) follows the same condition where composability of d is checked. Since $*c?(d).P$ replicates P, it cannot contain free linear session names, guaranteed by $*\mathsf{able}(\Delta)$ in ($*$recv). Note that replicated outputs can be weakened (weak$*$). The rest is standard from [57] with subtyping as in [10, 11, 19].

The (select) rule introduces a selection type with only one label and branch. Duality, which must have corresponding labels for branches and selection, is thus achieved by using subtyping (sub) to extend select types with extra labels.

Definition 7 (Balanced)**.** Two processes must have *balanced* session environments to be composed in parallel, defined by the following symmetric $\bowtie$ relation:

$$(c\!:\!S) \bowtie (\overline{c}\!:\!\overline{S}) \quad (c\!:\!*![\tau].S) \bowtie (c\!:\!*![\tau].S) \quad (c\!:\!S) \bowtie (d\!:\!T)$$

where $c \neq d$. Thus, dual names must have dual types and channels may only appear in two different processes if they are both outputs ($*!$). A partial commutative function $\odot$ takes the union of two environments if they are balanced, *i.e.* $\Delta_1 \odot \Delta_2 = \Delta_1 \cup \Delta_2$ *if* $\Delta_1 \bowtie \Delta_2$ (with $\bowtie$ lifted to all pairs of named session types $\Delta_1 \times \Delta_2$) otherwise it is undefined. Note, $\mu\alpha.?[\tau].\alpha$ is dual to both $\mu\alpha.![\tau].\alpha$ and $*![\tau]$, but $(c : \mu\alpha.![\tau].\alpha) \odot (c : *![\tau])$ is undefined.

Since a session environment represents forthcoming communications, during process interactions, the session environment will change. We define the relation $\Delta \longrightarrow \Delta'$ [25] as follows:

$$\begin{array}{c}
\Delta, c : \&[\tilde{l} : \tilde{S}], \overline{c} : \oplus[\tilde{l} : \overline{\tilde{S}}] \longrightarrow \Delta, c : S_i, \overline{c} : \overline{S}_i \\
\Delta, c : ![\tau].S, \overline{c} : ?[\tau].T \longrightarrow \Delta, c : S, \overline{c} : T \\
\Delta, c : *![\tau].S, \overline{c} : *?[\tau] \longrightarrow \Delta, c : S, \overline{c} : *?[\tau]
\end{array}$$

Proposition 2 (Subject reduction, [4, 57])**.** Suppose $\Gamma; \Delta \vdash P$ and $P \rightarrow^* Q$. Then $\Gamma; \Delta' \vdash Q$ with $\Delta \rightarrow^* \Delta'$. In addition, if Δ is (self-)balanced, then Δ' is balanced.

$$\boxed{\Gamma \vdash V : \tau} \ (\textit{values}) \quad \text{var}\ \frac{v : \tau \in \Gamma}{\Gamma \vdash v : \tau} \quad \text{unit}\ \frac{}{\Gamma \vdash \mathsf{unit} : \mathbf{unit}} \quad \text{zero}\ \frac{}{\Gamma \vdash 0 : \mathbf{nat}} \quad \text{suc}\ \frac{\Gamma \vdash V : \mathbf{nat}}{\Gamma \vdash \mathsf{suc}\, V : \mathbf{nat}} \quad \text{pred}\ \frac{\Gamma \vdash V : \mathbf{nat}}{\Gamma \vdash \mathsf{pred}\, V : \mathbf{nat}}$$

$$\boxed{\Gamma; \Delta \vdash P} \quad (\textit{processes}) \quad \text{weak}*\ \frac{\Gamma; \Delta \vdash P}{\Gamma; \Delta, c : *![S] \vdash P} \quad \text{end}\ \ \Gamma; \tilde{c} : \mathbf{end} \vdash \mathbf{0} \quad \text{par}\ \frac{\Gamma; \Delta_i \vdash P_i}{\Gamma; \Delta_1 \odot \Delta_2 \vdash P_1 \,|\, P_2} \quad \text{restr}\ \frac{\Gamma; \Delta, c : S, \overline{c} : \overline{S} \vdash P}{\Gamma; \Delta \vdash \nu c.P}$$

$$\text{recv}\ \frac{\Gamma, x : \tau; \Delta, c : S \vdash P}{\Gamma; \Delta, c : ?[\tau].S \vdash c?(x).P} \quad \text{chan-recv}\ \frac{\Gamma; \Delta, c : T, d : S \vdash P}{\Gamma; \Delta, c : ?[S].T \vdash c?(d).P} \quad *\text{recv}\ \frac{\Gamma; \Delta, d : S \vdash P \quad *\mathsf{able}(\Delta)}{\Gamma; \Delta, c : *?[S] \vdash *c?(d).P}$$

$$\text{send}\ \frac{\Gamma \vdash V : \tau \quad \Gamma; \Delta, c : S \vdash P}{\Gamma; \Delta, c : ![\tau].S \vdash c!\langle V \rangle.P} \quad \text{chan-send}\ \frac{\Gamma; \Delta, c : T \vdash P}{\Gamma; (\Delta, c : ![S].T) \odot d : S \vdash c!\langle d \rangle.P} \quad *\text{send}\ \frac{\Gamma; \Delta, c : T \vdash P}{\Gamma; (\Delta, c : *![S].T) \odot d : S \vdash P \vdash c!\langle d \rangle.P}$$

$$\text{if}\ \frac{\Gamma; \Delta \vdash P_i \quad \Gamma \vdash V : \mathbf{nat}}{\Gamma; \Delta \vdash \mathbf{if}[V = 0]\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2} \quad \text{branch}\ \frac{\Gamma; \Delta, c : S_i \vdash P_i}{\Gamma; \Delta, c : \&[\tilde{l} : \tilde{S}] \vdash c \rhd \{\tilde{l} : \tilde{P}\}} \quad \text{select}\ \frac{\Gamma; \Delta, c : S\ \vdash P}{\Gamma; \Delta, c : \oplus[l : S] \vdash c \lhd l.P} \quad \text{sub}\ \frac{\Gamma; \Delta \vdash P \quad \Delta <: \Delta'}{\Gamma; \Delta' \vdash P}$$

Figure 3. Session typing relation over the π-calculus with session primitives.

3. Effects as sessions: *FPCF* into session calculus

Our encoding is based on the encoding of the pure call-by-value λ-calculus into the π-calculus [33, 49] and is *type directed*, mapping *FPCF* derivations to session-type derivations. We define an overloaded embedding function $[\![-]\!]$ mapping terms, types, effects, contexts, and judgements of *FPCF* to session calculus constructs.

Key idea: effect channel carriers Side effects are modelled by interactions with an *effect handler* process (*e.g.*, the variable agent of the introduction) over a channel which we call the *effect channel*. Encoded *FPCF* terms have two free channels which we call *effect channel carriers*, one which receives the effect channel (incoming) and one that sends the effect channel after it has been used (outgoing). To see why this is needed, rather than just using the effect channel directly, consider a standard encoding for pure *let*-binding

$$[\![\mathbf{let}\, x = M\, \mathbf{in}\, N]\!]_r = \nu q.([\![M]\!]_q \mid \overline{q}?(x).[\![N]\!]_r)$$

The subscript on encodings $[\![-]\!]_r$ specifies the channel r over which a result is sent. Subterms are inductively encoded, with a fresh channel q passing the result from $[\![M]\!]$ to be bound in the scope of $[\![N]\!]$. If each encoded subterm were to simulate side effects by interacting with a handler via a channel c, then the encoding would not be well-typed; encodings of $[\![M]\!]$ and $[\![N]\!]$ would have $c : S$ and $c : T$ respectively which may be different. The *balanced* predicate in the (par) typing rule prevents this composition. Instead, our encoding has an intermediate form parameterised by effect carriers, written $(\!|-|\!)_r^{\mathsf{ei},\mathsf{eo}}$ for an incoming effect channel carrier ei and outgoing carrier eo. The *let*-binding encoding is then:

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

where ea is an intermediate carrier between M and N. This approach resembles a continuation-passing style semantics or "threading" a store in an operational semantics, $\langle e, s\rangle \to \langle e', s'\rangle$. We previously gave a similar encoding for a first-order imperative language into session types [42] (see Section 9.3 on related work).

Key steps for encoding FPCF The design of *FPCF* is general, but parametric in its effect algebra and constants. Our encoding is therefore similarly generic but parameterised, so that it can be systematically instantiated to embed different notions of effect.

Definition 8. *Effect-encoding parameters* comprise:

1. an *effect handler* process $H(\mathit{eff})$ parameterised by eff, typed:
$$\textit{for some } S \qquad \emptyset; \mathit{eff} : \mu\alpha.S \vdash H(\mathit{eff})$$
2. a *terminator* process $T(\overline{\mathit{eff}})$ such that $T(\overline{\mathit{eff}}) \mid H(\mathit{eff}) \to^* \mathbf{0}$
3. an interpretation functor $[\![-]\!] : \mathcal{F} \to S$ from effect algebra elements to session types, satisfying the following homomorphism property (discussed more in Section 3.4) where $\blacklozenge$ is a (partial) sequential composition for session types:
$$[\![I]\!] = \mathbf{end} \qquad [\![F \bullet G]\!] = [\![F]\!] \blacklozenge [\![G]\!]$$
4. an encoding $(\!|-|\!)_r^{\mathsf{ei},\mathsf{eo}}$ for all constants C satisfying that, $\exists P$:
$$\forall g.\ (\!|\, \emptyset \vdash C : C_\tau, C_F \,|\!)_r^{\mathsf{ei},\mathsf{eo}} = r :![\![C_\tau]\!], \mathsf{ei} :?[\![C_F \bullet g]\!], \overline{\mathsf{eo}} :![\![g]\!] \vdash P$$
5. an encoding for $\mathbf{case}/\oplus$ and (relatedly) for the (sub) rule since the semantics of conditionals and subeffecting is effect dependent (we however defined a general encoding for a restricted form of $\mathbf{case}$ in Section 3.2 and a general construction for free upper bounds in Section 3.3).

In session-type prefixes, we omit the brackets when an interpretation is inside, *e.g.* $![\![\tau]\!]$ instead of $![[\![\tau]\!]]$.

The ***top-level embedding*** is defined in terms of an intermediate (equation (4) below) and the above parameters:

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

where $[\![\Gamma]\!]^\circ$ and $[\![\Gamma]\!]$ map environments to values and session environments respectively, and $[\![\tau]\!]$ maps *FPCF* types τ to session types (defined below). The intermediate encoding is composed in parallel with a process which sends effect channel *eff* on $\overline{\mathsf{ei}}$ and receives a channel on eo (which is similarly named *eff* as this ends up being the same channel) before completing with $T(\mathit{eff})$.

Encoding types Ground types of *FPCF* are mapped to corresponding values types of the session calculus $[\![\mathbf{nat}]\!] = \mathbf{nat}$ and $[\![\mathbf{unit}]\!] = \mathbf{unit}$. Function types are interpreted as session types, where $\forall g$ at the meta level:

$$[\![\sigma \xrightarrow{F} \tau]\!] = *![?[\![\sigma]\!], ?[\![F \bullet g]\!], ![\![g]\!], ![\![\tau]\!]]$$

using a polyadic variant of sending (the extension is straightforward). Function types are interpreted as session types for channels sending: (1) a channel to receive a $[\![\sigma]\!]$ value for the argument, (2) an effect carrier to receive an effect channel capable of simulating effects $F \bullet g$ (3) a carrier to send an effect channel capable of effects g, and (4) a channel to send a $[\![\tau]\!]$ value for the result.

Free-variable contexts of *FPCF* are interpreted into a value-variable context (for ground types), written $[\![-]\!]^\circ$, and a session-variable context for functions, written $[\![-]\!]$:

$$\begin{array}{ll} [\![\emptyset]\!]^\circ = \emptyset & [\![\Gamma, x : T]\!]^\circ = [\![\Gamma]\!]^\circ, x : [\![T]\!] \\ [\![\emptyset]\!] = \emptyset & [\![\Gamma, x : \sigma \xrightarrow{F} \tau]\!] = [\![\Gamma]\!], x : [\![\sigma \xrightarrow{F} \tau]\!] \end{array} \tag{3}$$

Terms/derivations The core embedding is the intermediate $(\!|-|\!)_r^{\mathsf{ei},\mathsf{eo}}$ from *FPCF* derivations to session-calculus derivations, of the form:

$$\begin{array}{l} (\!|\, \Gamma \vdash M : \tau, F \,|\!)_r^{\mathsf{ei},\mathsf{eo}} \\ \quad = \forall g.\ [\![\Gamma]\!]^\circ;\ [\![\Gamma]\!], r :![\![\tau]\!], \mathsf{ei} :?[\![F \bullet g]\!], \overline{\mathsf{eo}} :![\![g]\!] \vdash P \end{array} \tag{4}$$

where P is the encoded term. The incoming carrier ei receives an effect channel of type $[\![F \bullet g]\!]$ (capable of carrying out effect interactions $F \bullet g$) and $\overline{\mathsf{eo}}$ sends an effect channel of type $[\![g]\!]$ (can simulate effect interactions g) where g is universally quantified at the meta level (similarly in function types). The following partial type derivation shows, for the above encoding of **let**, how the universally quantified meta variables (in red) are instantiated to support sequential composition:

$$\dfrac{\begin{array}{l}\mathsf{ei} :?[\![F \bullet g]\!], \overline{\mathsf{ea}} :![\![g]\!] \vdash (\!| M |\!)_q^{\mathsf{ei},\mathsf{ea}} \\ \mathsf{ea} :?[\![G \bullet h]\!], \overline{\mathsf{eo}} :![\![h]\!] \vdash \overline{q}?(x).(\!| N |\!)_r^{\mathsf{ea},\mathsf{eo}}\end{array}}{\begin{array}{l}\mathsf{ei} :?[\![F \bullet g]\!], \overline{\mathsf{ea}} :![\![g]\!], \\ \mathsf{ea} :?[\![G \bullet h]\!], \overline{\mathsf{eo}} :![\![h]\!]\end{array} \vdash (\!| M |\!)_q^{\mathsf{ei},\mathsf{ea}} \mid \overline{q}?(x).(\!| N |\!)_r^{\mathsf{ea},\mathsf{eo}}}\ g \mapsto [\![G \bullet h]\!]$$

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

We show first the encoding of the λ-calculus subset of *FPCF*. We elide types where possible for brevity.

3.1 λ-calculus and natural numbers

Variables are pure and therefore receive and send the effect channel c without use, and simply send the variable over r. That is,

$$(\!| x |\!)_r^{\mathsf{ei},\mathsf{eo}} = \mathsf{ei}?(c).r!\langle x\rangle.\overline{\mathsf{eo}}!\langle c\rangle$$

The typing of the effect carrier channels is $\forall g.\mathsf{ei} :?[\![g]\!], \overline{\mathsf{eo}} :![\![g]\!]$ revealing the pure nature of variables.

As our encoding is type directed, π-calculus terms can be overloaded on whether the encoding is on ground or function types. If the variable's type τ is a function type then x is a channel variable and $r!\langle x\rangle$ is typed by (chan-send) (Figure 3). If τ is a ground type then x is a value variable and $r!\langle x\rangle$ is typed by (send). The rest of the encoding has similar overloading for either channels or values.

Abstraction is similarly pure, hence an effect channel is received and sent without any use. The encoding is defined:

$$\begin{array}{l}(\!| \lambda x.M |\!)_r^{\mathsf{ei},\mathsf{eo}} = \\ \quad \nu d.\,(\mathsf{ei}?(c).r!\langle d\rangle.\overline{\mathsf{eo}}!\langle c\rangle.{*}\overline{d}?(p,\mathsf{ea},\mathsf{eb},q).p?(x).(\!| M |\!)_q^{\mathsf{ea},\mathsf{eb}})\end{array}$$

The new channel endpoint d is sent on the result channel r, then the opposite endpoint $\overline{d}$ receives four channels needed for M, the function body: p receives the argument x, ea receives the incoming effect channel, eb sends the outgoing effect channel, and q sends the result. Replicated input is used as a (bound) function value may be called multiple times.

Application encoding comprises a function and argument: the left-hand side (function) encoding uses fresh channels q and ea to send the resulting session simulating functions and the outgoing effect channel respectively. The right-hand side (argument) receives the effect channel from the left-hand side on ea and uses fresh channels s and eb to send the result and outgoing effect channel respectively.

$$\begin{array}{l}(\!| MN |\!)_r^{\mathsf{ei},\mathsf{eo}} = \nu q, s, \mathsf{ea}, \mathsf{eb}, p. \\ \quad ((\!| M |\!)_q^{\mathsf{ei},\mathsf{ea}} \mid (\!| N |\!)_s^{\mathsf{ea},\mathsf{eb}} \mid \overline{q}?(y).\overline{s}?(x).y!\langle p, \mathsf{eb}, \overline{\mathsf{eo}}, r\rangle.\overline{p}!\langle x\rangle)\end{array}$$

The result of the function part M is a channel y over which is sent the channel p for receiving the argument, the channel eb for receiving the incoming effect channel, channel $\overline{\mathsf{eo}}$ for sending the outgoing effect channel, and r to send the result of the function.

Natural number constructors are encoded as follows, where 0 resembles the variable encoding (since it is a pure constant) and suc resembles the λ-abstraction encoding, since suc is a function:

$$(\!| 0 |\!)_r^{\mathsf{ei},\mathsf{eo}} = \mathsf{ei}?(c).r!\langle 0\rangle.\overline{\mathsf{eo}}\langle c\rangle \tag{5}$$

$$\begin{array}{l}(\!| \mathsf{suc} |\!)_r^{\mathsf{ei},\mathsf{eo}} = \nu d.(\mathsf{ei}?(c).r!\langle d\rangle.\overline{\mathsf{eo}}!\langle c\rangle.{*}\overline{d}?(p,\mathsf{ea},\mathsf{eb},q). \\ \qquad\qquad p?(x).\mathsf{ea}?(c).q!\langle \mathsf{suc}\, x\rangle.\overline{\mathsf{eb}}!\langle c\rangle)\end{array} \tag{6}$$

The encoding of unit is similar to 0, modulo the value constructor.

3.2 Control flow: conditionals and fixed-points

The above encodes the λ-calculus subset of *FPCF*, giving the sequential composition of effects. We now encode the control-flow operators which PCF adds to the λ-calculus: **case** and recursion.

Conditionals Our type-and-effect system for *FPCF* defines the effects of **case** as $F \bullet (G \oplus H)$ for effects F of the guard and G and H of the zero and successor branches. This provides a general characterisation of the control-flow, allowing various kinds of data flow analysis including *may* and *must* analyses. The encoding of $\oplus$ is thus dependent on the notion of effect and so we cannot give a general encoding (work on effect control-flow algebras elucidates this [36]). We can however encode a restricted version of **case**.

Traditional set-based effect systems often provide rules for **case** which either have the same effect in each branch or take the union (least-upper bound) of the branches (*i.e.*, $\oplus = \cup$ in our calculus). These two approaches are equivalent since subeffecting can be used to get the same (upper bound) effect for each branch of a **case**. Consider the following alternate type-and-effect rule:

$$\frac{\Gamma \vdash M : \mathbf{nat}, F \quad \Gamma \vdash N_1 : \tau, G \quad \Gamma, x : \mathbf{nat} \vdash N_2 : \tau, G}{\Gamma \vdash \mathbf{case}\, M\, \mathbf{of}\, 0 \mapsto N_1,\, (\mathsf{suc}\, x) \mapsto N_2 : \tau, G} \tag{7}$$

Given $\Gamma \vdash N_1 : \tau : G_1$ and $\Gamma \vdash N_2 : \tau : G_2$ then the above rule is equivalent to the previous with $\oplus = \sqcup$ (least-upper bound, if it exists) such that $G_1 \sqsubseteq G$ and $G_2 \sqsubseteq G$ with $G = G_1 \sqcup G_2$. Then subeffecting can be used to match the premises of the above rule. This provides a *may*-style analysis. We embed the above restricted **case** as it provides a general encoding (when $\oplus$ is idempotent):

Definition 9 (Restricted case)**.** The **case** rule (7) is encoded:

$$\begin{array}{l}(\!| \mathbf{case}\, M\, \mathbf{of}\, 0 \mapsto N_1, (\mathsf{suc}\, x) \mapsto N_2 |\!)_r^{\mathsf{ei},\mathsf{eo}} = \nu \mathsf{ea}, q.((\!| M |\!)_q^{\mathsf{ei},\mathsf{ea}} \mid \\ \overline{q}?(x).\mathbf{if}\ [x = 0]\ \mathbf{then}\ (\!| N_1 |\!)_r^{\mathsf{ea},\mathsf{eo}}\ \mathbf{else}\ \mathit{pred}\langle x\rangle(x).(\!| N_2 |\!)_r^{\mathsf{ea},\mathsf{eo}})\end{array}$$

where $\mathit{pred}\langle V\rangle(x).P = \nu c.(c!\langle \mathsf{pred}\, V\rangle \mid \overline{c}?(x).P)$ is syntactic sugar for performing the predecessor operation on a natural number and binding it. Thus, we receive the result of the guard M and bind it to x, which parameterises a conditional process which either continues with the N_1 encoding or the N_2 encoding. By the typing of **if**, each branch must have the same session types, thus the effect of N_1 must equal that of N_2.

A more fine-grained encoding provides an encoding for a *free* representation of $\oplus$ via subeffecting, which places additional requirements on the handler. This is shown in Section 3.3.

Recursion The embedding of recursion is very similar to the embedding for abstraction, where the replication inherent in the encoding is utilised for the recursive behaviour:

$$\begin{array}{l}(\!| \mathbf{rec}\, (\lambda f.\lambda x.M) |\!)_r^{\mathsf{ei},\mathsf{eo}} = \\ \nu d.(\mathsf{ei}?(c).r!\langle d\rangle.\overline{\mathsf{eo}}!\langle c\rangle.{*}\overline{d}?(p,\mathsf{ea},\mathsf{eb},q).p?(x).(\!| M |\!)_q^{\mathsf{ea},\mathsf{eb}}[d/f])\end{array}$$

The key difference between this and the abstraction encoding is the syntactic substitution $[d/f]$ of f with the channel d. Thus, if f is free in M then an intermediate encoding $(\!| f |\!)_s^{\mathsf{ec},\mathsf{ed}}[d/f] = \mathsf{ec}?(c).s!\langle d\rangle.\overline{\mathsf{ed}}!\langle c\rangle$. Thus any applications of f recursively use the encoding of the function body (over the dual $\overline{d}$). Recursion is then terminated when f is not called within a program trace of M.

3.3 Subeffecting

Subeffecting allows the effects of an expression to be overapproximated. In a related way, (traditional) subtyping in session types allows an approximation on branch and select [10, 11, 19]. However, encoding subeffecting is highly dependent on the rest of the effect encoding, and so parameterises the encoding (Definition 8).

As one universal possibility (for all notions of effect) we define here a *free* upper bound construction written $+$:

Definition 10 (Free upper bound)**.** For an effect system over $\mathcal{F}$, extend the carrier set to $\mathcal{F}_+ = \mathcal{F} \cup \{F + G \mid F, G \in \mathcal{F}_+\}$ where $+$ is a term constructor with no equations; it is a *free* constructor. Extend subeffecting such that $F \sqsubseteq F + G$ and $G \sqsubseteq F + G$ for all $F, G \in \mathcal{F}_+$ and extend $\bullet$ (sequential composition) to include a right-distributivity: $(F + G) \bullet H = (F \bullet H) + (G \bullet H)$. Note, this is not a least upper-bound, for one $+$ is not idempotent.

Effects $F + G$ can be encoded as two alternate effect behaviours simulated over one effect channel with branching/selection:

$$[\![F + G]\!] = \oplus[\mathsf{alt} : \oplus[\mathsf{L} : [\![F]\!], \mathsf{R} : [\![G]\!]]] \tag{8}$$

The interpretation of a typing derivation ending in an instance of the subeffecting rule (sub) is then, for $F \sqsubseteq F + G$:

$$\begin{aligned}&(\!|\, \Gamma \vdash M : \tau, F + G \,|\!)^{\mathsf{ei},\mathsf{eo}}_r \\ &\quad = \nu \mathsf{ea}.(\mathsf{ei}?(c).c \lhd \mathsf{alt}.c \lhd \mathsf{L}.\overline{\mathsf{ea}}!\langle c\rangle \mid (\!|\, \Gamma \vdash M : \tau, F \,|\!)^{\mathsf{ea},\mathsf{eo}}_r)\end{aligned}$$

On the effect channel c, received on ei, the alt label is selected followed by the L label before passing on the channel to the interpretation of M. Subtyping on the selection of L introduces the R branch giving the following typing for ei when $\overline{\mathsf{eo}} : [\![g]\!]$:

$$\mathsf{ei} : \oplus[\mathsf{alt} : \oplus[\mathsf{L} : [\![F \bullet g]\!], \mathsf{R} : [\![G \bullet g]\!]]]$$

Note, this makes use of the right-distributivity rule in order that the embedding is well-typed (see *typability*, Proposition 3, p. 7).

For the subtyping $G \sqsubseteq F + G$ the interpretation above differs by selecting R and introducing the L branch via subtyping.

The above definitions imply requirements on the handler H such that it has dual behaviour to match the encoding of $+$:

$$\emptyset; \mathit{eff} : \mu\alpha.S \vdash H(\mathit{eff}) \quad \Rightarrow \quad S <: \&[\mathsf{alt} : \&[\mathsf{L} : \alpha, \mathsf{R} : \alpha]]$$

(subtypes of a branching offer more choices than the supertype).

Remark 1. Setting $\oplus = +$ gives a free representation of alternation in effects (computation trees). For the general encoding, restricted **case** (Def. 9) can be composed with subeffecting in the premises to introduce $+$ effects.

Commutativity and associativity of $+$ is up to isomorphism, though the interpretation of $+$ (eq. 8) is commutative due to commutativity of labelled types in a selection type.

3.4 Homomorphic embedding of effect annotations

In the parameters to our embedding (Definition 8), the effect annotation interpretation $[\![-]\!] : \mathcal{F} \to S$ is required to satisfy the following homomorphism property, which ensures the continuation-passing style approach is well typed:

$$[\![I]\!] = \mathbf{end} \qquad [\![F \bullet G]\!] = [\![F]\!] \blacklozenge [\![G]\!]$$

where $\blacklozenge$ is *sequential session-type composition*, defined:

$$\begin{aligned}
T \blacklozenge \mathbf{end} &= T & \mathbf{end} \blacklozenge T &= T \\
![\tau].S \blacklozenge T &= ![\tau].(S \blacklozenge T) & ?[\tau].S \blacklozenge T &= ?[\tau].(S \blacklozenge T) \\
\oplus[\tilde{l} : \tilde{S}] \blacklozenge T &= \oplus[\tilde{l} : (\tilde{S} \blacklozenge T)] & \&[\tilde{l} : \tilde{S}] \blacklozenge T &= \&[\tilde{l} : (\tilde{S} \blacklozenge T)] \\
*![\tau].S \blacklozenge T &= *![\tau].(S \blacklozenge T) & & \\
(\mu\alpha.S) \blacklozenge (\mu\beta.T) &= \mu\gamma.(S[\gamma/\alpha] \sqcup T[\gamma/\beta]) \quad (\gamma \text{ fresh}) & &
\end{aligned}$$

where $\tilde{S} \blacklozenge T$ is the vector-scalar lifting of $\blacklozenge$. All other cases are undefined, *e.g.*, $*?[\tau] \blacklozenge T = \bot$. Arguments of the composition are taken up-to equivalence of session types (which is decidable).

The case for μ is defined for a least upper bound of S and T (as defined by session subtyping), introducing a fresh variable γ, *e.g.*:

$$(\mu\alpha.\oplus[l_1 : \alpha]) \blacklozenge (\mu\beta.\oplus[l_2 : ?[\tau].\beta]) = \mu\gamma.\oplus[l_1 : \gamma, l_2 : ?[\tau].\gamma]$$

3.5 Examples

In our examples, we define effect handlers as recursive processes via replicated input, of the form $\nu h.(*?h(c, \tilde{x}).P \mid \overline{h}!\langle c, \tilde{V}\rangle)$ for some effect channel c and value arguments $\tilde{V}$. The process P tends to contain an output on $\overline{h}$ to create the recurring handler behaviour.

Example 2 (Simple state)**.** Example 1 instantiated PCF for simple state effects with *get* and *put* operations and a list-based effect system. We restrict these to a single type τ (mono-typed stores) for simplicity. We instantiate our encoding to embed this into the session calculus. Effect annotations are interpreted as:

$$\begin{aligned}[\![(rd\ \tau) :: F]\!] &= \oplus[\mathsf{rd} : ?[\tau].[\![F]\!]] & [\![\,]\!] &= \mathbf{end} \\ [\![(wr\ \tau) :: F]\!] &= \oplus[\mathsf{wr} : ![\tau].[\![F]\!]]\end{aligned}$$

The interpretation of alternation effects $\oplus$ here is the free encoding $+$ via subeffecting (Section 3.3). The effect handler H is defined similarly to the variable agent in the introduction, where $H(\mathit{eff}) =$

$$\begin{aligned}&\mathit{eff} : \mu\alpha.\&[\mathsf{rd} : ![\tau].\alpha,\ \mathsf{wr} : ?[\tau].\alpha,\ \mathsf{alt} : \&[\mathsf{L} : \alpha, \mathsf{R} : \alpha],\ \mathsf{stop} : \mathbf{0}] \\ &\vdash \nu h.(*?h(c, x).c \rhd \{\mathsf{rd} : c!\langle x\rangle.\overline{h}!\langle c, x\rangle, \mathsf{wr} : c?(y).\overline{h}!\langle c, y\rangle, \\ &\qquad\qquad \mathsf{alt} : c \rhd \{\mathsf{L} : \overline{h}!\langle c, x\rangle, \mathsf{R} : \overline{h}!\langle c, x\rangle\} \\ &\qquad\qquad \mathsf{stop} : \mathbf{0}\} \mid h\langle \mathit{eff}, 0\rangle)\end{aligned}$$

with terminator $T(\mathit{eff}) = \mathit{eff} \lhd \mathsf{stop}$. State operations are encoded:

$$\begin{aligned}(\!|\, get \,|\!)^{\mathsf{ei},\mathsf{eo}}_r &= \mathsf{ei}?(c).c \lhd \mathsf{rd}\,.\, c?(x).r!\langle x\rangle.\overline{\mathsf{eo}}!\langle c\rangle \\ (\!|\, put \,|\!)^{\mathsf{ei},\mathsf{eo}}_r &= \nu d.(\mathsf{ei}?(c).r!\langle \overline{d}\rangle.\overline{\mathsf{eo}}!\langle c\rangle.*d?(p, \mathsf{ea}, \mathsf{eb}, q). \\ &\qquad p?(x).\mathsf{ea}?(c).c \lhd \mathsf{wr}\,.\, c!\langle x\rangle.q!\langle \mathsf{unit}\rangle.\overline{\mathsf{eb}}!\langle c\rangle)\end{aligned}$$

The encoding 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 encoding is similar to get, but with function-like encoding.

Example 3 (Simple state, with sets)**.** Classical effect systems tend to record just sets of effects, hence the order is not captured. That is, effect annotations are sets $\mathcal{F} = \mathcal{P}(\{\mathsf{rd}\,\tau, \mathsf{wr}\,\tau \mid \forall\tau\})$, with $(\mathcal{F}, \cup, \emptyset)$ for sequencing and $\oplus = \cup$, $F^* = F$. This system can be encoded in the session calculus. The effect embedding parameters (Definition 8) are the same as for the causal state encoding (above), apart from the effect interpretation $[\![-]\!]$ which is instead given in terms of a recursive type over a selection:

$$\begin{aligned}[\![F]\!] = \mu\alpha.S \ \ \textit{where}\ &\forall(\mathsf{rd}\,\tau) \in F \Rightarrow \oplus[\mathsf{rd} : ?[\![\tau]\!].\alpha] <: S \\ &\wedge\ \forall(\mathsf{wr}\,\tau) \in F \Rightarrow \oplus[\mathsf{wr} : ![\![\tau]\!].\alpha] <: S\end{aligned} \tag{9}$$

That is, a computation that may perform some effects F has some number of effect interactions S with the handler, where S is a selection type between all possible operations in F.

Thus, $[\![\{\mathsf{rd}\,\tau, \mathsf{wr}\,\tau\}]\!] = \mu\alpha.\oplus[\mathsf{rd} : ?[\![\tau]\!].\alpha, \mathsf{wr} : ![\![\tau]\!].\alpha]$. This interpretation of effects into session types is well-typed with respect to the encoding of *get* and *put* in Example 2 and its handler, and satisfies the homomorphism property. Section 9.1 embeds a traditional effect system for first-class references.

Remark 2 (Composition)**.** Our encoding is inspired by the approach of *algebraic effects handlers* for giving effect semantics and implementations [3, 29, 46]. In this approach, and in our encoding, multiple effects can be easily composed with distinct, independent handlers. Interactions between effects in our encoding can then be described via communication between handlers.

Example 4 (Resource counting / complexity)**.** A common effect system over natural numbers $\mathbb{N}$ is used to count resource use, such as "steps" in a computation or number of times a (costly) resource is used (see, *e.g.*, the work of Çiçek *et al.* [12] and Danielsson, via the *Thunk* annotated monad) [13]. The sequential part of this effect system is given by the ordered monoid $(\mathbb{N}, +, 0, \leq)$ with $\oplus = \mathsf{max}$. For recursion, the domain is extended to $\mathbb{N} \cup \{\omega\}$ such that $n^* = \omega$ which is greatest element w.r.t $\sqsubseteq$ and absorbing w.r.t to $+$. Thus, *FPCF* can be instantiated for this effect system with the additional constant $\emptyset \vdash \mathsf{tick} : \mathbf{unit}, 1$.

Resource counting effects are embedded via the following handler, terminator, and operation encoding:

$$H(c) = \nu h.(*h?(c).c?(d).d \triangleright \{\mathsf{tick} : d?(x).\overline{h}!\langle c\rangle, \mathsf{stop} : \mathbf{0}\} \mid \overline{h}!\langle c\rangle)$$
$$T(c) = \nu d.(c!\langle d\rangle.\overline{d} \triangleleft \mathsf{stop})$$
$$[\![\mathsf{tick}]\!]_r^{\mathsf{ei},\mathsf{eo}} = \nu d.(\mathsf{ei}?(c).c!\langle\overline{d}\rangle.d \triangleleft \mathsf{tick}.d!\langle\mathsf{unit}\rangle.\overline{\mathsf{eo}}!\langle c\rangle)$$

That is, a "tick" is encoded by sending a channel d over which is selected the tick behaviour of the handler and a unit value is then sent. The encoding of effect annotations $[\![-]\!] : \mathbb{N} \to \mathcal{S}$ is then essentially a base-1 encoding of naturals:

$$[\![0]\!] = \mathbf{end} \quad [\![n+1]\!] = *![\&[\mathsf{tick} : ?[\mathbf{unit}]]].[\![n]\!]$$
$$[\![\omega]\!] = \mu\alpha.*![\&[\mathsf{tick} : ?[\mathbf{unit}]]].\alpha$$

The handler has type $c : *?[\&[\mathsf{tick} : ?[\mathbf{unit}]]]$. Type soundness of this encoding requires a small extension to definition of session type duality for *repeated* output.

Definition 11 (Repeated output)**.** We write $*![\tau]^n$ for a finite sequence of outputs: $*![\tau]^n = *![\tau].*![\tau]^{n-1}$ with $*![\tau]^0 = \mathbf{end}$. We extend the duality function to a symmetric relation *dual* with:

$$\forall n.\ dual(*![\tau]^n, *?[\tau]) \tag{10}$$

The limit of this is a fixed point, *i.e.* $\lim_{n\to\infty} eq.\ (10) = \mu\alpha.*![\tau].\alpha$ giving $dual(\mu\alpha.*![\tau].\alpha, *?[\tau])$ which provides the duality of $[\![\omega]\!]$ effects with the handler. Subtyping is then extended with:

$$*![\tau]^m <:\ *![\tau]^n \quad (\textit{where } m \leq n) \tag{11}$$

Subeffecting by $\leq$ follows from equation (11). That is, the number of outputs (dual to replicated input) can be overapproximated.

4. Correctness: from effects to sessions

This section discusses the correctness of our encoding: the main results are (1) the typability of encoding; (2) sound and complete operational correspondence; and (3) soundness up to observational equivalence. Full proofs are given in the technical report [41].

Proposition 3 (Typability)**.** Given $\Gamma \vdash M : \tau, F$, then we have:

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

for the intermediate encoding and for the top-level encoding:

$$[\![\Gamma]\!]^\circ; [\![\Gamma]\!], r :![\![\tau]\!] \vdash [\![\Gamma \vdash M : \tau, F]\!]_r$$

Observational equivalence for the session calculus uses a *barbed reduction-based congruence* from [23, 49]. As an intermediate step, we define *barbs*: predicates which classify processes P by their observable reductions with respect to a channel c.

Definition 12 (Barbs)**.** 1. (*untyped barb*) $P \downarrow_c$ if $\exists V, P_2, P_3$ s.t. $P \equiv \nu\tilde{m}.(c!\langle V\rangle.P_2 \mid P_3)$ where $c \notin \tilde{m}$
2. (*barb*) $\Gamma; \Delta \vdash P \downarrow_c$ if $P \downarrow_c$ and $\overline{c} \notin dom(\Delta)$
3. (*weak barb*) $\Gamma; \Delta \vdash P \Downarrow_c$ if $P \to^* Q \wedge \Delta \to^* \Delta' \wedge \Gamma; \Delta' \vdash Q \downarrow_c$

A barb is an observable send prefix with subject c; a weak barb is a barb after some reduction steps. Typed barbs of c occur on typed processes where $\overline{c}$ is not free (else the send would be unobservable).

Definition 13 (Reduction-closed barbed congruence)**.** A relation $\mathcal{R}$ is a *reduction-closed barbed congruence* if given processes P, Q such that $\Gamma; \Delta_1 \vdash P$ and $\Gamma; \Delta_2 \vdash Q$ then $(\Gamma, \Delta_1, P, \Delta_2, Q) \in \mathcal{R}$, written $\Gamma, \Delta_1 \vdash P\ \mathcal{R}\ \Delta_2 \vdash Q$, if:

- if $\Gamma; \Delta_1 \vdash P \Downarrow_c$ then $\Gamma; \Delta_2 \vdash Q \Downarrow_c$
- $\forall\ P \to P'$ then $\exists Q', \Delta_1', \Delta_2'$ such that $Q \to^* Q'$ and $\Gamma; \Delta_1' \vdash P'\ \mathcal{R}\ \Delta_2' \vdash Q'$
- $\forall \mathbb{C}.\ \exists \Delta_1'', \Delta_2''$ such that $\Gamma; \Delta_1'' \vdash \mathbb{C}[P]\ \mathcal{R}\ \Delta_2'' \vdash \mathbb{C}[Q]$
- $\mathcal{R}$ is symmetric

The first condition ensures both processes preserve the barb; the second means that a relation is a closure only based on reductions.

We write $\cong$ for the largest reduction-closed barbed congruence over our session calculus.

The correctness of our general embedding in Section 3 relies on the following intermediate lemmas. Lemma 1 is important for deciding a shape of a value encoding. Lemma 2 states the encoding mimics the substitution of values. Lemma 3 shows a forwarding (link) agent corresponds to the identity process (*cf.* [7, 51]), which is ensured by the linearity of session types.

Lemma 1 (Value-encoding lemma)**.** For all values V, then

$$\exists P, d, y\ .\ (\!|\, V \,|\!)_r^{\mathsf{ei},\mathsf{eo}} \equiv \nu d.(\mathsf{ei}?(c).r!\langle y\rangle.\overline{\mathsf{eo}}!\langle c\rangle.P)$$

The purity of values is encoded by passing the effect channel c unused, interleaved with sending some result y.

Lemma 2 (Substitution distributes with embedding)**.**

$$\nu s, \mathsf{ea}.\ (\overline{s}?(x).(\!|\, M \,|\!)_r^{\mathsf{ea},\mathsf{eo}} \mid (\!|\, V \,|\!)_s^{\mathsf{ei},\mathsf{ea}}) \ \cong\ (\!|\, M[V/x] \,|\!)_r^{\mathsf{ei},\mathsf{eo}}$$

Lemma 3 (Forwarding)**.** For process P and channel $\mathsf{ea} \notin \mathsf{fv}(P)$

$$\nu\mathsf{eb}.(\mathsf{ea}?(c).\overline{\mathsf{eb}}!\langle c\rangle.P \mid (\!|\, M \,|\!)_r^{\mathsf{eb},\mathsf{eo}}) \ \cong\ P \mid (\!|\, M \,|\!)_r^{\mathsf{ea},\mathsf{eo}}$$
$$\wedge\ \nu\mathsf{eb}.((\!|\, M \,|\!)_r^{\mathsf{ei},\mathsf{ea}} \mid \mathsf{ea}?(c).\overline{\mathsf{eb}}!\langle c\rangle.P) \ \cong\ P \mid (\!|\, M \,|\!)_r^{\mathsf{ei},\mathsf{eb}}$$

The first theorem is sound and complete operational correspondence. Soundness states the encoding mimics *FPCF*, while completeness ensures that if an encoding of an *FPCF*-term reduces one step, there is a corresponding computation that happens in *FPCF*.

Theorem 1 (Operational correspondence)**.** $\forall M, F, \tau, \Gamma.$

- (sound) $\forall N.\ \Gamma \vdash M : \tau, F\ \wedge\ (M \to N)\ \wedge\ F \sqsubseteq G \Rightarrow\ \exists P.\ ([\![\Gamma \vdash M : \tau, F]\!]_r \to P)\ \wedge\ P \cong [\![\Gamma \vdash N : \tau, G]\!]_r$
- (complete) $\forall P.\ (\Gamma \vdash M : \tau, F)\ \wedge\ ([\![\Gamma \vdash M : \tau, F]\!]_r \to P) \Rightarrow\ \exists N, G.\ (M \to^* N) \wedge\ P \cong [\![\Gamma \vdash N : \tau, G]\!]_r\ \wedge F \sqsubseteq G$

Lemma 4. Whilst **let** is syntactic sugar, it had its own encoding (Section 3). This encoding is sound w.r.t. the definition of **let**:

$$[\![\Gamma \vdash \mathbf{let}\, x = M\, \mathbf{in}\, N : \tau, F \bullet G]\!]_r \cong [\![\Gamma \vdash (\lambda x.N)\, M : \tau, F \bullet G]\!]_r$$

The main theorem of this section is then equational soundness of the encoding with respect to $\beta\eta$-equality of *FPCF*:

Theorem 2 (Soundness)**.** (with respect to $\beta\eta$-equality)

$$\Gamma \vdash M \equiv N : \tau, F \Rightarrow [\![\Gamma \vdash M : \tau, F]\!]_r \cong [\![\Gamma \vdash N : \tau, F]\!]_r$$

Soundness is the standard statement to show the correctness of the encoding, *e.g.*[44]. We have also proved completeness with respect to contextual equivalence (not $\beta\eta$-equality) (*cf.* [56, cor. 5.2]):

$$[\![\Gamma \vdash M : \tau, F]\!]_r \cong [\![\Gamma \vdash N : \tau, F]\!]_r \Rightarrow \Gamma \vdash M \cong N : \tau, F \tag{12}$$

Contextual completeness is a consequence of Theorem 1 together with computational adequacy, *cf.* [56, Corollary 5.2]. Proving soundness with respect to contextual equality $\cong$ for *FPCF* terms would provide full abstraction. This is further work.

Note, these results are for the *general* encoding. Correctness of any effect-specific embeddings of effect operations (*e.g.*, *get*/*put*) requires the above conditions to be checked on these encodings.

5. Concurrency and effects

In a concurrent setting, side effects are a common source of unintended program behaviour, allowing implicit interactions between threads. Consider an extension to *FPCF* for parallel composition, written $M \parallel N$. Unrestricted effects in parallel branches may cause race conditions, *e.g.* $put\,(get + 1) \parallel put\,(get + 2)$ has three possible values due to non-deterministic interleaving of get and put.

We extend our encoding to possibly *interfering* concurrent effects and show an example using state effects.

To add parallel composition into *FPCF*, the effect algebra is extended with a semigroup $(F, \diamond)$ representing parallel computation where I (purity) is the unit of $\diamond$. Typing and equality is extended:

$$\frac{\Gamma \vdash M : \mathbf{unit}, F \quad \Gamma \vdash N : \mathbf{unit}, G}{\Gamma \vdash M \parallel N : \mathbf{unit}, F \diamond G} \tag{13}$$

$$M \parallel \mathsf{unit} \equiv M \quad M \parallel N \equiv N \parallel M \quad M \parallel (N \parallel P) \equiv (M \parallel N) \parallel P$$

The semantics for the parallel composition $\parallel$ is standard, allowing non-deterministic reduction of the left or right process. A completed parallel composition reduces by $\mathsf{unit} \parallel \mathsf{unit} \to \mathsf{unit}$.

Concurrent state effects Recall the state effects of Example 2. To incorporate concurrent interactions, we first redefine the state handler (eliding alternation) with an extra layer of indirection:

$$\nu h.(*h?(c,x).c?(d).d \rhd \{\mathsf{rd} : d!\langle x\rangle.\overline{h}!\langle c,x\rangle, \mathsf{wr} : d?(y).\overline{h}!\langle c,y\rangle, \mathsf{stop} : \mathbf{0}\} \mid \overline{h}!\langle \mathit{eff}, 0\rangle)$$

This handler has the following recursive session type for eff:

$$\mathit{eff} : \mu\alpha.?[S].\alpha \quad \textit{where} \quad S = \&[\mathsf{rd} :![\tau], \mathsf{wr} :?[\tau], \mathsf{stop} : \mathbf{end}]$$

i.e., *eff* repeatedly receives a channel over which the standard variable agent behaviour S is offered. State operations are encoded similarly to before but with the extra indirection, *e.g.*, for *get*:

$$(\!|\, get \,|\!)^{\mathsf{ei},\mathsf{eo}}_r = \mathsf{ei}?(c).\nu e.(c!\langle\overline{e}\rangle.e \lhd \mathsf{rd}\,.\, e?(x).r!\langle x\rangle.\overline{\mathsf{eo}}!\langle c\rangle)$$

The associated effect annotation interpretation is then as follows, where $c!\langle\overline{e}\rangle$ in the above definition (sending a fresh channel to the handler) is typed with a replication type:

$$\begin{array}{ll} [\![\,]\!] = \mathbf{end} & [\![(rd\ \tau) :: F]\!] = *![\&[\mathsf{rd} :![\tau]]].[\![F]\!] \\ & [\![(wr\ \tau) :: F]\!] = *![\&[\mathsf{wr} :?[\tau]]].[\![F]\!] \end{array} \tag{14}$$

The parallel composition operator is then defined:

$$\begin{array}{l} (\!|\, M \parallel N \,|\!)^{\mathsf{ei},\mathsf{eo}}_r = \nu q_1, q_2, \mathsf{eo}_1, \mathsf{eo}_2.(\mathsf{ei}?(c).(\\ \quad \nu \mathsf{ea}.(\overline{\mathsf{ea}}!\langle c\rangle \mid (\!|\, M \,|\!)^{\mathsf{ea},\mathsf{eo}_1}_{q_1}) \mid \nu \mathsf{ea}.(\overline{\mathsf{ea}}!\langle c\rangle \mid (\!|\, N \,|\!)^{\mathsf{ea},\mathsf{eo}_2}_{q_2})) \\ \quad \mid \overline{q_1}?(x).\overline{q_2}?(y).r!\langle \mathsf{unit}\rangle \\ \quad \mid \nu c.(\mathsf{eo}_1?(c_1).\mathsf{eo}_2?(c_2).(\nu c_r. \\ \quad (*c_r?(c).c?(d).c_1!\langle d\rangle.c?(d).c_2!\langle d\rangle.\overline{c_r}!\langle c\rangle \mid \overline{c_r}!\langle c\rangle)) \mid \overline{\mathsf{eo}}!\langle\overline{c}\rangle)) \end{array}$$

That is, the effect channel c is received on ei, and sent concurrently in two parallel branches to $(\!|\, M \,|\!)$ and $(\!|\, N \,|\!)$ (in the second line). The third line receives the unit results from the parallel branches and sends the final return unit. The fourth line plumbs together the outgoing effect channels c_1 and c_2 from the intermediate encodings into a single outgoing effect channel.

By interpreting effect annotations using output (14), parallel use of the same channel in each branch is typeable by balancing (Def. 7): $*![T].S \bowtie *![T].S$. This requires a single, unifying session type for c in each branch. Let M, N have for c the session types $[\![F \bullet h_1]\!]$ and $[\![G \bullet h_2]\!]$, of the form $*![S_1]...*![S_n]$ and $*![T_1]...*![T_m]$ respectively. An upper-bound type can then be given $c : *![S]^{(n \max m)}$ where S is the sequential state handler behaviour $S = \&[\mathsf{rd} :![\tau], \mathsf{wr} :?[\tau], \mathsf{stop} : \mathbf{end}]$ which is the common supertype of each possible effect interaction $S_1...S_n$ and $T_1...T_m$:

$$*![\&[\mathsf{rd} :![\tau]]] <: *![S] \qquad *![\&[\mathsf{wr} :?[\tau]]] <: *![S]$$

Thus, the interpretation of parallel effect annotations is *lossy*, as:

$$[\![F \diamond G]\!] = *![S]^{(n \max m)}$$

This can be understood as describing the possible arbitrary interleaving, thus potential interference, provided by parallel effects. The encoding is well-typed when the duality function is extended to a relation as in Definition 11 (with $\forall n.\mathit{dual}(*![S]^n, *?[S])$).

Type-preservation, soundness, and the following operational correspondence theorem hold for this extension.

Theorem 3 (Correspondence for parallel effects)**.** $\forall M, F, \tau, \Gamma.$

- (sound) $\forall N.\ \Gamma \vdash M : \tau, F \ \wedge\ (M \to N) \ \wedge\ F \sqsubseteq G \Rightarrow \exists P.\ ([\![\Gamma \vdash M : \tau, F]\!]_r \to P) \ \wedge\ P \cong [\![\Gamma \vdash N : \tau, G]\!]_r$
- (complete) $\forall P.\ (\Gamma \vdash M : \tau, F) \ \wedge\ ([\![\Gamma \vdash M : \tau, F]\!]_r \to P) \Rightarrow \exists N, G.\ (M \to^* N) \wedge\ P \cong [\![\Gamma \vdash N : \tau, G]\!]_r \ \wedge F \sqsubseteq G$

Theorem 4 (Soundness *w.r.t.* extended $\beta\eta$-equality (13))**.** Lemma 2 (substitution) extends to the parallel encoding. Then: $\Gamma \vdash M \equiv N : \tau, F \Rightarrow [\![\Gamma \vdash M : \tau, F]\!]_r \cong [\![\Gamma \vdash N : \tau, F]\!]_r$

For this extended encoding, equational completeness (*w.r.t* observational equivalence (12)) does not hold. This is because any process $P \cong [\![\Gamma \vdash N : \tau, F]\!]^{\mathit{eff}}_r$ could be placed under a context which interacts non-deterministically with the handler.

6. Summarising the requirements

To conclude this part of the paper, we summarise the features of the session calculus that were required to encode *FPCF*. We consider increasing subsets of *FPCF* and the session calculus (π_S here). The *core* subset of π_S (send, receive, delegation, branch, select, parallelism, and restriction) is contained by the subset providing replicated input and output, denoted $*$ below. We further split off **if** and subtyping $<:$, as well as polymorphic sessions (not used so far, but later in Section 9.1) and the condition that multiple sequentially-composed outputs are dual to a replicated input (eq. 10 of Def. 11).

FPCF $\setminus\ \pi_S$	core	$*$	**if**	$<:$	*eq.* 10	$\forall$
$\bullet$	✓	✓				
$\to$	✓	✓				
$-^*$	✓	✓				
$\oplus = (\equiv)$	✓	✓	✓			
$\oplus = (+)$	✓	✓	✓	✓		
$\parallel$	✓	✓	✓	✓	✓	
$\mathcal{F}^\tau$	✓	✓	.	.	.	✓

The $\oplus = (\equiv)$ line represents the restricted version of **case** which has $F \oplus F = F$ and $F \oplus G = \bot$ when $F \neq G$. The $\oplus = (+)$ line represents $\oplus$ constructed via subeffecting with the free alternation operator $+$ (Section 3.3). The $\mathcal{F}^\tau$ line represents effects with varying types, such as effects for first-class references, which are discussed in Section 9.1.

Note that the requirements here are only on the general encoding. Specific instantiations may use all/any of these features, *e.g.*, Example 4 used the duality relation extension of equation 10.

Linear control-flow effects The examples given in Section 3.5 can be characterised by linearity in their control flow (equivalent to algebraic effect handlers [3, 46] that are linear in their continuation). Effects such as exceptions (which interrupt control-flow) and non-determinism (with branching control-flow effects) may plausibly be captured via encodings that explicitly include their continuation, *e.g.* $[\![\mathbf{let}\ x = \mathit{choose}\ V_1\ V_2\ \mathbf{in}\ M]\!]$. This is further work.

7. Back again: sessions as effects

This section considers the reverse direction of encoding, showing that *FPCF* with parallel composition (of Section 5) can be instantiated with a notion of *session effect*, into which the session calculus can be embedded. The key insight is that session types and *causal* (non-commutative) effects have the same structure. Both give an ordered analysis of the operational behaviour of a program. The prefixing style of session types is replaced with the monoidal style of effect systems, akin to the difference lists (prefixing) versus normal concatenation of lists. Furthermore, our rich effect system provides a way to represent branching/selection session types via alternation $\oplus$, and replication via recursion and $-^*$. The effect system for

sessions is *partial*–some operations may not be defined for all arguments, modelling the program-logic behaviour of session types.

Types We give an embedding of sessions types into an effect algebra (Definition 1), making clear the homomorphic nature of session types and effect systems. We first define a variant of session types S called *effect sessions* where an alternation operator $+$ replaces select $\oplus$ and branch $\&$, with $\tau ::= \mathbf{nat} \mid \mathbf{unit} \mid [\mathsf{S}]$ and

$$\mathsf{S}, \mathsf{T} ::= !{[\tau]}.\mathsf{S} \mid ?{[\tau]}.\mathsf{S} \mid *!{[\tau]}.\mathsf{S} \mid \mathbf{end} \mid \mathsf{S} + \mathsf{S} \mid \mu\alpha.\mathsf{S} \mid \alpha \mid \hat{\odot}\mathsf{S}$$

where $\hat{\odot}\mathsf{S}$ is an intermediate representation for the session type of channels being composed via $\odot$ (balanced composition). We assume equirecursive equations on S, *e.g.*, $\mu\alpha.\mathsf{S} = \mathsf{S}[\mu\alpha.\mathsf{S}/\alpha]$.

Definition 14 (Session effects)**.** Let $\mathcal{F} = (\mathcal{C} \to \mathsf{S}) \cup \{\bot\}$ be maps from channel names $\mathcal{C}$ to effect session types with algebra:

- $(\mathcal{F}, \bullet, \emptyset)$ where $I = \emptyset$ is the empty mapping, $\bullet$ is pointwise sequential composition of session types (via $\blacklozenge$, Section 3.4) where $\forall c : \mathsf{S} \in F \wedge c : \mathsf{T} \in G$ then $c : \mathsf{S} \blacklozenge \mathsf{T} \in (F \bullet G)$ but with additional equations for balancing composition triggered by $\hat{\odot}$ types *e.g.* $\Delta \blacklozenge (c : \hat{\odot}\mathsf{S}) = \bot$ *if* $\overline{c} : \overline{\mathsf{S}} \notin \Delta$ amongst other cases (the technical report [41] gives the full definition).
- $\oplus$ is set union $\cup$ if $\Delta_1 \oplus \Delta_2$ satisfies the following condition:
$$\forall c : \mathsf{S} \in \Delta_i \Rightarrow c : \mathsf{S} \in \Delta_j \vee c \notin \mathit{dom}(\Delta_j)$$
i.e., a channel is either in Δ_1 or Δ_2, or in both but with the same session type, otherwise $\Delta_1 \oplus \Delta_2 = \bot$.
- $\sqsubseteq$ is a preorder where $\forall \Delta, c.$ where $c \notin \mathit{dom}(\Delta)$:
$$\begin{array}{ll} \Delta \sqsubseteq (\Delta, c : \mathbf{end}) & (\Delta, c : \mathsf{S}) \sqsubseteq (\Delta, c : \mathsf{S} + \mathsf{T}) \\ \Delta \sqsubseteq (\Delta, c : *![\mathsf{S}]) & (\Delta, c : \mathsf{S}) \sqsubseteq (\Delta, c : \mathsf{T} + \mathsf{S}) \end{array} \quad (15)$$
- $\diamond = \odot$, for parallel effects, takes the union of two mappings if they are balanced in their channels, otherwise $\bot$;
- F^* is defined $\forall c : \mathsf{S} \in F$ then $c : \mu\alpha.(\mathbf{end} + \mathsf{S} \blacklozenge \alpha) \in F^*$.

We extend *FPCF* values with *channel values* $\mathcal{C}$ ranged over by $\mathbf{c}, \mathbf{d}, \mathbf{e}$ and their dual endpoints $\overline{\mathbf{c}}, \overline{\mathbf{d}}, \overline{\mathbf{e}}$. Channel values belong to singleton types corresponding to the channel name: $\mathbf{c} : \mathit{Ch}\ c$. This provides (simple) value dependency in the types.

Remark 3. The $\oplus$ operation above is not the least-upper bound with respect to $\sqsubseteq$. Defining subeffecting as subset inclusion instead of (15) above would give least-upper bounds with the union-like behaviour of $\oplus$, however this is unsound: arbitrary session types could be introduced without a corresponding implementation. This shows the need for separate $\oplus$ and $\sqsubseteq$.

Sending, receiving and restriction operations We add to the constants C of *FPCF* operations that send and receive values, send and receive channels, and restrict channels:

$$\begin{array}{rl} \mathit{send}_{c,\tau} : & \mathit{Ch}\ c \to \tau \xrightarrow{\{c :![\tau]\}} \mathbf{unit} \\ \mathit{rsend}_{c,d,s} : & \mathit{Ch}\ c \to \mathit{Ch}\ d \xrightarrow{\{c : *![s], d : \hat{\odot}s\}} \mathbf{unit} \\ \mathit{chSend}_{c,d,s} : & \mathit{Ch}\ c \to \mathit{Ch}\ d \xrightarrow{\{c :![s], d : \hat{\odot}s\}} \mathbf{unit} \\ \mathit{recv}_{c,\tau} : & \mathit{Ch}\ c \xrightarrow{\{c :?[\tau]\}} \tau \\ \mathit{chRecv}_{c,d,s,F,\tau} : & \mathit{Ch}\ c \xrightarrow{\{c :?[s]\}} (\mathit{Ch}\ d \xrightarrow{F \bullet \{d : s\}} \tau) \xrightarrow{F} \tau \\ \mathit{new}_{c,s,F,\tau} : & (\mathit{Ch}\ c \to \mathit{Ch}\ \overline{c} \xrightarrow{F \bullet \{c : s, \overline{c} : \overline{s}\}} \tau) \xrightarrow{F} \tau \end{array}$$

Each operation is a family of operations, indexed by the types shown as subscripts. This allows our type-directed encoding to choose the correctly typed operation. Each operation has latent effects which give the session environment induced by the operation.

The *send* and *recv* operations correspond to session send/receive prefixes with effect types describing the single action on their channel. For *chSend*, the second channel parameter d is sent over c where d must be balanced with the rest of the environment when composing due to the $\hat{\odot}$ operator; *rsend* is identical to *chSend*, but with the $*!$ session type. The *chRecv* operation is higher-order, taking a channel c and over which a channel of session type s is received and passed to the parameter function which maps a channel d to a value τ with the effect $F \bullet \{d : s\}$. From this, a computation is returned with F channels where $d, c \notin F$. This typing is important for the typability of replicated input. The *new* operation is similarly higher-order, where the resulting effect is the effect of the parameter function, but with the session types $c : s, \overline{c} : \overline{s}$ deleted from the environment since they are in scope only for the parameter function.

Operational semantics We instantiate the operational semantics of *FPCF* for the session effect operations. Configurations are pairs $\langle M, s\rangle$ of a term M with store s mapping channel endpoint names $\mathcal{C}$ to (unbounded) queues of values. We write $\mathsf{enq}\, s\, c\, V$ to update store s with the value V added to end of the queue belonging to c. The operation $\mathsf{deq}\, s\, c$ returns a pair of the first element in the queue for c and an updated store if c is non-empty, otherwise deq is undefined. Both $\mathsf{enq}\, s\, c\, V$ and $\mathsf{deq}\, s\, c$ require $c \in \mathit{dom}(s)$ otherwise they are undefined; ϵ denotes the empty queue. Sending and receiving have the following reductions from their redex form (with arguments reduced to values by the usual application rules):

$$\begin{array}{rll} \langle \mathit{send}\ \mathbf{c}\ V, s\rangle & \to \langle \mathsf{unit}, \mathsf{enq}\ s\ \overline{c}\ V\rangle & \\ \langle \mathit{rsend}\ \mathbf{c}\ V, s\rangle & \to \langle \mathsf{unit}, \mathsf{enq}\ s\ \overline{c}\ V\rangle & \\ \langle \mathit{chSend}\ \mathbf{c}\ \mathbf{d}, s\rangle & \to \langle \mathsf{unit}, \mathsf{enq}\ s\ \overline{c}\ \mathbf{d}\rangle & \\ \langle \mathit{recv}\ \mathbf{c}, s\rangle & \to \langle V, s'\rangle & \textit{where } \mathsf{deq}\, s\, c = (s', V) \\ \langle \mathit{chRecv}\, \mathbf{c}, s\rangle & \to \langle (\lambda k.k\, \mathbf{e}), s'\rangle & \textit{where } \mathsf{deq}\, s\, c = (s', \mathbf{e}) \end{array}$$

For *send*, *rsend*, and *chSend*, the value or channel is added to the end of the queue belonging to the opposite end-point $\overline{c}$. For *recv*, a value is dequeued from the endpoint c queue (if it exists in s); *chRecv* is a little different, receiving first a channel $\mathbf{e}$ over c and returning a function which takes a continuation k and applies it to the received channel. Thus receiving the channel is separated from substitution of that channel, which is reflected in the type of *chRecv*. This semantics allows asynchronous communication. Section 7.2 defines a *stable reduction* relation that characterises the reductions of our encoding, which are only the synchronous subset.

The reduction of *new* exposes a key difference between the two calculi. In processes, $\nu c.P$ introduces channel names c and $\overline{c}$ in the scope of P. In *FPCF*, *new* encodes restriction but *new* $(\lambda x.\lambda y.M)$ binds arbitrarily named variables x, y in the scope of M, which are substituted for concrete channel names. The (new1) rule below deals with this by first α-renaming the variables of a *new* redex to fresh variable names corresponding to fresh channels:

$$\text{(new1)} \quad \begin{array}{r} \textit{where } x, y, c, \overline{c} \notin \mathrm{dom}(s),\ c, \overline{c} \textit{ fresh in } M \\ \langle \mathit{new}(\lambda x\lambda y.M), s\rangle \to \langle \mathit{new}(\lambda c\lambda\overline{c}.M[c/x, \overline{c}/y]), s[c \mapsto \epsilon, \overline{c} \mapsto \epsilon]\rangle \end{array}$$

That is, if x, y are not in the store, then x, y are α-renamed to names $c, \overline{c}$ which are fresh in M (not free or bound) and are not already in the store. The store is then extended with empty channels for c and $\overline{c}$. The following two rules then reduce *new* further:

$$\text{(new2)} \ \frac{\langle M[\mathbf{c}/c, \overline{\mathbf{c}}/\overline{c}], s\rangle \to \langle N[\mathbf{c}/c, \overline{\mathbf{c}}/\overline{c}], s'\rangle \quad c, \overline{c} \in \mathit{dom}(s)}{\langle \mathit{new}\ (\lambda c\lambda\overline{c}.M), s\rangle \to \langle \mathit{new}\ (\lambda c\lambda\overline{c}.N), s'\rangle}$$

$$\text{(new3)}\ \langle \mathit{new}\ (\lambda c\lambda\overline{c}.V), s[c \mapsto \epsilon, \overline{c} \mapsto \epsilon]\rangle \to \langle V, s[c \mapsto \bot, \overline{c} \mapsto \bot]\rangle$$

In (new2), if M reduces with its variables c and $\overline{c}$ replaced by the corresponding channel values, then the restricted term reduces. Once a value is reached with empty restricted channels (new3), then *new* is eliminated and its restricted channels are deleted.

From this semantics, $\equiv$ can be extended with an η-expansion for *new* over pure values: $\mathit{new}(\lambda x\lambda y.V) \equiv V$ and distributivity $P \parallel \mathit{new}(\lambda x\lambda y.Q) \equiv \mathit{new}(\lambda x\lambda y.P \parallel Q)$ if $x, y \notin \mathsf{fv}(P)$.

7.1 Embedding the session calculus into *FPCF*

Encoding of types The encoding from the session calculus to *FPCF* is type-directed, giving the following typability lemma on the mapping from session type derivations to *FPCF* derivations:

Lemma 5 (Typability)**.** Let $\Gamma;\Delta \vdash P$ then

$$\exists M.\ [\![\Gamma;\Delta \vdash P]\!] = [\![\Gamma]\!] \vdash M : \mathbf{unit}, [\![\Delta]\!]$$

Value contexts are encoded $[\![\Gamma]\!]$ with **nat** and **unit** mapped to their corresponding *FPCF* value types. Session environments are mapped to effects $\mathcal{F}$ from Def. 14 by $[\![\Delta]\!] = [\![c_1 : S_1, ...c_n : S_n]\!] = c : [\![S_1]\!], ..., c_n : [\![S_n]\!]$ and the following interpretation:

Definition 15 (Session types to effect sessions annotations)**.**

$$[\![\mathbf{end}]\!] = \mathbf{end} \qquad [\![![\tau].S]\!] = ![\![\tau]\!].[\![S]\!] \qquad [\![?[\tau].S]\!] = ?[\![\tau]\!].[\![S]\!]$$
$$[\![*?[\tau]]\!] = \mu\alpha.?[\![\tau]\!].\alpha \quad [\![*![\tau].S]\!] = *![\![\tau]\!].[\![S]\!]$$
$$[\![\mu\alpha.S]\!] = \mu\alpha.[\![S]\!] \qquad [\![\oplus[l_1 : S, l_2 : T]]\!] = ![\mathbf{nat}].([\![S]\!] + [\![T]\!])$$
$$[\![\&[l_1 : S, l_2 : T]]\!] = ?[\mathbf{nat}].([\![S]\!] + [\![T]\!])$$

Natural numbers are used in the encoding to administer control-flow for branching and selection. A selection type $\oplus$ corresponds to sending a **nat** to select a branch prior to alternation. For $\&$, this corresponds to receiving a **nat** then alternating.

Encoding of processes We define the encoding of typed processes $[\![\Gamma;\Delta \vdash P]\!]$ by induction over type derivations.

In the majority of rules we elide the types (when the syntax has a single corresponding typing rule). Types are included only when the encoding has a (non-syntactically implied) type-dependence. For brevity we write $M\,;\,N$ for $\mathbf{let}\ x = M\ \mathbf{in}\ N$ when the bound variable x is free in N (*i.e.*, unused– a wildcard).

We first give the encoding of linear send and receive into *FPCF*:

$$[\![c!\langle V\rangle.P]\!] = \mathit{send}\ c\ [\![V]\!]\,;\,[\![P]\!]$$
$$[\![c?(x).P]\!] = \mathbf{let}\ x = \mathit{recv}\ c\ \mathbf{in}\ [\![P]\!]$$
$$[\![c?(d).P]\!] = \mathbf{let}\ k = \mathit{chRecv}\ c\ \mathbf{in}\ k\ (\lambda d.[\![P]\!])$$
$$[\![\Delta \vdash c!\langle d\rangle.P]\!] = \mathit{chSend}\ c\ d\,;\,[\![P]\!] \qquad (\mathit{if}\ c :![S].T \in \Delta)$$

The encoding straightforwardly uses the *send* and *recv* operations in *FPCF*. The **let**-binding encodes prefixing by sequential composition. In the case of sending a channel, *chSend* is used when c is marked as a linear send in Δ, showing the type-directed nature. Alternatively, the syntax $c!\langle d\rangle.P$ may be an output action, which is handled below in the encoding of output and replicated input:

$$[\![\Delta \vdash c!\langle d\rangle.P]\!] = \mathit{rsend}\ c\ d\,;\,[\![P]\!] \qquad (\mathit{if}\ c : *![S].T \in \Delta)$$
$$[\![*c?(d).P]\!] = \mathbf{rec}\ (\lambda f.\lambda x.\mathbf{let}\ k = \mathit{chRecv}\ c \ \mathbf{in}\ (k\ (\lambda d.[\![P]\!]))\parallel f\ \mathsf{unit})\ \mathsf{unit}$$

For output, *rsend* sends d before continuing with $[\![P]\!]$. Replicated input is defined via a recursive function. It repeatedly receives a channel on c which is bound as d in the scope of $[\![P]\!]$ in parallel (via the continuation k) with the recursive call. Typability for replicated input holds since the recursive call is balanced with respect to $(k\ (\lambda d.[\![P]\!]))$ (which has no session effect involving c).

Restricted, parallel, and empty processes are encoded directly:

$$[\![P \mid Q]\!] = [\![P]\!] \parallel [\![Q]\!] \quad [\![\nu c.P]\!] = \mathit{new}(\lambda c.\lambda\overline{c}.\ [\![P]\!]) \quad [\![\mathbf{0}]\!] = \mathsf{unit}$$

Parallel composition is encoded by the parallel extension of *FPCF* (Section 5) and restriction by *new*, binding names c and $\overline{c}$.

The encoding of branching, selection, and conditionals uses **case**. We consider just two concrete labels l_1, l_2, though this can be easily generalised to any finite set of labels (isomorphic to **nat**).

$$[\![c \lhd l_1.P]\!] = \mathit{send}\ c\ 0\,;\,[\![P]\!] \qquad [\![c \lhd l_2.P]\!] = \mathit{send}\ c\ 1\,;\,[\![P]\!]$$
$$[\![c \rhd [l_1 : P, l_2 : Q]]\!] = \mathbf{let}\ x = \mathit{recv}\ c \ \mathbf{in}\ (\mathbf{case}\ x\ \mathbf{of}\ 0 \mapsto [\![P]\!], \mathsf{suc}\ n \mapsto [\![Q]\!])$$
$$[\![\mathbf{if}\ [V = 0]\ \mathbf{then}\ P\ \mathbf{else}\ Q]\!] = \mathbf{case}\ V\ \mathbf{of}\ 0 \mapsto [\![P]\!], \mathsf{suc}\ n \mapsto [\![Q]\!]$$

To select label l_1, the encoding sends 0 then continues as $[\![P]\!]$ with 1 sent instead for label l_2. Subeffecting (Def. 14) provides the effect $c :![\mathbf{nat}].(\mathsf{S} + \mathsf{T})$ for the l_1 select (where $c : \mathsf{S}$ in the effect of $[\![P]\!]$) and $c :![\mathbf{nat}].(\mathsf{T} + \mathsf{S})$ in the l_2 select. Since the encoding is type-directed, subeffecting rules are generated to match the session types. Branching has the dual behaviour of type $c :?[\mathbf{nat}].(\mathsf{S} + \mathsf{T})$ where a natural number is received and matched upon with $[\![P]\!]$ and $[\![Q]\!]$ for the branches of **case**. The encoding for **if** is similar, but each branch must have matching effects (no subeffecting).

Weakening (introducing either $c : \mathbf{end}$ or $c : *![S]$) is encoded via subeffecting: if $[\![\Gamma;\Delta \vdash P]\!] = [\![\Gamma]\!] \vdash M : \mathbf{unit}, [\![\Delta]\!]$ then $[\![\Gamma]\!] \vdash M : \mathbf{unit}, ([\![\Delta]\!], c : \mathbf{end})$ via subeffecting (eq.15) with $[\![\Delta]\!] \sqsubseteq ([\![\Delta]\!], c : \mathbf{end})$ (and similarly for output).

Value encoding is fairly direct as PCF has the same constructors as the session calculus (modulo pred): $[\![\mathsf{unit}]\!] = \mathsf{unit}$, $[\![\mathsf{suc}\ V]\!] = \mathsf{suc}\ [\![V]\!]$, $[\![\mathsf{0}]\!] = \mathsf{0}$, $[\![\mathsf{pred}\ V]\!] = \mathbf{case}\ [\![V]\!]\ \mathbf{of}\ 0 \mapsto \mathsf{0}, (\mathsf{suc}\ x) \mapsto x$, and $[\![v]\!] = v$ by the context-preserving embedding of variables.

7.2 Correctness

We define *stable reductions* for our *FPCF* instance, which characterise the equivalent of β-reductions (synchronisation) in the session calculus. This requires the notion of *top-level* contexts, in which the terms of a (session) β-redex from our encoding reside:

Definition 16 (Top-level contexts)**.**

$$\mathbb{T} ::= \mathbf{let}\ x = \mathbb{T}\ \mathbf{in}\ M \mid \mathit{new}(\lambda x\lambda y.\mathbb{T}) \mid \mathbb{T} \parallel \mathbb{T} \mid \mathbb{T} \parallel M \mid M \parallel \mathbb{T} \mid [-]$$

where a hole is denoted $[-]$. Note that multiple holes can occur in a top-level context due to the parallel composition $\mathbb{T} \parallel \mathbb{T}$.

Definition 17. For the session instantiation of *FPCF*, the relation $\Longrightarrow$ provides *stable reductions* between pairs of a term and store:

$$(\beta\Rightarrow)\ \ \langle \mathbb{T}[\mathit{send}\ \mathbf{c}\ V][\mathit{recv}\ \overline{\mathbf{c}}], s\rangle \Longrightarrow \langle \mathbb{T}[\mathsf{unit}][V], s\rangle$$
$$(\mathrm{ch}\beta\Rightarrow)\ \ \langle \mathbb{T}[\mathit{chSend}\ \mathbf{c}\ \mathbf{d}][\mathit{chRecv}\ \overline{\mathbf{c}}], s\rangle \Longrightarrow \langle \mathbb{T}[\mathsf{unit}][(\lambda k.k\ \mathbf{d})], s\rangle$$
$$(\mathrm{new}\Rightarrow)\ \frac{\langle M[\mathbf{c}/c, \overline{\mathbf{c}}/\overline{c}], s \blacktriangleleft [c \mapsto \epsilon, c' \mapsto \epsilon]\rangle \Longrightarrow \langle N[\mathbf{c}/c, \overline{\mathbf{c}}/\overline{c}], s'\rangle}{\langle \mathit{new}\ (\lambda c\lambda c'.M), s\rangle \Longrightarrow \langle \mathit{new}\ (\lambda c, c', N), s'\rangle}$$
$$(\equiv\Rightarrow)\ \frac{M \equiv N \quad \langle N, s\rangle \Longrightarrow \langle N', s'\rangle}{\langle \mathbb{T}[M], s\rangle \Longrightarrow \langle \mathbb{T}[N'], s'\rangle}\ (\rightarrow)\ \frac{M \rightarrow^* N}{\langle \mathbb{T}[M], s\rangle \Longrightarrow \langle \mathbb{T}[N], s\rangle}$$

where $\blacktriangleleft$ is the union of two finite maps with a left-bias, $[c \mapsto t] \blacktriangleleft [c \mapsto s] = [c \mapsto t]$ but $\emptyset \blacktriangleleft [c \mapsto s] = [c \mapsto s]$. In the premise of $(\rightarrow)$, only the pure β-reductions (from Section 2.1) are allowed. A similar rule to $(\mathrm{ch}\beta\Rightarrow)$ is provided where *rsend* replaces *chSend*.

Let Δ be *closed* if $\forall c \in \mathit{dom}(\Delta)$ then $\overline{c} \in \mathit{dom}(\Delta)$. A store s is *well-formed* with respect to Δ, $\mathsf{wf}(\Delta, s)$, if $\mathit{dom}(\Delta) \subseteq \mathit{dom}(s)$.

Lemma 6. Let $\Gamma;\Delta \vdash P$ where Δ is closed and balanced, and $\mathsf{wf}(\Delta, s)$, then $\langle [\![P]\!]\theta, s\rangle \Longrightarrow \langle N, s'\rangle \Rightarrow \langle [\![P]\!]\theta, s\rangle \rightarrow^+ \langle N, s'\rangle$ where θ is a substitution of free channel variables for channel values of the corresponding name in the store s.

Further, $\langle [\![P]\!]\theta, s\rangle \rightarrow \langle N, s'\rangle \Rightarrow (\exists M, s''.\langle [\![P]\!]\theta, s\rangle \Longrightarrow \langle M, s''\rangle \wedge \langle N, s'\rangle \rightarrow^* \langle M, s''\rangle)$. Thus, under the image of our encoding and session typing, stable reductions (synchrony) and the operational semantics coincide.

Theorem 5 (Operational correspondence)**.** Then $\forall \Gamma, \Delta, P$.

- (sound) $\forall Q.\ \ \Gamma;\Delta \vdash P \wedge P \rightarrow Q \wedge \mathsf{wf}(\Delta, s) \Rightarrow \exists s', M, \theta.\langle [\![P]\!]\theta, s\rangle \Longrightarrow \langle M\theta, s'\rangle \wedge M \equiv [\![Q]\!]$
- (complete) $\forall s', \theta.\ \ \Gamma;\Delta \vdash P \wedge \langle [\![P]\!]\theta, s\rangle \Longrightarrow \langle M\theta, s'\rangle \wedge \Delta$ balanced & closed $\Rightarrow \exists Q.P \rightarrow Q \wedge M \equiv [\![Q]\!]$

For soundness, session calculus β-reductions correspond to $(\beta\Rightarrow)$ and $(ch\beta\Rightarrow)$, combined with $(\equiv\Rightarrow)$ for replication, branching, and selection. For **if**, $(\rightarrow)$ is used. Reduction under *new* corresponds to (new). Reduction under parallel composition and extension of reduction along structural congruence corresponds to $(\equiv\Rightarrow)$. Completeness similarly relates stable and session calculus reductions.

```
class Effect (m :: ef -> * -> *) where
   type Unit m :: ef
   type Plus m (f :: ef) (g :: ef) :: ef
   return :: a -> m (Unit m) a
   (>>=) ::  m f a -> (a -> m g b) -> m (Plus m f g) b
instance Effect Process where
  type Plus Process f g = SeqUnion f g
  type Unit Process     = '[]

class Sub m f g where sub :: m f a -> m g a
instance Sub Process f g =>
     Sub Process ((c :-> s) ': f) ((c :-> s :+ t) ': g)
```

Figure 4. Effect-graded monad and `Process` instances

Remark 4 (Termination and replicated input)**.** Replicated input is encoded as a recursive function which eventually becomes blocked once there are no more outputs (*rsend*). The same occurs in the π-calculus: any replicated inputs persist at the end of the computation. The *garbage collection* property shows this is observationally equivalent to the empty process: $\nu c.(*c?(d).P) \cong \mathbf{0}$. This property holds for the encoding up to non-termination effects.

8. Implementation

We use the encoding of the session calculus into *FPCF* to derive a new implementation of session types in Haskell. The gap between *FPCF* and Haskell is bridged using recent work to embed effect systems into Haskell types [39]. The implementation provides a proof-of-concept use for the encoding (rather than a polished user-friendly library). A brief description is given here, but more information (and the code) can be found on the artifact webpage `http://dorchard.co.uk/popl16`. Section 9.3 compares our implementation to existing work.

Effect systems in Haskell Haskell's **do**-notation provides a specialised form of *let*-binding for sequentially composing effectful computations represented via monads. Whilst Haskell does not have a user-visible effect system, monads can be generalised to *graded monads* to carry effect information as a type index [30, 39, 40]. Furthermore, Wadler and Thiemann showed [54] that an impure λ-calculus with an effect system (similar to *FPCF*) can be encoded into a *monadic metalanguage* (à la Moggi [34])– essentially Haskell's **do**-notation. Thus, *FPCF* session terms can be translated into Haskell programs where the graded monad embeds effects.

Figure 4 gives a Haskell definition for graded monads via the `Effect` type class over binary type constructors `m :: ef -> * -> *` (mapping from `ef`, the *kind* of effect annotations, to a unary type constructor) where `ef` models a set of effect annotations $\mathcal{F}$. A value of type `m` f a thus denotes a computation with effects described by the type index f. The domain `ef` is equipped with a type-level binary function `Plus m` implementing $\bullet$ of the effect algebra for $\mathcal{F}$ and a constant `Unit m` providing the unit element I. The `return` operation of the graded monad lifts a value to a trivially effectful computation, marked with I. The "bind" operation (`>>=`) provides the sequential composition of effectful computations. This is used by Haskell to desugar the **do**-notation, giving the typing:

$$\text{bind}\ \frac{\Gamma \vdash e_1 : \texttt{m}\ F\ \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \texttt{m}\ G\ \tau}{\Gamma \vdash (\texttt{do x <- } e_1\texttt{; } e_2)\ : \texttt{m (Plus m } F\ G\texttt{)}\ \tau}$$

The (bind) rule models the *let*-binding of *FPCF*, propagating effect information in the same way. By Wadler and Thiemann's translation, an *FPCF* judgment $\Gamma \vdash M : \tau, F$ maps to a monadic metalanguage judgment $\Gamma \vdash [M] : \texttt{m}[F]\,\tau$, embedding effects into types. We use this graded monadic embedding along with Haskell's advanced type system features (*e.g.*, *closed type families* [17] and *data kinds* [55]) to implement the encoding of Section 7 on top of the core Concurrent Haskell library.

Session effects We provide a graded monad instance (shown partially in Figure 4) for the `Process` data type which encapsulates concurrent computations:

```
data Process (s :: [Map Name Session]) a = Proc (IO a)
```

The first parameter `s` is a type-level finite map modelling an environment of session type information. This is of the form `'[c :-> s, d :-> t, ...]` describing an environment where a channel `c` has session type `s`, a channel `d` has session type `t`, and so on. Session environments are composed sequentially via the `SeqUnion` type-level function which models $\bullet$ from Section 7.

Session effects are modelled by the `Session` data type:

```
data Session = forall a . a :! Session    -- send
             | forall a . a :? Session    -- receive
             | forall a . a :*! Session   -- output
             | Session :+ Session    -- alternation
             | Bal Session  | End    -- 'balanced' & end
             | Fix Session Session   -- -* prefix
```

Each concurrent *FPCF* operation from the previous section has a Haskell counterpart, *e.g.*, sending and receiving ground values:

```
send  :: Chan c -> t -> Process '[c :-> t :! End] ()
recv  :: Chan c      -> Process '[c :-> t :? End] t
```

where `Chan c` is a channel named `c` (a type-level symbol). Channels are implemented via Concurrent Haskell channels. Though Concurrent Haskell channels have a single (boxed) type, they are used at any type with unchecked casts in the implementation of send/receive operations. This is proven safe by session-type duality, which is encoded as a type predicate (type-class constraint). For example, the `new` combinator enforces duality of sessions over the restricted channel `c` via the `Duality` type class:

```
new :: (Duality env c)
    => ((Chan (Ch c), Chan (Op c)) -> Process env t)
    -> Process ((env :\ (Op c)) :\ (Ch c)) t
```

where `:\` deletes a channel from an environment. Some operations have a slightly different (but isomorphic) form to their *FPCF* counterparts, managing environments via type functions, *e.g.* `chRecv`:

```
chRecv :: Chan c ->
      Process '[c :-> (Delg (e :@ d)) :? End]
         ((Chan d -> Process e t) -> Process (e :\ d) t)
```

where `:@` looks up a session type by the channel name.

Since Haskell does not have subtyping, subeffecting is explicit using `sub :: Sub f g => m f a -> m g a`, with one instance shown in Figure 4 (computing `:+` as an upper bound). This models the subeffecting relation in Definition 14 (p. 9). Relatedly, Haskell does not have equirecursive types, so the implementation restricts recursion to only definitions that induce an affine effect equation. A specialised combinator `affineFix` is used, where the fixed-point a^*b (represented by `Fix a b`) is computed via a type-level function given an affine effect equation $s \mapsto a \bullet s + b$.

Example 5. The following simple example corresponds to session calculus term $\nu c.(\nu d.(c!\langle d\rangle.\overline{d}?(\mathsf{Ping})) \mid \overline{c}?(x).x!\langle\mathsf{Ping}\rangle)$:

```
client (c :: (Chan (Ch "c")))
    = new (\(d :: (Chan (Ch "d")), d') ->
                  do  chSend c d
                      Ping <- recv d'
                      print "Client: got a ping")

server c = do { k <- chRecv c; k (\x -> send x Ping) }
process = new (\(c, c') -> (client c) `par` (server c'))
```

where `client` models the left process and `server` the right. The type of `client` is inferred as: `client :: Chan (Ch "c") -> Process '[Ch "c" :-> (Delg (Msg :! End) :! End)] ()`

9. Extensions and related works

9.1 Polymorphism for state with first-class references

Early effect systems targeted stateful computations with first-class references, as in ML [20]. We can instantiate our encoding for the more general setting of first-class references. This relies on extending our session calculus with *session polymorphism* [5, 6].

Effects are $\mathcal{F} = \mathcal{P}(\{\mathsf{rd}\,\rho\,\tau, \mathsf{wr}\,\rho\,\tau, \mathsf{alloc}\,\rho\,\tau | \forall \rho, \tau\})$, where ρ are *regions*, with $(\mathcal{F}, \cup, \emptyset), \oplus = \cup$, and $F^* = F$. We add reference types $\mathbf{ref}_\rho\,\tau$ tagged with their region and type, and constants:

$$\mathbf{get} : \mathbf{ref}_\rho\,\tau \xrightarrow{\{\mathsf{rd}\,\rho\,\tau\}} \tau \qquad \mathbf{put} : \mathbf{ref}_\rho\,\tau \to \tau \xrightarrow{\{\mathsf{wr}\,\rho\,\tau\}} \mathbf{unit}$$
$$\mathbf{new} : \tau \xrightarrow{\{\mathsf{alloc}\,\rho\,\tau\}} \mathbf{ref}_\rho\,\tau \quad \textit{with fresh } \rho$$

The idea behind the encoding is that each mutable store has its own handler. Thus, when **new** creates a reference (pointing to a new store), a new state handler is created. References are then encoded as channels to interact with this handler. A central handler (the main handler for the encoding) forwards requests to the state handlers by means of the reference channels. The central handler is defined:

$$\begin{aligned} *?h(c).c?(x_\rho).c \triangleright \{&\mathsf{alloc} : \overline{h}!\langle c\rangle,\\ &\mathsf{act} : c?(r).c \triangleright \{\mathsf{rd} : r \lhd \mathsf{rd}.r?(x).c!\langle x\rangle.\overline{h}!\langle c\rangle,\\ &\qquad\qquad \mathsf{wr} : r \lhd \mathsf{wr}.c?(y).r!\langle y\rangle.\overline{h}!\langle c\rangle\}\} \end{aligned}$$

A value is received (bound to x_ρ) which represents a region. There are then two behaviours: the alloc behaviour records when a new reference is created and act forwards effectful operations to other state handlers. The act mode receives the reference channel r and then forwards the get/put requests on c to r to interact with that mutable cell. The encoding of the **new** operation is then:

$$\begin{aligned} (\!|\,\mathbf{new}\ M\,|\!)^{\mathsf{ei},\mathsf{eo}}_r = \nu e, q, \mathsf{ea}.((\!|\,M\,|\!)^{\mathsf{ei},\mathsf{ea}}_q &\mid \overline{q}?(x).\mathit{Var}\langle\overline{e}, x\rangle \mid r!\langle e\rangle\\ &\mid \mathsf{ea}?(c).c!\langle\rho\rangle.c \lhd \mathsf{alloc}.\overline{\mathsf{eo}}!\langle c\rangle) \end{aligned}$$

where *Var* refers to a simple state handler (from Example 2). Thus, the initial value M is received as x and is used to start a new simple, state handler with a new effect channel $\overline{e}$. The opposite end-point e is returned by sending on r. The effectful behaviour is to send a value ρ of a fresh singleton type $\hat{\rho}$ and then request the alloc behaviour from the central handler (which does nothing). The **get** operation is defined as follows (**put** is similar):

$$\begin{aligned} (\!|\,\mathbf{get}\ M\,|\!)^{\mathsf{ei},\mathsf{eo}}_r = \nu q, \mathsf{ea}.((\!|\,M\,|\!)^{\mathsf{ei},\mathsf{ea}}_q\\ \mid \overline{q}?(r_c).\mathsf{ea}?(c).c!\langle\rho\rangle.c \lhd \mathsf{act}.c!\langle r_c\rangle.c \lhd \mathsf{rd}.c?(x).r!\langle x\rangle.\overline{\mathsf{eo}}!\langle c\rangle) \end{aligned}$$

The result of the encoded reference value M is an effect channel received on q as r_c. The central effect channel is received on ea and the corresponding ρ value is sent to mark the type of the region (since the rules are type directed, the value ρ comes from the type of M, elided here). The act behaviour is then chosen before c sends the reference channel r_c and the rest of the interaction is as before for simple state.

This encoding requires session polymorphism so that differently typed references can be handled by the central handler. The polymorphic session type for the effect channel c of $H(c)$ is:

$$\begin{aligned} c : \mu\alpha.\ \forall\rho.\ ?[\rho].\&[\mathsf{alloc} : \alpha, \mathsf{act} : \forall\tau.\\ ?[\mu\alpha.\ \oplus[\mathsf{rd} :?[\tau].\alpha, \mathsf{wr} :![\tau].\alpha]].\&[\mathsf{rd} :![\tau].\alpha, \mathsf{wr} :?[\tau].\alpha]] \end{aligned}$$

Note the polymorphic region type ρ and type τ for the reference value. We show the interpretation of effect annotations for a causal version of the above system for brevity, but this can easily be converted to the set-based style following the approach of Example 3

$$\begin{aligned} [\![(\mathsf{rd}\,\rho\,\tau) : F]\!] &= ![\![\rho]\!].\oplus[\mathsf{act} :![\![\mathbf{ref}_\rho\,\tau]\!].\oplus[\mathsf{rd} :?[\![\tau]\!].[\![F]\!]]]\\ [\![(\mathsf{wr}\,\rho\,\tau) : F]\!] &= ![\![\rho]\!].\oplus[\mathsf{act} :![\![\mathbf{ref}_\rho\,\tau]\!].\oplus[\mathsf{wr} :![\![\tau]\!].[\![F]\!]]]\\ [\![(\mathsf{alloc}\,\rho\,\tau) : F]\!] &= ![\![\rho]\!].\oplus[\mathsf{alloc} : [\![F]\!]] \end{aligned}$$

where $[\![\mathbf{ref}_\rho\,\tau]\!] = \mu\alpha.\ \oplus[\mathsf{rd} :?[\![\tau]\!].\alpha, \mathsf{wr} :![\![\tau]\!].\alpha]$, *i.e.*, the type of an effect channel for interacting with a handler.

9.2 Monadic metalanguage for effects

We considered here an impure variant of *FPCF* where any term may be effectful. Our effect systems therefore give effect annotations to every term. An alternate presentation of effectful calculi takes Moggi's monadic metalanguage [34] and augments the monadic type constructor T with an effect annotation [54] (discussed briefly in Section 8). In this approach, effects are limited to a subset of the syntax and hence their scope is more easily delimited. A monadic metalanguage variant of PCF (call it *metaFPCF*)[1] would extend the syntax and type system of pure PCF with $M, N ::= \ldots \mid \mathbf{let}\,x \leftarrow M;\ N \mid \langle M\rangle$ and typing rules:

$$\text{let}\ \frac{\Gamma \vdash M : T_f\,\sigma \qquad \Gamma, x : \sigma \vdash N : T_g\tau}{\Gamma \vdash \mathbf{let}\,x \leftarrow M;\ N : T_{(f \bullet g)}\tau} \qquad \text{unit}\ \frac{\Gamma \vdash M : \tau}{\Gamma \vdash \langle M\rangle : T_I\tau}$$

for an effect monoid $(\mathcal{F}, \bullet, I)$. The type constructor T takes two arguments, the first an effect annotation, the second the type of the value produced by the computation. The **let** construct provides composition of effectful computations and $\langle M\rangle$ raises a pure term to a trivially effectful term. This is essentially what we have in our Haskell implementation (Section 8).

Our encoding from *FPCF* to the session calculus can be reworked for *metaFPCF*. Since the metalanguage style for effects separates more clearly in the type system what is definitely pure from what is potentially effectful, we could combine existing typed encodings of the pure λ-calculus (such as that of [51]) with an encoding for the effectful parts (*let* and *unit* above). The encoding of (*let*) above would be the same as the encoding for $\mathbf{let}\ x = M\ \mathbf{in}\ N$ shown at the start of Section 3. The encoding for (*unit*) is similar to values in our calculus (variables and constants) with $(\!|\,\langle M\rangle\,|\!)^{\mathsf{ei},\mathsf{eo}}_r = \mathsf{ei}?(c).\overline{\mathsf{eo}}!\langle c\rangle \mid [\![M]\!]_r$ where $[\![M]\!]_r$ is any other (sound) typed encoding of PCF, perhaps building on the many possible encodings in the literature (*e.g.*, [33, 49, 51], with various trade-offs).

However, we cannot take one of these existing pure typed-encodings and use it unmodified in combination with the encoding of (*let*) and (*unit*). Consider a pure encoding of the λ-calculus part of *metaFPCF* via $[\![-]\!]_r$ defined for functions as:

$$[\![\lambda x.M]\!]_r = \nu d.\ (r!\langle\overline{d}\rangle.\ *d?(p, q).p?(x).[\![M]\!]_q)$$

This is the same as in Section 3, but with the effect channel carriers erased. The interpretation of the monadic fragment is written $(\!|-|\!)^{\mathsf{ei},\mathsf{eo}}_r$. Consider then the following encoding of a term:

$$\begin{aligned} [\![\mathbf{let}\ f \leftarrow \langle\lambda x.\mathit{put}\ x\rangle;\ f\ 0]\!]^{\mathsf{ei},\mathsf{eo}}_r = \nu\mathsf{ea}, q.(\mathsf{ei}?(c).\overline{\mathsf{ea}}!\langle c\rangle \mid\\ \nu y.(q!\langle\overline{y}\rangle.*y?(p, r').p?(x).[\![\mathit{put}\ x]\!]_{r'}) \mid \overline{q}?(f).(\!|\,f\ 0\,|\!)^{\mathsf{ea},\mathsf{eo}}_r) \end{aligned}$$

Whilst the function body is effectful, the encoding here tries to apply the pure embedding to put. However put requires access to an effect channel, which is bound nowhere in the scope of the encoded function. Instead, a type-directed encoding is needed where pure constructors (such as abstraction and application) have a different encoding if they apply or create functions with target type $T_f\,A$ for some f, A. Therefore, a pure encoding cannot be used exactly "as is" with the rest of our encoding for effects.

Future work is to investigate further whether it is possible to factor our encoding through a monadic structuring of effects. The work of Toninho *et al.* provides a Curry-Howard correspondence between session types and linear logic [51], which provides a way to consider more traditional monadic encodings of effects. An interesting avenue might be to unify our encoding with recent work on a monadic integrations of session types and processes [52].

9.3 Related work

We previously showed an encoding of a simple first-order imperative language with a simple effect system for state into a ver-

[1] Filinski calls this *Effect*-PCF [18], but this is not be confused with *FPCF*.

sion of the session calculus (without $*!/*?$ types but only recursive types) [42]. This paper greatly expanded [42], extending the encoding to the higher-order setting of PCF with non-sequential control flow, and considering the reverse encoding and implementation.

Communication effects Effect systems have previously been used to describe communication effects for CML [28, 38]. These effect systems were *causal* (similar to our earlier state example), defined over lists of tokens $E ::= \rho![\tau] \mid \rho?[\tau] \mid \mathsf{spawn}\,[E]$ denoting sending on a channel ρ, receiving on a channel ρ of type τ and spawning a new thread. Alternation and recursion where also included, similar in structure to our effect algebra (Definition 1).

These communication effect systems resemble our instantiation of *FPCF* to encode the session calculus, but restricted to just sending and receiving of values. In this first-order setting, the above system can be translated to our *FPCF* encoding by grouping all actions on a particular channel into a single session type per channel *e.g.*, a communication effect $\rho_1![\tau_1].\rho_2?[\tau_2].\rho_1![\tau_2]$ can be mapped to a session effect environment $\rho_1 :![\tau_1].![\tau_2].\mathbf{end}, \rho_2 :?[\tau_2].\mathbf{end}$.

There is no reverse encoding from session types into communication effects though since session types do not describe the relative causality between channels, which is recorded by communication effects. Session types are however more expressive with respect to the higher-order communication (delegation). Future work is to explore the relative expressive power further.

Encodings of functions into typed processes Types limit the contexts in which processes can interact, therefore typed equivalences usually offer a *coarser* semantics than untyped semantics, where stronger properties can be proved. For example, Pierce and Sangiorgi [44] demonstrate that the barbed congruence under IO-subtyping can justify the correctness of the optimal encoding of the λ-calculus by Milner [33]. This was not possible in the polyadic π-calculus [33]. After [44], many works on typed π-calculi have investigated correctness of encodings in order to examine semantic consequences of proposed typing systems. Our work follows this tradition, studying properties of typed calculi and their encodings.

Session types are closely related to linear typing disciplines. In [4, 5, 56], typed equivalences of a family of linear and affine calculi were used to encode PCF and System F fully abstractly [21]. A subsequent work [24] adapted these linear types in a practical direction. It proposed new typing systems for secure higher-order and multi-threaded programming languages. In these works, typed properties and linearity play a fundamental role in the analysis. In general, linear types or session types are suitable to encode "sequentiality" in the sense of [1, 26], as shown in Section 3 and 4.

Wadler shows a tight correspondence between a linear functional language with session types "GV" and a session-typed process calculus "CP" (whose types correspond to classical linear logic propositions), via an encoding of GV into CP [53]. Lindley and Morris later provided the reverse encoding, from CP to GV, showing operational correspondence [32]. Theses works are similar to our own in that they give encodings between a functional language and process calculi. Our functional language *FPCF* differs to GV in that it is not fundamentally linear, though linearity for sessions is implemented via the effect algebra and typing.

Encodings related to session-typed processes. The works [14, 15] study encodings of binary session calculi into a linearly typed π-calculus. While [15] gives a full abstract encoding of a session calculus into a linear calculus (an extension of [4]), the work [14] gives the operational correspondence for the first- and higher-order π-calculi into [31]. By [15], we can encode our session calculus into an extension of [4]; however to encode effect systems, we require a sequence of linear types (sessions) as well as recursive types (for typing effect handlers). Note that [14, 15] investigates embedability of two different type systems of the π-calculus, whose main aims differ from ours.

Among works on the session types, the most related work is [51] which elegantly proves the operational correspondence between a simply typed λ-calculus and a session calculus via a Curry-Howard interpretation. Another work [43] explores a typed behavioural theory for their logically motivated binary session calculus. In [6], they extend these works to polymorphism and parametricity. They demonstrate the importance of encodings into session calculi for a fine-grained analysis of higher-order functions.

Our work differs from the above; we extend the encoding from simple types to effect systems, and we give a reverse encoding (from the π-calculus back to PCF), giving also an implementation.

Sessions in Haskell There are four relevant works adding session types to Haskell: by Neubauer and Thiemann [37], Sackman and Eisenbach [48], Pucella and Tov [47], and Imai *et al.* [27].

Both [47, 48] use a *parameterised monad* [2] indexed by type-level pre- and post-conditions on session environments, enforcing linearity of channel usage. Our graded monadic effects instead specify a change to the environment, rather than a pre-post condition. The approach in [37] instead threads a single channel implicitly through a computation to avoid aliasing, ensuring linearity.

In terms of features, both [37, 47] have first-order sessions with branch/select and recursion, but without delegation which Imai *et al.* [27] and we include; [48] allows more flexible primitives, but session types must be manually constructed. In our work, these are mostly inferred. In [47], session environments are stacks (built with tuples) requiring manual manipulation to access sessions, essentially indexing sessions by their position. This has the disadvantage that the programmer must perform context management themselves. In contrast, our approach uses a finite map representation allowing indexing by name, rather than position. This however requires the user to give fresh names via type signatures.

Imai *et al.* provide a more convenient system for manipulating the multi-channel session environments of [47] using de-Bruijn indexed heterogeneously-typed lists [27]. This does not require manual manipulation of the stack nor normalising a type-level representation of finite maps. Future work is to explore their approach combined with our effect-based embedding.

9.4 Concluding remarks

Future work is to explore notions of effect which change the control flow, such as *exceptions*. Previous work added exceptions to session types via an *escape* or *interrupt* mechanism [8, 9, 16]. We plan to investigate their relationship. Other further work is to explore applications, *e.g.*, using our encoding of *FPCF* as an optimisation step for compilation, providing implicit parallelism optimisations informed by the encoded effect information.

Our encodings have shown that the algebraic structure of rich effect systems and session types is very similar, with analogous components for sequentiality, choice, recursion, and subtyping in each. Session types may seem more fine grained, but the same level of information can be captured in an effect system (Section 7). This raises the question: *are effects and sessions in fact equivalent*? (or at least isomorphic)? To answer this question, we would need to show that our encodings are mutually inverse. Exploring this is future work. If this is the case, we may be moving towards a new unified, typed calculus for general effectful and concurrent programming.

Acknowledgements We are grateful to Bernardo Toninho for his comments, Bernardo and Julien Lange for their feedback on the artifact, and Martin Berger for discussion. Thanks also to the anonymous reviewers for their helpful comments. The work has been partially sponsored by EPSRC EP/K011715/1, EP/K034413/1, EP/L00058X/1, and EU project FP7-612985 UpScale.

References

[1] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *TCS*, 163:409–470, 2000.

[2] Robert Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19(3-4):335–376, 2009.

[3] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *JLAMP*, 84(1):108–123, 2015.

[4] Martin Berger, Kohei Honda, and Nobuko Yoshida. Sequentiality and the π-calculus. In *Proc. TLCA'01*, volume 2044 of *LNCS*, pages 29–45, 2001.

[5] Martin Berger, Kohei Honda, and Nobuko Yoshida. Genericity and the pi-calculus. *Acta Inf.*, 42(2-3):83–141, 2005.

[6] Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. Behavioral polymorphism and parametricity in session-based communication. In *ESOP*, volume 7792 of *LNCS*, pages 330–349. Springer, 2013.

[7] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010.

[8] Sara Capecchi, Elena Giachino, and Nobuko Yoshida. Global escape in multiparty sessions. *MSCS*, 29:1–50, 2015.

[9] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured interactional exceptions in session types. In *CONCUR*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008.

[10] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *TOPLAS*, 34:8:1–8:78, 2012.

[11] Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. On the preciseness of subtyping in session types. In *PPDP'14*, pages 146–135. ACM Press, 2014.

[12] Ezgi Çiçek, Deepak Garg, and Umut Acar. Refinement types for incremental computational complexity. In *ESOP*, pages 406–431. Springer, 2015.

[13] Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *ACM SIGPLAN Notices*, volume 43, pages 133–144. ACM, 2008.

[14] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *PPDP'12*, pages 139–150. ACM, 2012.

[15] Romain Demangeon and Kohei Honda. Full abstraction in a subtyped pi-calculus with linear types. In *CONCUR*, volume 6901 of *LNCS*, pages 280–296. Springer, 2011.

[16] Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: Distributed dynamic verification with multiparty session types and Python. *Formal Methods in System Design*, 46(3):197–225, 2015.

[17] Richard A Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations. In *Proceedings of POPL 2014*, pages 671–684, 2014.

[18] Andrezej Filinski. Controlling effects. Technical report, DTIC Document, 1996.

[19] Simon Gay and Malcolm Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.

[20] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *LFP'86*, pages 28–38. ACM, 1986.

[21] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. CUP, 1989.

[22] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[23] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *TCS*, 151(2):437–486, 1995.

[24] Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. *TOPLAS*, 29(6), 2007.

[25] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *POPL*, pages 273–284. ACM, 2008. a full version will appear in *JACM*.

[26] Martin Hyland and Luke Ong. On full abstraction for PCF. *Inf. & Comp.*, 163:285–408, 2000.

[27] Keigo Imai, Shoji Yuen, and Kiyoshi Agusa. Session Type Inference in Haskell. In *Proc. of PLACES*, volume 69 of *EPTCS*, pages 74–91, 2010.

[28] Pierre Jouvelot and David K. Gifford. Communication Effects for Message-Based Concurrency. Technical report, MIT, 1989.

[29] Ohad Kammar and Gordon D Plotkin. Algebraic foundations for effect-dependent optimisations. In *ACM SIGPLAN Notices*, volume 47, pages 349–360. ACM, 2012.

[30] Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In *POPL*, pages 633–646. ACM, 2014.

[31] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the Pi-Calculus. *TOPLAS*, 21(5):914–947, September 1999.

[32] Sam Lindley and Garrett Morris. A semantics for propositions as sessions. In *ESOP*, pages 560–584. Springer, 2015.

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

[34] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[35] Dimitris Mostrous and Nobuko Yoshida. Session typing and asynchronous subtyping for the higher-order π-calculus. *Inf. Comput.*, 241:227–263, 2015.

[36] Alan Mycroft, Dominic Orchard, and Tomas Petricek. Effect systems revisited– control-flow algebra and semantics. volume 9560 of *LNCS*. Springer, 2016.

[37] Matthias Neubauer and Peter Thiemann. An Implementation of Session Types. In *PADL*, volume 3057 of *LNCS*, pages 56–70. Springer, 2004.

[38] Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In *Correct System Design*, pages 114–136. Springer, 1999.

[39] Dominic Orchard and Tomas Petricek. Embedding effect systems in Haskell. In *Haskell Symposium*, pages 13–24, 2014.

[40] Dominic Orchard, Tomas Petricek, and Alan Mycroft. The semantic marriage of monads and effects. *arXiv:1401.5391*, 2014.

[41] Dominic Orchard and Nobuko Yoshida. Effects as Sessions, Sessions as Effects (with appendices). Technical Report 2015/5, Department of Computing, Imperial College London, UK, 2015. `http://www.doc.ic.ac.uk/research/technicalreports/2015/DTR15-5.pdf`.

[42] Dominic Orchard and Nobuko Yoshida. Using session types as an effect system. In *Post-proceedings of PLACES*, EPTCS, 2015.

[43] Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations and observational equivalences for session-based concurrency. *Inf. Comput.*, 239:254–302, 2014.

[44] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *MSCS*, 6(5):409–454, 1996.

[45] Benjamin Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of ACM*, 47(3):531–584, 2000.

[46] Gordon Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

[47] Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *Proc. of Haskell symposium '08*, pages 25–36. ACM, 2008.

[48] Matthew Sackman and Susan Eisenbach. Session Types in Haskell (Updating Message Passing for the 21st Century), 2008. Technical report, Imperial College London.

[49] Davide Sangiorgi and David Walker. *The π-Calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[50] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Proc. LICS'92*, pages 162–173, 1992.

[51] Bernardo Toninho, Luís Caires, and Frank Pfenning. Functions as Session-Typed Processes. In *FoSSaCs*, LNCS, pages 346–360. Springer, 2012.

[52] Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *ESOP*, volume 7792 of *LNCS*, pages 350–369. Springer, 2013.

[53] Philip Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014.

[54] Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4:1–32, January 2003.

[55] Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Proc. of TLDI*, pages 53–66. ACM, 2012.

[56] Nobuko Yoshida, Martin Berger, and Kohei Honda. Strong normalisation in the pi -calculus. *Inf. Comput.*, 191(2):145–202, 2004.

[57] Nobuko Yoshida and Vasco Thudichum 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.