Citation for published version


DOI

https://doi.org/10.1145/2837614.2837634

Link to record in KAR

http://kar.kent.ac.uk/57481/

Document Version

Publisher pdf

Copyright & reuse

Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research

The version in the Kent Academic Repository may differ from the final published version. Users are advised to check http://kar.kent.ac.uk for the status of the paper. Users should always cite the published version of record.

Enquiries

For any further enquiries regarding the licence status of this document, please contact:

researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at http://kar.kent.ac.uk/contact.html
Effects as Sessions, Sessions as Effects

Dominic Orchard
Imperial College London, UK
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 \( \lambda \)-calculus and its variants, the latter for the \( \pi \)-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-type \( \pi \)-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; E.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—Type structure

Keywords

session types, \( \pi \)-calculus, effect systems, PCF, encoding, type systems, Concurrent Haskell

1. Introduction

The simple type theory for the \( \lambda \)-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 \( \lambda \)-calculus and session types in the \( \pi \)-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 \( \pi \)-calculus, with types for sequencing (via prefixing), branching, and recursion.

The \( \pi \)-calculus and the \( \lambda \)-calculus are simple yet powerful prototypes of computation. But they do not stand apart. The \( \pi \)-calculus subsumes the power of the \( \lambda \)-calculus, and is universal with respect to sequential computation. This result was first proved by Milner [33], opening up the use of \( \lambda \)-calculus encodability to show the expressiveness of various type disciplines for the \( \pi \)-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 \( \lambda \)-terms are encodable more tightly into session-type processes via a Curry-Howard interpretation, providing a new logical explanation of sharing and copying parallel \( \lambda \)-evaluation strategies.

This paper goes further: we show that the session-typed \( \pi \)-calculus is expressive enough to systematically encode various classes of effect system for the \( \lambda \)-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:

\[
\text{def } \text{Var}(c, x) = c \triangleright \{ \text{get } : c!(x), \text{Var}(c, x), \text{put } : c?!(y), \text{Var}(c, y) \} \]

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

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

\[
c \triangleleft \text{get } : c?(x).c \triangleleft \text{put } : c!(x+1) \mid \text{Var}(\tau, i) \quad (1)
\]

On channel \( c \) the get branch is selected (via \( \triangleleft \)), 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 \( c : +[\text{get }, \text{put } : [2]] \). This complements the session type of the variable agent and together their interaction is
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 $\pi$-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) 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 typingability from previous implementations. This is our Artificet, 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 $\lambda$-calculus with natural numbers, conditions, and recursion) which we call $\text{FPCF}$ (effectful PCF) with syntax:

$$
M, N ::= V | M \cdot N | \text{case } M | 0 \rightarrow N | (\text{let } x \rightarrow N | C)
$$

$$
V ::= x | \lambda x.M | \text{rec } (\lambda f, \lambda x.M) | C
$$

$V$ 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 $0, \text{succ, } unit \in C_v$. The case syntax pattern matches on natural number constructors.

We define a type-and-effect system for $\text{FPCF}$ similar in style to the rich effect systems of Nielson and Nielson [38], which differ from traditional effect systems (e.g., [22]) by distinguishing sequential control flow from branching control flow (alternation).

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

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

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

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

$$
\sigma, \tau ::= \sigma \cdot \tau \vee \tau | \text{unit}
$$

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

Figure 1 gives the type-and-effect rules. The (var) and (abs) rules describe variable use and abstraction as pure (with $I$). For (app), 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 $C_e$ with effects $C_e$. The effect of (case) sequentially composing the effect $F$ of the matched expression $M$ with the branching composition $G \cdot H$. The effect of a recursive binding (rec) is the same as that of its body and recursive call. The closure operator (‘$\sqsubseteq$’) is often required to provide a valid typing for a recursive definition.

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

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

$$
[] \oplus f = f \quad (e::f) \oplus g = e::(f \cdot g) \quad (f + g) + h = (f + h) + (g + h)
$$

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

$$
\emptyset \vdash \tau, \cdot | rd \tau \quad \emptyset \vdash \tau, \cdot | wr \tau, \cdot \text{unit, } \cdot
$$

For example, $\emptyset \vdash (\lambda x. \text{put} \cdot (\text{succ } x)) \cdot \text{get} \cdot \text{nat} \cdot [\text{rd nat, wr nat}]$ 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** ($\beta$-equivalence). Let $\equiv$ be an equivalence relation over terms with (call-by-value) $\beta$-equations:

$$
\begin{align*}
(\beta) & \quad (\lambda x. M) \cdot V \equiv M[V/x] \\
(\text{rec}) & \quad \text{rec } (\lambda f, \lambda x.M) \equiv \lambda x. M[\text{rec } (\lambda f, \lambda x.M) / f] \\
(\text{case}1) & \quad \text{case } 0 \rightarrow M \cdot (\text{succ } x) \rightarrow N \equiv M \\
(\text{case}2) & \quad \text{case } (\text{succ } V) \cdot 0 \rightarrow M \cdot (\text{succ } x) \rightarrow N \equiv N[V/x]
\end{align*}
$$

We add two further equations on $\text{let}$:

$$
\begin{align*}
(\text{let-id}) & \quad \text{let } x = M \in x \equiv M \\
(\text{let-associ}) & \quad \text{let } y = (\text{let } x = M \in N) \in N' \equiv \text{let } x = M \in \text{let } y = N \in N' \quad (\text{if } x \notin \text{fv}(N'))
\end{align*}
$$

\[\]
Note, (β) above is equivalent to \((\text{let } x = V \text{ in } M) \equiv M[V/x]\). 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, \(\eta\)-equality is type-dependent: \(M \equiv \lambda x.M.x\) holds only when \(M\) is a function type. Further, in FPCF \(\eta\)-equality holds only when \(M\) is also pure. That is, \(\Gamma \vdash M \equiv (\lambda x.M : x) : \sigma \rightarrow \tau, I\).

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

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

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

\[
\forall \Gamma \vdash M : \tau, F \land C[M] \rightarrow D[N] \Rightarrow \exists \Delta : \tau, G \land G \sqsubseteq F
\]

### 2.2 Session calculus

We consider the \(\pi\)-calculus with session primitives, which we call the session calculus. Figure 2 shows the syntax, where \(l\) ranges over labels, \(I : P\) 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 \(\bar{c}\) with \(\bar{c} = c\). Intuitively, names \(c\) and \(\bar{c}\) are two dual endpoints. Throughout we elide trailing occurrences of \(0\) and \(\text{end}\), e.g., writing \(\pi l(x)\) instead of \(\pi l(x)0\).

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

\[
(\beta) \quad c?l(x).P, c@l(V).Q \rightarrow c@l(V).Q \mid Q
\]

\[
(\beta\text{-chan}) \quad c?l(d).P, c@l(c\alpha).Q \rightarrow c@l(c\alpha).Q \mid P[c\alpha/d] \mid Q
\]

\[
(\beta+) \quad c\alpha l.P, c\alpha l.Q \rightarrow c\alpha l.d.P \mid P[c\alpha/d] \mid Q
\]

\[
(\beta\text{-if}) \quad c\alpha l.P \sqsupset Q \rightarrow l : \tilde{Q} \mid P, Q_1, l \in \tildeset
\]

**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 \(\text{replicated input over channel}\) \(c\), that is, \(c\) is not linear in the usual sense, but can repeatedly receive values via the persistence of \(c\alpha l.d\).P. Equation (β-if) 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 \textit{structural congruence} relation \(\equiv [57]\), which provides that \(\|\). It is a commutative monoid with \(0\) unit, amongst other things.

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

\[
S ::= l[l]\tau.S \mid ?\tau.S \mid \ast\tau.S \mid \text{end} \mid [l : \tilde{S}] \mid \&[l : \tilde{S}] \mid \mu\alpha.S \mid \alpha
\]

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

The output type \(\ast\tau\) differs from the send type \(!\tau\) since the former interacts only with replicated input: messages on a channel typed \(\ast\tau\) may occur in multiple processes. On the other hand, \(!\tau\) is a \textit{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 \(\text{dual}\) \(\bar{S}\) is defined:

\[
!\tau.S = \bar{!}\tau.S \quad \bar{?}\tau.S = !\tau.S \quad \bar{\ast}\tau = !\tau \quad \bar{\bar{\ast}\tau} = \ast\tau
\]

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

\[
\text{able}(\Delta) \equiv \forall (c : S) \in \Delta. \text{able}(S)
\]

\[
\text{able}(S) \equiv \exists T, S = \ast\tau.T \land \text{able}(T) \lor (S = \text{end})
\]

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 \(\Gamma\) is a mapping from variables to base types and \(\Delta\) 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 \(\ast\tau.d\).P replicates \(P\), it cannot contain free linear session names, guaranteed by \(\text{able}(\Delta)\) in (recv). Note that replicated outputs can be weakened (weak\(\ast\)). 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 \textit{balanced} session environments to be composed in parallel, defined by the following symmetric \(\bowtie\) relation:

\[
(c : S) \bowtie (\tilde{s} : \tilde{S}) \bowtie (c : !\tau.S) \bowtie (c : S) \bowtie (d : T)
\]

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 \((\ast)s\). 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, \(\mu\alpha.\ast\tau\) is dual to both \(\mu\alpha.\ast\tau\) and \(\ast\tau\), but (\(c : \mu\alpha.\ast\tau\)) and (\(c : \ast\tau\)) is undefined.

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

\[
\Delta, c : l[l] : \tilde{S}, \tau : \bar{[l] : [l]} \rightarrow \Delta, c : l[l] : \tilde{S}, \tau : \bar{[l] : [l]}, \\
\Delta, c : !\tau.S, \tau : ?\tau.T \rightarrow \Delta, c : !\tau.S, \tau : ?\tau.T
\]

\[
\Delta, c : \ast\tau.S, \tau : ?\tau \rightarrow \Delta, c : \ast\tau.S, \tau : ?\tau
\]

**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 \(\Delta\) is (self-)balanced, then \(\Delta'\) is balanced.
3. Effects as sessions: FPCF into session calculus

Our encoding is based on the encoding of the pure call-by-value \(\lambda\)-calculus into the \(\pi\)-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 (out-going). To see why this is needed, rather than just using the effect channel directly, consider a standard encoding for pure let-binding:

\[
\text{let } x = M \text{ in } N = \nu q. ([M]_q \mid ?! q(x). ![N]_q)
\]

The subscript on encodings \([-] \) 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 \(-\rightarrow \_ \) for an incoming effect channel carrier \(\text{ei} \) and outgoing carrier \(\text{eo} \). The let-binding encoding is then:

\[
\text{let } x = M \text{ in } N = \nu q. \text{ea}([M]_{\text{ei}} \mid ?! q(x). ![N]_{\text{eo}})
\]

where \(\text{ea} \) is an intermediate carrier between \(M \) and \(N \). This approach resembles a continuation-passing style semantics or “threading” a store in an operational semantics, \(c, s \rightarrow (c', s') \). 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 instantiatted to embed different notions of effect.

**Definition 8.** *Effect-encoding parameters comprise:*

1. an *effect handler* process \(H(eff) \) parameterised by \(eff \), typed:
   
   \[
   \text{for some } S \quad \emptyset; eff : \mu c.S \ni H(eff)
   \]

2. a *terminator* process \(T(eff) \) such that \(T(eff) \mid H(eff) \rightarrow^* \emptyset \)

3. an *interpretation functor* \([-] : \mathcal{F} \rightarrow \mathcal{S} \) from effect algebras to session types, satisfying the following homomorphism property (discussed more in Section 3.4) where \(\bullet \) is a (partial) sequential composition for session types:

\[
[I] = \text{end} \quad [F \cdot G] = [F] \bullet [G]
\]

4. an encoding \(\langle - \rangle_{\text{eff}}^{\text{eo}} \) for all constants \(C \) satisfying that, \(\forall P, \emptyset ; \emptyset C, C_{\text{eff}}^{\text{eo}} = r ![C_{\text{eff}}], \text{ei} ; ![C_{\text{eff}} \bullet g], \text{eo} ; ![g] \ni P \)

5. an encoding for *case/\text{if} \) (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 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. \([\_ \] \) instead of \([\_] \).

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

\[
\Gamma \ni M : \tau, F, [\_]_{\text{eff}} = \Gamma^{\text{eo}} : \Gamma^{\text{eo}}, \text{r} : ![\_ \text{eff}], \text{ei} ; ![\_ \text{eff}], \text{eo} ; ![\_ \text{eff}].T(\text{eff})(H(\text{eff}))
\]

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

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

\[
[\sigma \langle F \rangle_{\text{eff}}^{\text{eo}}] = ![\sigma \text{eff}], ![F \bullet g], ![g]_r
\]

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 \([\sigma \text{eff}] \) value for the argument, (2) an effect carrier to receive an effect channel capable of simulating effects \(g \) and (3) a channel to send a \([\sigma \text{eff}] \) value for the result.

Free-variable contexts of FPCF are interpreted into a value-variable context (for ground types), written \([-\_]_{\text{eo}} \), and a session-variable context for functions, written \([-\_] \) as:

\[
\begin{align*}
[\emptyset] &= \emptyset \\
[\Gamma, x : T] &= \Gamma^{\text{eo}}, x : ![T] \\
[\emptyset] &= \emptyset \\
[\Gamma, x : \sigma \langle F \rangle_{\text{eff}}^{\text{eo}}] &= \Gamma^{\text{eo}}, x : ![\sigma \text{eff}]_r
\end{align*}
\]

**Terms/derivations** The core embedding is the intermediate \(\langle - \rangle_{\text{eff}}^{\text{eo}} \) from FPCF derivations to session-calculus derivations, of the form:

\[
\langle \Gamma \ni M : \tau, F \rangle_{\text{eff}}^{\text{eo}} = \forall P, \emptyset ; \emptyset C, C_{\text{eff}}^{\text{eo}} = r ![C_{\text{eff}}], \text{ei} ; ![C_{\text{eff}} \bullet g], \text{eo} ; ![g]_r \ni P
\]

\[
\langle \Gamma \ni M : \tau, F \rangle_{\text{eff}}^{\text{eo}}
\]

\[
\langle \Gamma \ni M : \tau, F \rangle_{\text{eff}}^{\text{eo}}
\]
where $P$ is the encoded term. The incoming carrier $e_i$ receives an effect channel of type $\llbracket F \bullet g \rrbracket$ (capable of carrying out effect interactions $F \bullet g$) and $e_0$ sends an effect channel of type $\llbracket g \rrbracket$ (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:

$$
\begin{align*}
\text{ei} & : ?[F \bullet g], e_0 : ?[g] \vdash \langle M \rangle^{e_i, e_0} \\
\text{ea} & : ?[G \bullet h], e_0 : ?[h] \vdash \langle N \rangle^{e_0, e_0}
\end{align*}
$$

where $\llbracket F \bullet g \rrbracket$, $\llbracket g \rrbracket$ and $\llbracket h \rrbracket$ are universally quantified at the meta level.

We show first the encoding of the $\lambda$-calculus subset of FPCF. We elide types where possible for brevity.

### 3.1 $\lambda$-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,

$$
\langle x \rangle^{e_i, e_0} = e_i(r, c, \langle e_i \rangle^{c, c})
$$

The typing of the effect carrier channels is $V_{g,e_i} : ?[g], e_0 : ?[g]$ revealing the pure nature of variables.

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

**Abstraction** encoding comprises a function and argument: the left-hand side (function) encoding uses fresh channels $q$ and $e_0$ 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 $e_0$ and uses fresh channels $s$ and $eb$ to send the result and outgoing effect channel respectively.

$$
\langle \lambda x. M \rangle^{e_i, e_0} = \\
v.d. (e_i(r, c), r! (d), \langle \text{end} \rangle^{e_i, e_0}, s.d(?)(p, e_a, e_b, q), p? (x), \langle M \rangle^{e_0, e_0})
$$

The new channel endpoint $d$ is sent on the result channel $r$, then the opposite endpoint $\text{end}$ receives four channels needed for $M$, the function body: $p$ receives the argument $x$, $e_a$ receives the incoming effect channel, $e_b$ 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 $e_0$ 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 $e_0$ and uses fresh channels $s$ and $eb$ to send the result and outgoing effect channel respectively.

$$
\langle M N \rangle^{e_i, e_0} = \\
v.d. (e_i(r, c), r! (d), \langle \text{end} \rangle^{e_i, e_0}, s.d(?)(p, e_a, e_b, q), p? (x), \langle M \rangle^{e_0, e_0})
$$

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 $e_0$ 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 $\lambda$-abstraction encoding, since suc is a function:

$$
\langle 0 \rangle_{e_i, e_0} = e_i(r, d, \langle c \rangle^{e_0, e_0})
$$

$$
\langle \text{suc} \rangle_{e_i, e_0} = v.d. (e_i(r, c), r! (d), \langle \text{end} \rangle^{e_i, e_0}, s.d(?)(p, e_a, e_b, q), p? (x), (\text{suc} \langle \text{suc} \rangle) (\text{end}^{e_0, e_0}))
$$

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

### 3.2 Control flow: conditionals and fixed-points

The above encodes the $\lambda$-calculus subset of FPCF, giving the sequential composition of effects. We now encode the control-flow operators which PCF adds to the $\lambda$-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., $\oplus = \sqcup$ in our calculus). Consider these two approaches are equivalent since subeffecting can be used to get the same (upper bound) effect for each branch of a case. The following two alternate type-and-effect rule:

$$
\begin{align*}
\Gamma & \vdash M : \text{nat}, F \quad \Gamma \vdash N_1 : \tau, G \quad \Gamma, x : \text{nat} \vdash N_2 : \tau, G \\
\Gamma & \vdash \text{case } M \text{ of } 0 \to N_1, (\text{suc } x) \to N_2 : \tau, G
\end{align*}
$$

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 $G_1 \sqcup G_2 \sqcup G = G_1 \sqcup G_2$. Subtyping 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:

$$
\begin{align*}
\{ x \in \text{nat}. \text{succ}_G \mid x = 0 \} & \vdash (\text{case } M \text{ of } 0 \to N_1, (\text{suc } x) \to N_2) : \tau, G \\
\{ x \in \text{nat}. \text{succ}_G \mid x \neq 0 \} & \vdash (\text{pred}_G (x) \langle \text{suc } M \rangle) : \tau, G
\end{align*}
$$

where $\text{pred}_G (x)$.

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:

$$
\begin{align*}
\langle \text{rec } (\lambda x. M) \rangle_{e_i, e_0} = \\
v.d. (e_i(r, c), r! (d), \langle \text{end} \rangle^{e_i, e_0}, s.d(?)(p, e_a, e_b, q), p? (x), \langle M \rangle^{e_0, e_0, d/f})
\end{align*}
$$

Key: same type restriction and 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 $\langle f \rangle^{e_i, e_0}_{e_d, d/f} = e_i(c), s.d(d) \langle \text{end} \rangle^{e_i, e_0}$. Thus any applications of $f$ recursively use the encoding of the function body (over the dual $d$). 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 $\sqcup$: 

Definition 10 (Free upper bound). For an effect system over \( \mathcal{F} \), extend the carrier set to \( \mathcal{F}_+ = \mathcal{F} \cup \{ F + G | F, G \in \mathcal{F}_+ \} \) with \(+\) a term constructor with no equations; it is a free constructor. Extend subeffecting such that \( F \subseteq F + G \) and \( G \subseteq F + G \) 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:

\[
[F + G] = [alt: ([L : [F]], R : [G])] \quad (8)
\]

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

\[
\langle \Gamma \vdash M : \tau, F \rangle^{\ast\ast} = \nu a. \langle e_i(c).c < \tau.c \in L.F(c) | \langle \Gamma \vdash M : \tau, F \rangle^{\ast\ast} \rangle
\]

On the effect channel \( c \), received on \( e_i \), 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 \( e_i \) via \( \mathcal{E}_R \) :

\[
e_i : ([alt: ([L : [F]], R : [G]])\
\]

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 \( G \subseteq F + G \) 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 \( +\) :

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

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

Remark 1. Setting \( \ast \ast = + \) gives a free representation of alternation in effects (computation trees). For the general encoding, restricted cease (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 \( [\cdot] : \mathcal{F} \rightarrow S \) is required to satisfy the following homomorphism property, which ensures the continuation-passing style approach is well typed:

\[
[I] = \text{end} \quad [F + G] = [F] \bullet [G]
\]

where \( \bullet \) is sequential session-type composition, defined:

\[
T \bullet end = T \quad \text{end} \bullet T = T
\]

\[
\begin{align*}
[!]T & = [!]T, (S \cdot T) & \Rightarrow [!]T, (S \cdot T) = [!]T, (S \cdot T) \\
\oplus & [I : S] \bullet T = \oplus [I : (S \cdot T)] & \& [I : S] \bullet T = \& [I : (S \cdot T)] \\
\ast & [I : S] \bullet T = \ast [I : S] (S \cdot T) & (\mu \alpha. S) \bullet (\mu \beta.T) = \mu \gamma. (S[\gamma/\alpha] \cup T[\gamma/\beta]) \quad (\gamma \text{ fresh})
\end{align*}
\]

where \( S \cdot T \) is the vector-scalar lifting of \( \bullet \). All other cases are undefined, e.g., \( \ast [\tau]T = \bot \). Arguments of the composition are taken up-to equivalence of session types (which is decidable).

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

\[
(\mu \alpha. [I : \alpha]) \bullet (\mu \beta. [I_2 : [\tau, \beta]] \Rightarrow (\mu \gamma, \gamma \in [I : [\tau, \gamma]])
\]

3.5 Examples

In our examples, we define effect handlers as recursive processes via replicated input, of the form \( v h_\cdot (\ast h(c, x), P | \text{R}(c, V)) \) for some effect channel \( c \) and value arguments \( V \). The process \( P \) tends to contain an output on \( \text{R} \) 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 \( \tau \) (mono-typed stores) for simplicity. We instantiate our encoding to embed this into the session calculus. Effect annotations are interpreted as:

\[
\begin{align*}
([r d \tau] : F) & = \oplus [r d : [\tau], [F]] & \text{end} \quad [w r \tau] : F & = \oplus [w r : [\tau], [F]]
\end{align*}
\]

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(\text{eff}) = \text{eff} : \mu \alpha. \& [r d : [\tau], \gamma. \& [w r : [\tau], \gamma.]

\[
\begin{align*}
\begin{array}{c}

\nu h.(\ast h(c, x), c \Rightarrow \& [r d : \text{get}(c, x), c \Rightarrow \& [w r : \text{get}(c, x)] \Rightarrow \text{stop} 0) | h(\text{eff}, 0)
\end{array}
\end{align*}
\]

with terminator \( H(\text{eff}) \) = \text{eff} \ast \text{stop}. State operations are encoded:

\[
\begin{align*}
\langle get \rangle = \oplus [\text{get}(c, x), r d : \text{get}(c, x), \gamma. \& [r d : \text{get}(c, x), \text{stop} : 0]
\end{align*}
\]

The decoding 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 \( \tau \) before sending \( c \) on \( \mathcal{E}_R \). 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 \( F = P\{[r d \tau, w r \tau] | \forall \tau \}, \) with \( \langle F, \cup, \emptyset \rangle \) for sequencing and \( \oplus = \cup, \bigwedge F = F \). 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 \( [\cdot] \) which is instead given in terms of a recursive type over a selection:

\[
[F] = \mu \alpha.S \text{ where } \forall \langle \tau c, \alpha \rangle \in F \Rightarrow \oplus [r d : [\tau], \emptyset < S) \quad (9)
\]

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,

\[
\begin{align*}
\langle [r d \tau, w r \tau] : F, \emptyset \rangle & = \mu \alpha. \oplus [r d : [\tau], \alpha. \emptyset \oplus [w r : [\tau], \emptyset. [S, \emptyset]]
\end{align*}
\]

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 Cicek et al. [12] and Danielsoni, via the \text{thunk} annotated monad) [13]. The sequential part of this effect system is given by the ordered monoid \( (\mathbb{N}, +, 0, \leq) \) with \( \oplus = \max. \) For recursion, the domain is extended to \( \mathbb{N} \cup \{w.t\} \) such that \( n^\ast = r \) which is greatest element \( \emptyset \subseteq \mathbb{N} \) and absorbing \( w \cdot r = \). Thus, \( FPCF \) can be instanci ated for this effect system with the additional constant \( \emptyset \vdash \text{tick} : 1. \)
Resource counting effects are embedded via the following handler, terminator, and operation encoding:

\[ H(c) = \nu h. (\mu e. (c). e.C(\mathcal{D})) \& \{ \text{tick} \mapsto d \} \nu l. (c) \& \text{stop} : 0 \} \nu l. (c) \]

\[ T(c) = \nu d. (c). d \& \text{stop} \]

\[ \text{tick} \mapsto_r e \vdash \nu l. (c). e.C(\mathcal{D}) \& \{ \text{tick} \mapsto d \} \]

That is, a “tick” is encoded by a channel \( d \) which is selected the tick behaviour of the handler and a unit value is then sent. The encoding of effect annotations \([-] : \mathbb{N} \rightarrow \mathcal{S}\) is then essentially a base 1-encoding of naturals:

\[ [0] = \text{end} \quad [n + 1] = \#(![\text{tick} \mapsto \text{[unit]}], n) \]

\[ [\omega] = \mu \alpha. ([\text{tick} \mapsto \text{[unit]}], \alpha) \]

The handler has type \( c : \#(![\text{tick} \mapsto \text{[unit]}]) \). Type soundness of this encoding requires a small extension to define type duality for repeated output.

**Definition 11** (Repeated output). We write \( ![\tau]^n \) for a finite sequence of outputs: \( ![\tau] = ![\tau] \).

We extend the duality function to a symmetric relation dual with:

\[ \forall n. \text{dual}([\tau]^n, ![\tau]) \quad (10) \]

The limit of this is a fixed point, i.e. if \( \lim_{m \to \infty} eq. (10) = \mu \alpha. ![\tau], \sigma \]

giving \( \text{dual}([\tau]^n, \sigma, ![\tau]) \) which provides the duality of \([\omega]\) with the handler. Subtyping is then extended with:

\[ ![\tau]^n \prec ![\tau]^m \quad (\text{where } m \leq n) \quad (11) \]

Subfactoring by \( \prec \) follows from equation (11). That is, the number of outputs (dual to replicated input) can be upper approximated.

### 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:

\[ \forall g, \#([\Gamma])^n, \#([\Gamma]), \#(F \circ g), \#([g]), r : ![\tau]^n \vdash \{ \Gamma \vdash M : \tau, F \}^{\#(g)} \]

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

\[ \#([\Gamma])^n : \#([\Gamma]), r : ![\tau]^n \vdash \{ \Gamma \vdash M : \tau, F \} \]

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

**Definition 12** (Barbs). 1. (untyped barb) \( P \prec \#(c) \) if \( \exists V, P_2, P_3 \) s.t. \( P \equiv \text{vkin}(c(V), P_2 \mid P_3) \) where \( c \notin \text{in} \)

2. (barb) \( \Gamma \vdash P \prec \#(c) \) if \( P \prec \#(\text{in}) \) and \( \tau \notin \text{dom}(\Delta) \)

3. (weak barb) \( \Gamma \vdash P \prec \#(c) \), if \( P \vdash Q \wedge P \vdash Q \prec \#(c) \) 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 \( \tau \) is not free (else the send would be unobservable).

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

- if \( \Gamma \vdash \Delta_1 \vdash P \Re \Delta_2 \vdash Q \), then \( \Delta_2 \vdash Q \Re \Delta_1 \)
- \( \forall P \vdash P' \) then \( \exists Q', \Delta_1', \Delta_2' \) such that \( Q \mapsto * \) and \( \Gamma \vdash \Delta_1' \vdash P' \Re \Delta_2' \vdash Q' \)
- \( \forall C. \exists \Delta_1', \Delta_2' \) such that \( \Gamma \vdash \Delta_1' \vdash C[P] \Re \Delta_2' \vdash C[Q] \)
- \( R \) 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 \( \equiv \) for the largest reduction-closed barred 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 calculus.

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

\[ \exists P, d, y. \{ V \}^{\#(y)} \equiv \nu d. (e)(c). r(y). \#(c) \]

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).

\[ \nu v, e. (\#(x)). \#(y) \equiv \nu v, e. \#(x). \#(y) \]

**Lemma 3** (Forwarding). For process \( P \) and channel \( e \in \#(F) \)

\[ \nu e. (\#(c)). \#(b) \equiv \nu e. \#(c). \#(b) \]

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). \( \forall M, F, T, \Gamma, \)

- (sound) \( \forall N, \Gamma \vdash M : \tau, F \wedge (M \rightarrow N) \wedge F \subseteq G \Rightarrow \exists P. (\Gamma \vdash M : \tau, F \rightarrow P) \wedge P \equiv \Gamma \vdash M : \tau, G \wedge F \subseteq G \)
- (complete) \( \forall P. (\Gamma \vdash M : \tau, F) \wedge (\Gamma \vdash M : \tau, F \rightarrow P) \Rightarrow \exists N, G. (M \rightarrow N) \wedge P \equiv \Gamma \vdash M : \tau, G \wedge F \subseteq G \)

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

\[ \Gamma \vdash \#(x) = \#(N) : \tau, F \circ g \Rightarrow \Gamma \vdash \#(x) : \tau, F \circ g \]

The main theorem of this section is the observational congruence of the encoding with respect to \( \beta \eta \)-equality of FPCF.

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

\[ \Gamma \vdash M \equiv \#(N) : \tau, F \Rightarrow \Gamma \vdash M \equiv \#(N) : \tau, F \]

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, Corollary 5.2]):

\[ \Gamma \vdash M : \tau, F \Rightarrow \Gamma \vdash M : \tau, F \]

Contextual completeness is a consequence of Theorem 1 together with computational adequacy, cf. [56, Corollary 5.2]. Proving soundness with respect to contextual equivalence \( \equiv \) 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 \( M || N \). Unrestricted effects in parallel branches may cause race conditions, e.g. put (get + 1) || 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 \((F, \circ)\) representing parallel computation where \(I\) (purity) is the unit of \(\circ\). Typing and equality is extended:

\[
\Gamma \vdash M : \text{unit}, F \quad \Gamma \vdash N : \text{unit}, G
\]

\[
\Gamma \vdash M \parallel N : F \parallel G
\]

\[
M \parallel \equiv M \quad M \parallel N \equiv N \parallel M \quad (M \parallel N) \parallel P \equiv (M \parallel N) \parallel P
\]

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 unit \(\parallel\) unit \(\to\) 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:

\[
\nu_e.(\nu h?((c, x), e!?(d), d) > \{ rd : d!(x), \nu l?(c, x), wr : d?!(y), \nu l?(c, y), stop : 0\}) \mid \nu l?(\text{eff}, 0)
\]

This handler has the following recursive session type for \(\varepsilon\):

\[
\varepsilon : \mu \alpha . ?[S] \alpha \quad \text{where} \quad S = \| \nu r : ?[\tau], \nu r : ?[\tau], \text{stop : end}
\]

i.e., \(\varepsilon\) 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:

\[
\langle \text{get} \rangle_{\nu e} = \nu r?(!c, \nu r.e!(\pi, r \cdot d), e \cdot rd, e!?(x), r!(x), \nu r=(\pi)\}
\]

The associated effect annotation interpretation is then as follows, where \(\nu r=(\pi)\) in the above definition (sending a fresh channel to the handler) is typed with a repitition type:

\[
\| [] \| = \text{end} \quad \langle [(rd \tau) \cdot F] \cdot \| [\text{wr} \tau \cdot \| \rangle \rangle = \| ![\nu r : ?[\tau], \| [\nu r : ?[\tau], \text{stop : end}]\}
\]

The parallel composition operator is then defined:

\[
\| \nu e, (\nu e, (\nu r, \nu r, \nu r)\} \langle [(rd \tau) \cdot F] \cdot \| [\text{wr} \tau \cdot \| \rangle \rangle = \| ![\nu r : ?[\tau], \| [\nu r : ?[\tau], \text{stop : end}]\}
\]

That is, the effect channel \(c\) is received on \(e\), 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 channels 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): \(\nu e, (\nu e, (\nu r, \nu r, \nu r)\} \rangle\). This requires a single, unifying session type for \(c\) in each branch. Let \(M, N\) have for \(c\) the session types \(\nu e, (\nu e, (\nu r, \nu r)\} \langle [(rd \tau) \cdot F] \cdot \| [\text{wr} \tau \cdot \| \rangle \rangle = \| ![\nu r : ?[\tau], \| [\nu r : ?[\tau], \text{stop : end}]\}

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

\[
(F \circ G) = \| ![\nu r : ?[\tau], \| [\nu r : ?[\tau], \text{stop : end}]\}
\]

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 \(\forall n.duall!{\nu e, (\nu e, (\nu r, \nu r)\}}\).

**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 \and (M \to N) \and F \in G \Rightarrow \exists P. (\Gamma \vdash M : \tau, F, \nu r \rightarrow P) \and P \equiv \Gamma \vdash N : \tau, G, \nu r \rightarrow P)

- (complete) \(\forall P, \Gamma \vdash M : \tau, F, \nu r \rightarrow P) \and (\Gamma \vdash M : \tau, F, \nu r \rightarrow P) \Rightarrow \exists N, G. (M \to N) \and P \equiv \Gamma \vdash N : \tau, G, \nu r \rightarrow F \and F \in G

**Theorem 4** (Soundness w.r.t. extended \(\beta\eta\)-equality (13)). Lemma 2 (substitution) extends to the parallel encoding. Then:

\[
\Gamma \vdash M \equiv N : \tau, F \Rightarrow (\Gamma \vdash M : \tau, F, \nu r \rightarrow P) \equiv (\Gamma \vdash N : \tau, F, \nu r \rightarrow P)
\]

For this extended encoding, equational completeness (w.r.t observational equivalence (12)) does not hold. This is because any process \(P \equiv \Gamma \vdash N : \tau, F, \nu r \rightarrow P\) 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 \(\pi_S\) here. The core subset of \(\pi_S\) (send, receive, delegation, branch, select, parallelism, and restriction) is contained by the subset providing replicated input and output, denoted \(\ast\) 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).

<table>
<thead>
<tr>
<th>FPCF \ \pi_S</th>
<th>core</th>
<th>\ast</th>
<th>if</th>
<th>:&lt;:</th>
<th>eq. 10</th>
<th>\forall</th>
</tr>
</thead>
<tbody>
<tr>
<td>\ast</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>\ast</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>\ast</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>\ast</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

The \(\ast\) line represents the restricted version of case which has \(F + F = F\) and \(F + G = \bot\) when \(F \neq G\). The \(\ast\) line represents \(\pi_S\) constructed via subeffecting with the free alternation operator \(+\) (Section 3.3). The \(\ast\) 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 \([3, 46]\) which 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., \(\| \text{let } x = \text{choose } V_1 \text{ } V_2 \text{ in } M \|\). 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 \(+\), 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 \( \otimes \) and branch & with \( \tau ::= \text{nat} \mid \text{unit} \mid \{S \} \) and \( S, T ::= |[\tau]|S \mid ?[\tau]|S \mid ![\tau]|S \mid \text{end} \mid S + S \mid \mu \alpha.S \mid \alpha \mid \tilde{\alpha}S \) where \( \tilde{\alpha}S \) is an intermediate representation for the session type of channels being composed via \( \tilde{\otimes} \) (balanced composition). We assume equimorphic equations on S, e.g., \( \mu \alpha.S = \{S[\mu \alpha.S/\alpha]\} \).

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

- \( (\cdot, \bullet, \emptyset) \) where \( I = \emptyset \) is the empty mapping, \( \bullet \) is pointwise sequential composition of session types (via \( \cdot \), Section 3.4) where \( \forall c : S \in F \land c : T \in G \) then \( c : S \otimes T \in (F \otimes G) \) but with additional equations for balancing composition triggered by \( \tilde{\otimes} \) types e.g. \( \Delta \bullet (c \tilde{\otimes} S) = \Delta \) if \( \tau : \tilde{\alpha} \notin \Delta \) amongst other cases (the technical report [41] gives the full definition).
- \( \otimes \) is set union \( \cup \) if \( \Delta_1 \otimes \Delta_2 \) satisfies the following condition:
  \[ \forall c : S \in \Delta_1 \Rightarrow c : S \in \Delta_2 \land c \notin \text{dom}(\Delta_2) \]
  i.e., a channel is either in \( \Delta_1 \) or \( \Delta_2 \), or in both but with the same session type, otherwise \( \Delta_1 \otimes \Delta_2 = \Delta_1 \).

\( \otimes \) is a preorder where \( \forall \Delta, c. c \notin \text{dom}(\Delta) \):

\[ \begin{align*}
\Delta \otimes (\Delta, c : \text{end}) & \subseteq (\Delta, c : S) \otimes (\Delta, c : S + T)
\end{align*} \]

\( \otimes \) is defined \( \forall c : S \in F \) then \( c : \mu \alpha.(\emptyset \tilde{\otimes} S \otimes \alpha) = F^\ast \).

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

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

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:

\[
\begin{align*}
send_{c,x} & : Ch c \rightarrow \tau \{c : \![\tau]\}\rightarrow \text{unit} \\
send_{c,d,s} & : Ch c \rightarrow Ch d \{c : \![s]\}, d : \tilde{\alpha}s \rightarrow \text{unit} \\
schSend_{c,d,s} & : Ch c \rightarrow Ch d \{c : \![s]\}, d : \tilde{\alpha}s \rightarrow \text{unit} \\
recv_{c,r} & : Ch c \rightarrow \tau \{c : ?[\tau]\} \\
chRecv_{c,d,s,r} & : Ch c \rightarrow (Ch d \{c : ?[s]\}) \rightarrow \tau \rightarrow \tau \\
new_{c,d,s,r} & : Ch c \rightarrow Ch \tau \{c : ?[s]\}, d : \tilde{\alpha}s \rightarrow \tau \rightarrow \tau \\
\end{align*}
\]

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 \( \otimes \) 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 \( \tau \) with the effect F \( \{d : s\} \).

For send, rsend, and chSend, the value or channel is added to the end of the queue belonging to the opposite end-point \( \overline{r} \). For recv, a value is dequeued from the end-point c queue (if it exists in s); chRecv is a little different, receiving first a channel 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, \( v.C \) introduces channel names c and \( \tau \) in the scope of C. In FPCF, new encodes restriction but new \( (\lambda x.M) \) binds arbitrarily named variables x, y in the scope of \( M \), which are substituted for concrete channel names. The (new) rule below deals with this by first \( \alpha \)-renaming the variables of a new redex to fresh variable names corresponding to fresh channels:

\[
\begin{align*}
\text{(new1)} & \quad \text{where } x, y, c, \overline{c} \notin \text{dom}(s), c, \overline{c} \text{ fresh in } M \\
\text{(new2)} & \quad (\text{new}(\lambda x.\lambda y.M), s) \rightarrow (\text{new}(\lambda x.\lambda y.M[c/x, \overline{c}/y]), s[c\rightarrow e, \overline{c} \rightarrow e])
\end{align*}
\]

That is, if \( x, y \) are not in the store, then \( x, y \) are \( \alpha \)-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:

\[
\begin{align*}
\text{(new3)} & \quad (\text{new}(\lambda x.\lambda y.V), s[c\rightarrow e, \overline{c} \rightarrow e]) \rightarrow (V, s[c\rightarrow e, \overline{c} \rightarrow e])
\end{align*}
\]

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 \( \eta \)-expansion for new over pure values: new \( (\lambda x.\lambda y.V) \equiv V \) and distributivity \( P \parallel \text{new}(\lambda x.\lambda y.Q) \equiv \text{new}(\lambda x.\lambda y.P \parallel Q) \) if \( x, y \notin fr(P) \).
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
\[
\exists M. \quad \Gamma; \Delta \vdash P \Rightarrow [\Gamma] \vdash M : \text{unit}, [\Delta]
\]

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

Definition 15 (Session types to effect sessions annotations).

[\mu \alpha \cdot S = \mu \alpha. S \mid [[\mu \alpha \cdot S]] = \mu \alpha. [[S]] \mid \exists d. [l_1 : S_1, l_2 : T] = !\text{nat}. ([S] + [T]) \mid \exists c. [l_1 : S_1, l_2 : T] = ?\text{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 $\text{nat}$ to select a branch prior to alternation. For $\&$, this corresponds to receiving a $\text{nat}$ 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 $\Gamma; \Delta \vdash P$.

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

\[
\begin{align*}
[|c|.(V).P] & = \text{send } c \ [V] ; [P] \\
[|c?|.(x).P] & = \text{let } x = \text{recv } c \ \text{in} \ [P] \\
[|c?|.(d).P] & = \text{let } k = \text{recv } c \ \text{in} \ k (\lambda d. [P]) \\
[|\Delta \triangleright c|.(d).P] & = \text{chSend } c \ d ; [P] \quad \text{(if } c : \exists ! S, T \in \Delta) \\
\end{align*}
\]

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

\[
\begin{align*}
[\Delta \triangleright c|(d).P] & = \text{rsend } c \ d ; [P] \\
[|\mu c?|.(c?).P] & = \text{recv } (\lambda x. \text{let } k = k (\lambda d. [P]) \ \text{in} \ (k (\lambda d. [P])) \ | f \ | \text{unit}) \\
\end{align*}
\]

For output, $\text{rsend}$ 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:

\[
[P \ | | Q] = [P] \ | | [Q] \quad [[\mu \alpha . P]] = \text{new}(\alpha . \lambda x . [P]) \quad [0] = \text{unit}
\]

Parallel composition is encoded by the parallel extension of FPCF (Section 5) and restriction by new, binding names $\alpha$ and $\tau$.

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 $\text{nat}$).

\[
\begin{align*}
[|c < l_1|.(P)] & = \text{send } c \ 0 ; [P] \\
[|c < l_2|.(P)] & = \text{send } c \ 1 ; [P] \\
[|c > l_1|.(l_1 : P, l_2 : Q)] & = \text{let } x = \text{recv } c \ \text{in} \ (\text{case } x \text{ of } 0 \Rightarrow [P], \text{succ } n \Rightarrow [Q]) \\
[\text{if } [V = 0] \ \text{then } P \ \text{else } Q] & = \text{case } V \text{ of } 0 \Rightarrow [P], \text{succ } n \Rightarrow [Q]
\end{align*}
\]

To select label $l_1$, the encoding sends 0 then continues as $[P]$ with 1 sent instead for label $l_2$. Sub effecting (Def. 14) provides the effect $c : [\text{nat}] 

\text{unit}(S + T)$ for the $l_1$ select (where $c : S$ in the effect of $[P]$ and $c :: [\text{nat}](T + S)$ in the $l_2$ select. Since the encoding is type-directed, sub effecting rules are generated to match the session types. Branching has the dual behaviour of type $c : [\text{nat}] (S + T)$ 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 sub effecting).

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

Value encoding is fairly direct as PCF has the same constructors as the session calculus (modulo pred): $[\text{unit}] = \text{unit}, [\text{Suc } V] = \text{Suc } [V]$. $[0] = 0$, $[\text{pred } V] = \text{case } [V] \text{ of } 0 \Rightarrow 0, (\text{succ } x) \Rightarrow x$, 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 $\beta$-reductions (synchronisation) in the session calculus. This requires the notion of top-level contexts, in which the terms of a (session) $\beta$-redex from our encoding reside:

Definition 16 (Top-level contexts).

\[
\text{T} := \text{let } x = T \ \text{in} \ M \ | \ \text{new}(\lambda x. \lambda y. T) \ | \ T \ | \ T \ | \ M \ | \ M \ | \ T \ | \ [-] \\
\]

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

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

\[
\begin{align*}
(\beta \Rightarrow) & \quad [T \text{send } c \ V] \ | | \text{recv } c, s \Rightarrow [T \text{unit}] [V], s \\
(\text{ch} \Rightarrow) & \quad [T \text{chSend } c \ d] \ | | \text{recv } c, s \Rightarrow [T \text{unit}] \ | | [\lambda k . k \ d], s \\
(\text{new} \Rightarrow) & \quad [M/c, \lambda x. \lambda y. T, s] \ | \ \text{let } c \Rightarrow e \Rightarrow e \Rightarrow [\lambda x/c, \lambda y/\beta, T, s] \\
(\Rightarrow) & \quad [\text{new}(\lambda x/c, \lambda y/\beta, T, s)] \ | | M \Rightarrow N \\
\end{align*}
\]

where $\beta$ is the union of two finite maps with a left-bias, $c \Rightarrow t$ $\beta$ $c \Rightarrow s$ if $c \Rightarrow t$ but $\beta$ $c \Rightarrow s$ if $c \Rightarrow s$. In the premise of $\Rightarrow$, only the pure $\beta$-reductions (from Section 2.1) are allowed. A similar rule to (\beta \Rightarrow) is provided where $\text{rsend}$ replaces $\text{chSend}$.

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

Lemma 6. Let $\Gamma; \Delta \vdash P$ where $\Delta$ is closed and balanced, and $\text{wf}(\Delta, s)$, then $((P)\theta, s) \Rightarrow (N, s')$ $\Rightarrow ((P)\theta, s) \Rightarrow (N, s')$ where $\theta$ is a substitution of free channel variables for channel values of the corresponding name in the store $s$.

Further, $((P)\theta, s) \Rightarrow (N, s') \Rightarrow (\exists M, s').((P)\theta, s) \Rightarrow (M, s') \wedge (N, s') \Rightarrow ((M, s') \wedge (N, s'))$ Thus, under the image of our encoding and session typing, stable reductions (synchrony) and the operational semantics coincide.

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

- (sound) $\forall Q. \quad \Gamma; \Delta \vdash P \ | \ Q \Rightarrow Q \ \wedge \text{wf}(\Delta, s)$
- (complete) $\forall s'. \quad \Gamma; \Delta \vdash P \ | \ (P)\theta, s \Rightarrow (M, s') \wedge M \subseteq [Q]$

For soundness, session calculus $\beta$-reductions correspond to $\Rightarrow$ (and $\text{ch} \Rightarrow$), combined with $(\Rightarrow)$ 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 $(\Rightarrow)$. Completeness similarly relates stable and session calculus reductions.
The (bind) rule models the let-binding of PFCF, propagating effect information in the same way. ByWatler and Thiemm’s translation, an FPCF judgment \( \Gamma \vdash M : \tau,F \) maps to a monadic metalanguage judgment \( \Gamma \vdash [M] : \text{m}[F] \tau \), embedding effects into types. We use this graded monadic encoding 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.

### 8. Implementation

We use the encoding of the session calculus into PFCF to derive a new implementation of session types in Haskell. The gap between PFCF 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 web-page http://dorvard.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, Walder and Thiemm showed [34] that an impure \( \lambda \)-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, PFCF 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 -> * -> *` where `ef` models a set of effect annotations, to a unary type constructor where `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` implementing `*` of the effect algebra for `f` and a constant `Unit` 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:

\[
\begin{align*}
\Gamma \vdash e : m \mathcal{F} \sigma & \quad \Gamma, x : \sigma \vdash e_2 : m \mathcal{G} \tau \\
\Gamma \vdash (\text{do } x \leftarrow e_1 ; e_2) : \text{m} (\text{Plus} \ m \mathcal{F} \mathcal{G}) \tau
\end{align*}
\]

The (bind) rule models the let-binding of PFCF, propagating effect information in the same way. By Walder and Thiemann’s translation, an FPCF judgment \( \Gamma \vdash M : \tau,F \) maps to a monadic metalanguage judgment \( \Gamma \vdash [M] : \text{m}[F] \tau \), embedding effects into types. We use this graded monadic encoding 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 (a :: [Map Name Session]) a = Proc (ID a)
```

The first parameter `a` is a type-level finite map modelling an environment of session type information. This is of the form `\([c :-> s, d :-> t, \ldots]\)` 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 `*` 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 PFCF 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] ()
```

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:

```
neg :: (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 PFCF counterparts, managing environments via type functions, e.g. `chRecv`.

```
chRecv :: Chan c -> Process '[c :-> (Delg (e :: d)) :? End] ()
```

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

Since Haskell does not have subtyping, subfecting is explicit using `Sub f g` (mapping from `ef` to `m f a -> m g a`, with one instance shown in Figure 4 (computing `:+` as an upper bound). This models the subfecting 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` (represented by `Fix a`) is computed via a type-level function given an affine effect equation `s -> a :* b + b`.

#### Example

The following simple example corresponds to session calculus term `vc.\text{v}(\text{c}!\text{d}!\text{Ping})`.

```
client (c :: (Chan (Ch "c"))) = new \(\langle d :: (Chan (Ch "d")) , d' \rangle \rightarrow
\text{do } chSend c d
\text{Ping} \leftarrow \text{recv} d'\text{ printf "Client: got a ping"}
```

server c = do { k \leftarrow chRecv c; k \langle x \rightarrow \text{send} x \text{Ping} \rangle
\text{process} = \text{new} \langle\langle c, c' \rangle \rightarrow \text{(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 $F = \mathcal{P}(\rho \tau, \text{val} \tau, \text{alloc} \tau \forall \rho, \tau)$, where $\rho$ are regions, with $(\mathcal{F}, \cup, \emptyset, \oplus = \cup, \setminus, F^* = F)$. We add reference types $\text{ref}_r \tau$ tagged with their region and type, and constants:

\[
\text{get} : \text{ref}_r \rho \tau \rightarrow \tau \\
\text{put} : \text{ref}_r \rho \tau \rightarrow \tau (\text{val} \rho \tau) \\
\text{new} : \tau \rightarrow \{\text{alloc} \rho \tau\} \rightarrow \text{ref}_r \tau (\text{with fresh } \rho)
\]

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:

\[
\text{handler for the encoding) forwards requests to the state handlers by }
\]

\[
\text{ref}_r \tau (\text{with fresh } \rho)
\]

9.2 Monadic metalanguage for effects

We considered here an impure variant of $\lambda$PCF 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 $\lambda$PCF (call it $\lambda$metaPCF) would extend the syntax and type system of pure $\lambda$PCF with $\text{with } M, N :: \ldots \mid \text{let } x \leftrightarrow M; N \mid T_! \tau$ and typing rules:

\[
\begin{align*}
\Gamma \vdash T : \tau & \\
\Gamma \vdash x : \sigma + N : T_! \tau & \\
\text{let } x \leftrightarrow M; N : T_! \tau & \\
\Gamma \vdash \langle M \mid \text{with } \Gamma \rangle : \tau
\end{align*}
\]

for an effect monoid $(\mathcal{F}, \cup, I)$. 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 $(T)$ raises a pure term to a trivially effectful term. This is essentially what we have in our Haskell implementation (Section 8).

Our encoding from $\lambda$PCF to the session calculus can be re-worked for $\lambda$metaPCF. 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 $\lambda$-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 $\text{let } x = \text{in } \mathcal{N}$ 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)^{\text{ref}_r \sigma} = \text{ref}_r \sigma (\text{of a fresh singleton type}) and type 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 $\lambda$-calculus part of $\lambda$metaPCF via $\lambda f \mid - \lambda$, defined for functions as:

\[
C(x, M) = \text{v}d_t. (r!() + d!((p, q), p!r) \mid [M]_\rho)
\]

This is the same as in Section 3, but with the effect channel carriers erased. The interpretation of the monadic fragment is written $[\lambda f \mid - \lambda]$. Consider then the following encoding of a term:

\[
\text{let } f \leftarrow \lambda x . \text{put } x; f 0^{\text{ref}_r \sigma} = \text{v}ea. q, \text{v}e(q, \text{ref}_r \sigma) \mid [M]_\rho)
\]

\[
\text{let } f \leftarrow \lambda x . \text{put } x; f 0^{\text{ref}_r \sigma} = \text{v}ea. q, \text{v}e(q, \text{ref}_r \sigma) \mid [M]_\rho)
\]

\[
\text{let } f \leftarrow \lambda x . \text{put } x; f 0^{\text{ref}_r \sigma} = \text{v}ea. q, \text{v}e(q, \text{ref}_r \sigma) \mid [M]_\rho)
\]

\[
\text{let } f \leftarrow \lambda x . \text{put } x; f 0^{\text{ref}_r \sigma} = \text{v}ea. q, \text{v}e(q, \text{ref}_r \sigma) \mid [M]_\rho)
\]

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 $T_! A$ 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.
tion of the session calculus (without \(\pi\) 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 \(E ::= \rho_1[\tau] | \rho_2[\tau] | \text{spawn}(E)\) denoting sending on a channel \(\rho_1\) receiving on a channel \(\rho_2\) of type \(\tau\) 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 \(\rho_1[\tau_1], \rho_2[\tau_2], \rho_3[\tau_3]\) can be mapped to a session effect environment \(\rho_1 : [\tau_1], \rho_2 : [\tau_2], \text{end}, \rho_3 : [\tau_3].\text{end}\).

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 bar for congruence under IO-subtyping can justify the correctness of the optimal encoding of the \(\lambda\)-calculus by Milner [33]. This was not possible in the polyadic \(\pi\)-calculus [33]. After [44], many works on typed \(\pi\)-calculus 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 \(\pi\)-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 \(\pi\)-calculus 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 \(\pi\)-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 \(\lambda\)-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 \(\pi\)-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 sequentality, 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/KO11715/1, EP/K034413/1, EP/L00058X/1, and EU project FP7-612985 UpScale.