Abstract
Effect and session type systems are two expressive behavioural type systems. The former is usually developed in the context of the λ-calculus and its variants, the latter for the \( \pi \)-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; F.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 \( \pi \)-calculus. 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-typed 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 simple mutable memory cell:

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

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 \( c \text{cl}(x) \) then the process recurses with the same parameters. In the put branch, a value \( y \) is received on \( c \text{cl}(y) \) which becomes the new stored value via the recursive call. The initial stored value is \( i \). Thus, \( \text{Var}(c, i) \) handles \text{get} and \text{put} operations performed by a client process.

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

\[
c \triangleright \text{get : } c(?x), c \triangleright \text{put : } cl(x + 1) | \text{Var}(c, i)
\]

On channel \( c \) the get branch is selected (via \( \triangleright \)), 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 \( c : \triangleright \{ \text{get : } ?Z, \triangleright \text{put : } !Z \} \). 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 3.5 and Section 9.1).
- We extend the encoding to a parallel variant of PCF where effects may interfere (Section 5). This requires only a small extension to the duality predicate of session typing.
- We instantiate PCF with concurrency primitives and an effect system for sessions. We reverse the embedding of Section 3, giving an embedding of the session calculus into this instantiation of PCF (Section 7). This shows that effect systems are powerful enough to encode session types. This is leveraged to give a new implementation of session types in Haskell (Section 8), enlarging the typability from previous implementations. This is our *Artifact*, available at http://dorchard.co.uk/popl16.

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

2. Background: two typed calculi

2.1 PCF with effects

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

$$M, N ::= V | M N | \text{case } M \text{ of } e \rightarrow N | C$$

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

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

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

**Definition 1 (Effect algebra).** Let $F$ 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 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:

$$\sigma, \tau ::= \sigma \oplus \tau | \text{nat} | \text{unit}$$

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

Figure 1 gives the type-and-effect rules. The $(\text{var})$ and $(\text{abs})$ rules describe variable use and abstraction as pure (with $I$). For $(\text{rec})$, the effect of the function body $F$ becomes a latent effect. The $(\text{sub})$ rule allows effects to be overapproximated (with respect to the partial order $\sqsubseteq$). The $(\text{app})$ 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 $(\text{const})$ rule introduces a constant of type $C_\tau$ with effects $C_F$. The effect of (case) sequentially composes 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 $(\cdot)^*$ is often required to provide a valid typing for a recursive definition.

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

$$\Gamma \vdash \text{let } x = M \text{ in } N : \tau, F \cdot G$$

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

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

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:

$$\emptyset \vdash \text{get } : \tau, [\text{rd } \tau] \quad \emptyset \vdash \text{put } : \tau \rightarrow [\text{wr } \tau] \rightarrow \text{unit}. \quad (2)$$

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

$$\beta (\lambda x. M) V \equiv M[V/x]
\text{(rec) rec } (\lambda f. \lambda x. M) \equiv \lambda x. M \cdot \text{rec } (\lambda f. \lambda x. M)/f
\text{(case) case } 0 \cdot 0 \rightarrow M, (\text{succ } x) \rightarrow N \equiv M
\text{(case) case } (\text{succ } V) \cdot 0 \rightarrow M, (\text{succ } x) \rightarrow N \equiv N[V/x]$$

We add two further equations on let:

$$\text{(let-id) let } x = M \text{ in } x \equiv M
\text{(let-assoc) let } y = (\text{let } x = M \text{ in } N) \text{ in } N' \equiv \text{let } x = M \text{ in } (\text{let } y = N \text{ in } N') \quad (\text{if } x \notin \text{fv}(N'))$$
Note, (β) above is equivalent to (let x = V in M) ≡ M[V/x]. Equality extends to type-and-effect judgements Γ ⊢ M ≡ N : τ, F where Γ ⊢ M : τ, F and Γ ⊢ N : τ, F. In the usual way, η-equality is type-dependent: M ≡ Λx.M x holds only when M is a function type. Further, in PFCF η-equality holds only when M is also pure. That is, Γ ⊢ M ≡ (λx.M x) : σ F → τ, I.

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

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

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

0 ⊢ M : τ, F ∧ C[M] → D[N] ⇒ 0 ⊢ N : τ, G ∧ G ⊆ F

2.2 Session calculus

We consider the π-calculus with session primitives, which we call the session calculus. Figure 2 shows the syntax, where l ranges over labels, 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 τ with c = c. Intuitively, names c and τ are two dual endpoints. Throughout we elide trailing occurrences of 0 and end, e.g., writing r!⟨x⟩ instead of r!⟨x⟩.0.

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

(β)

\( cI(x).P, P \upharpoonright (V), P \to Q \mid Q \)

(β-channel)

\( cI(d).P, P \upharpoonright (c0)Q \to P \mid c0/d \mid Q \)

(β+) \text{+} cI(l).P, P \upharpoonright c0(Q) \to \text{+} cI(d).P, P \mid c0/d \mid Q \)

(β→↓)

\( l, P \upharpoonright \langle i : Q \rangle \to P \mid Q \), \((l_i \in i)\)

(res)

\( P \to P' \Rightarrow vcP \to vcP' \)

(par)

\( P \to P' \Rightarrow P \mid Q \to P' \mid Q \)

(str)

\( Q \equiv P \rightarrow P' \equiv Q' \Rightarrow \)

(if1)

\( if0 = 0 \rightarrow \text{then } P \text{ else } Q \to P \)

(if2)

\( if\{v = 0\} \rightarrow \text{then } P \text{ else } Q \to Q \)

Equation (β) reduces complementary send/receive actions of a value V over a linear channel c; similarly equation (β-channel) gives the interaction of sending/receiving a linear channel d over a linear channel c. Equation (β+) resembles equation (β-channel) but involves replicated input over channel c, that is, c is not linear in the usual sense, but can repeatedly receive values via the persistence of + cI(d).P. Equation (β→↓) gives the reduction of complementary branching/select. Reduction is congruent with respect to parallel composition (par) and restriction (res). Furthermore, (str) extends → along the structural congruence relation ≡ [57], which provides that | is a commutative monoid on unit, amongst other things.

**Definition 6** (Session types [4, 57]). Session types record sequences of typed send (!![τ]) and receive (?[τ]) interactions, terminated by end, branched by select (⊕) and choice (κ) interactions, output (!![τ]), and with recursive types μα.S:

\( S := \times \tau.S \mid \times \tau.S \mid \times \tau.S \mid end \ | \oplus[i : S] \mid \kappa[i : S] \mid \mu\alpha.S \mid \alpha \)

where τ ranges over value types, including session types (Figure 2), l ranges over labels, I : S is a sequence of label-session pairs, and recursive types obey the fixed-point law μα.S = S[μα.S/α].

We assume recursive types are guarded and carried types (e.g., τ in !τ.S are closed; end is often omitted. We define the alias !τ[τ] := μα.?τ[τ] for replicated input.

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

For a session S, its dual S is defined:

\( ![τ, S] = ![τ, S] \mid ⋆[τ, S] = ![τ, S] \mid ⋆[τ, S] = ![τ, S] \mid ∀l : S_1, l_2 : S_n \equiv &ll[l : S_1, \ldots, l_n : S_n] \mid μ\alpha.S \equiv μ\alpha.S \)

Figure 2. Syntax of the session calculus

The predicate + able(Δ) classifies session types in the environment which comprise only replicated outputs or end.

+ able(Δ) ⇔ ∀(c : S) ∈ Δ, + able(S)

+ able(S) ⇔ (∃T. S = ![τ, T \land + able(T)] \lor (S = 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 Γ ; Δ ⊢ P where Γ is a mapping from variables to base types and Δ is a mapping from channels to session types. In (par), we check two processes are composable or not by ⊗ (see below); (send-receive) follows the same condition where composability of d is checked. Since + cI(d).P replicates P, it cannot contain free linear session names, guaranteed by + able(Δ) in (receive). Note that replicated outputs can be weakened (weak*). The rest is standard from [57] with subtyping as in [10, 11, 19].

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

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

\( (c : S) \Rightarrow (c : ![τ], S) \Rightarrow (c : ![τ], S) \Rightarrow (d : T) \)

where c ≠ d. Thus, dual names must have dual types and channels may only appear in two different processes if they are both outputs (+). A partial commutative function ⊗ takes the union of two environments if they are balanced, i.e. Δ_1 ⊗ Δ_2 = Δ_1 ∪ Δ_2 if Δ_1 ⊓ Δ_2 with ⊓ lifted to all pairs of named session types Δ_1 × Δ_2 otherwise it is undefined. Note, μα.?τ[τ] is dual to both μα.?τ[τ] and ![τ], but (c : μα.?τ[τ]) ⊗ (c : ![τ]) is undefined.

Since a session environment represents forthcoming communications, during process interactions, the session environment will change. We define the relation Δ → Δ′ [25] as follows:

\( Δ, c : ![τ], S, \tau : ![τ] \Rightarrow Δ, c, S, τ : S, τ \)

\( Δ, c : ![τ], S, τ : ![τ] \Rightarrow Δ, c : S, τ : T \)

\( Δ, c : ![τ], S, τ : ![τ] \Rightarrow Δ, c : S, τ : ![τ] \)

**Proposition 2** (Subject reduction, [4, 57]). Suppose Γ ; Δ ⊢ P and P →* Q. Then Γ ; Δ′ ⊢ Q with Δ′ →* Δ′. In addition, if Δ is (self)-balanced, then Δ′ 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 (outgoing). To see why this is needed, rather than just using the effect channel directly, consider a standard encoding for pure let-binding:

\[
\text{let } x = M \text{ in } N \]_{\Gamma} = \nu q. (M)_{\Gamma} \vdash x : (N)_{\Gamma}
\]

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 \( e : T \) respectively which may be different. The balanced predicate in the (para)typing rule prevents this composition. Instead, our encoding has an intermediate form parameterised by effect carriers, written \( \llbracket - \rrbracket_{e.o} \) for an incoming effect channel carrier \( e \) and outgoing carrier \( o \). The let-binding encoding is then:

\[
\text{let } x = M \text{ in } N \]_{\Gamma} = \nu q. (M)_{\Gamma} \vdash e : (N)_{\Gamma}
\]

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

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

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

2. a terminator process \( T(eff) \) such that \( T(eff) \vdash \ast \emptyset \)

3. an interpretation function \([-,-] : F \rightarrow S \) from effect algebra elements to session types, satisfying the following homomorphism property (discussed more in Section 3.4) where \( \bullet \) is a (partial)

\[
\text{sequential composition for session types:}
\]

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

4. an encoding \( \llbracket - \rrbracket_{e.o} \) for all constants \( C \), such that, \( \exists P: \forall g, (\emptyset \vdash C \cdot g)_{\llbracket e.o \rrbracket} = r \vdash [C \cdot g], e.i : [C \bullet g], \llbracket g \rrbracket \vdash P \)

5. an encoding for case@ 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., \( \llbracket \cdot \rrbracket \) instead of \( \llbracket \cdot \rrbracket \).

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

\[
\llbracket M \vdash \tau : F \rrbracket_{\llbracket e.o \rrbracket} = [\llbracket \cdot \rrbracket], r : [T] \vdash \llbracket eff \rrbracket, e.i, \llbracket g \rrbracket, \llbracket T \rrbracket
\]

where \( [\cdot] \) and \( [\cdot] \) map environments to values and session environments respectively, and \( [\cdot] \) maps FPCF types \( \tau \) to session types (defined below). The intermediate encoding is composed in parallel with a process which sends effect channel \( eff \) on \( e \) and receives a channel on \( o \) (which is similarly named \( eff \) as this ends up being the same channel) before completing with \( T(eff) \).

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

\[
[\sigma F \vdash \tau] = \ast \llbracket [\cdot \rrbracket, [F \bullet g], \llbracket g \rrbracket, \llbracket \cdot \rrbracket]
\]

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 \( [\cdot \cdot] \) value for the argument, (2) an effect carrier to receive an effect channel capable of simulating effects \( F \bullet g \) (3) a channel to send an effect channel capable of effects \( g \), and (4) a channel to send a \( [\cdot \cdot \cdot] \) value for the result.

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

\[
[\emptyset] = \emptyset \quad [\Gamma, x : T : \llbracket \cdot \rrbracket] = [\Gamma] \vdash x : [T] \\
[\emptyset] = \emptyset \quad [\Gamma, x : \sigma \vdash \tau]_{\llbracket F \rrbracket} = [\Gamma], x : \llbracket \sigma \bullet \tau \rrbracket
\]

\[
(3)
\]

**Terms/derivations** The core embedding is the intermediate \( \llbracket - \rrbracket_{e.o} \) from FPCF derivations to session-calculus derivations, of the form:

\[
\llbracket M \vdash \tau : F \rrbracket_{\llbracket e.o \rrbracket} = \forall g. [\Gamma]_{\llbracket \cdot \rrbracket}, [T] : [\cdot \cdot \cdot], e.i : [F \bullet g], \llbracket g \rrbracket \vdash P
\]

(4)
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 $\mathcal{E}$ 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:

\[
\begin{align*}
ei &: \llbracket F \bullet g \rrbracket, \mathcal{E} \vdash \llbracket g \rrbracket \vdash (\lambda M^{\sigma,\rho,\nu} \cdot N^{\sigma,\rho,\nu}) \\
eq &: \llbracket (G \bullet h) \rrbracket, \mathcal{E} \vdash \llbracket h \rrbracket \vdash \chi_{\mathcal{E}}(x) \cdot (\lambda N^{\sigma,\rho,\nu} \cdot \eta_{\mathcal{E}}^\ast) \\
ei &: \llbracket F \bullet g \rrbracket, \mathcal{E} \vdash \llbracket g \rrbracket \vdash (\lambda M^{\sigma,\rho,\nu} \cdot \eta_{\mathcal{E}}^\ast) \\
eq &: \llbracket (G \bullet h) \rrbracket, \mathcal{E} \vdash \llbracket h \rrbracket \vdash \chi_{\mathcal{E}}(x) \cdot (\lambda N^{\sigma,\rho,\nu} \cdot \eta_{\mathcal{E}}^\ast)
\end{align*}
\]

We show first the encoding of the \(\lambda\)-calculus subset of $\text{FPFC}$. 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, \(\llbracket x \rrbracket, c \vdash e_i(x), r(x), \chi_c(x)\) 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 $\tau$ is a function type then $x$ is a channel variable and $r!\langle x \rangle$ is typed by (channel-send) (Figure 3). If $\tau$ is a ground type then $x$ is a value variable and $r!\langle x \rangle$ is typed by (send). The result of the encoding has similar overloading for either channels or values.

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

\[
\begin{align*}
\langle x \rangle, c \vdash \lambda,M_{\sigma,\rho,\nu}^{\tau,\rho,\nu} &= \nu q.(\cdot) r!\langle x \rangle, \chi_{\mathcal{E}}(x)
\end{align*}
\]

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

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

\[
\begin{align*}
\langle M \rangle, q_{\sigma,\rho,\nu}^{\rho,\tau} &= \nu q.s.ea.eb.p. \\
\langle \cdot \rangle r!\langle x \rangle, y!\langle p, \cdot \rangle, \chi_{\mathcal{E}}(x), \chi_{\mathcal{E}}(y)
\end{align*}
\]

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 $\mathcal{E}$ 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 $\text{succ}$ resembles the $\lambda$-abstraction encoding, since $\text{succ}$ is a function:

\[
\begin{align*}
\langle 0 \rangle, c \vdash e_i(\cdot), r!\langle 0 \rangle, \chi_{\mathcal{E}}(c)
\end{align*}
\]

(5)

\[
\begin{align*}
\langle \text{succ} \rangle, c \vdash \nu d.(\cdot) r!\langle d \rangle, \chi_{\mathcal{E}}(c), \cdot d?\langle p, e_i, \cdot \rangle, p?\langle x \rangle, \chi_{\mathcal{E}}(\text{succ} c), \chi_{\mathcal{E}}(\text{succ} c), r!\langle \cdot \rangle
\end{align*}
\]

(6)

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

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

The above encodes the $\lambda$-calculus subset of $\text{FPFC}$, giving the sequential composition of effects. We now encode the control-flow operators which PCF adds to the $\lambda$-calculus: $\text{case}$ and recursion.

**Conditionals** Our type-and-effect system for $\text{FPFC}$ defines the effects of $\text{case}$ 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 $\text{case}$.

Traditional set-based effect systems often provide rules for case which either have the same effect in each branch or take the union (least-upper bound) of the branches (i.e., $\oplus = \cup$ in our calculus). Consider the following alternate type-and-effect rule:

\[
\begin{align*}
\Gamma \vdash M : \tau, G \mid \Gamma, x : \text{nat} \vdash N_1 : \tau, G \mid \Gamma, x : \text{nat} \vdash N_2 : \tau, G \\
\Gamma \vdash \text{case } M \text{ of } 0 \to N_1, \text{ (succ } x) \to N_2 \text{. (7)}
\end{align*}
\]

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

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

\[
\begin{align*}
\langle \text{case } M \text{ of } 0 \to N_1, \text{ (succ } x) \to N_2 \rangle, q \vdash \text{ where } \nu c.(\cdot) r!\langle V \rangle.P \vdash c!\langle \text{pred } V \rangle \mid \chi_{\mathcal{E}}(x), P \rangle
\end{align*}
\]

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

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

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

\[
\begin{align*}
\langle \text{rec } (\lambda f.\lambda x. M) \rangle, c \vdash \nu d.(\cdot) r!\langle d \rangle, \chi_{\mathcal{E}}(c), \cdot d?\langle p, e_i, \cdot \rangle, p?\langle x \rangle, \chi_{\mathcal{E}}(\cdot) \mid [d/f]
\end{align*}
\]

The key difference between this and the abstraction encoding is the syntactic substitution $[d/f]$ of $f$ with the channel $d$. Thus, if $f$ is free in $M$ then an intermediate encoding \(\llbracket f \rrbracket, c \vdash d?\langle \cdot \rangle \mid [d/f]\) = $ec_i\langle \cdot \rangle, s!\langle d \rangle, \chi_{\mathcal{E}}(c)$, 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 $+:$
In the parameters to our embedding (Definition 8), the effect anno-
lowing homomorphism property, which ensures the continuation-
simulated over one effect channel with branching/selection:
$$F \vdash M : \tau, F + G$$
The interpretation of a typing derivation ending in an instance of
the interpretation above differs from selecting the L label before passing on the channel to the interpretation of M. Subtyping on the selection of L introduces the R branch giving the following typing for ei when $[\gamma]$: $\text{Ex}$.

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 $F \subseteq F + G$ 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 type of expression $+:
\emptyset; \text{eff} : \alpha. S \vdash H(\text{eff}) \Rightarrow S < \&[\alpha] : \otimes\alpha, R : \otimes\alpha]$
(subtypes of a branching offer more choices than the supertype).

Remark 1. Setting $\otimes = +$ gives a free representation of alteration
in effects (computation trees). For the general encoding, re-
stricted case (Def. 9) can be composed with subtyping in the
premises to introduce + effects.

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

### 3.4 Homomorphic embedding of effect annotations

In the parameters to our embedding (Definition 8), the effect anno-
tation interpretation $[\gamma] : \mathcal{F} \rightarrow S$ is required to satis-fy the fol-
lowing homomorphism property, which ensures the continuation-
passing style approach is well typed:

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

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

$$T \bullet end = T \quad end \bullet T = T$$

$$\not{[\gamma]} : (\mathcal{S} \cdot T) \quad ?[\gamma] : (\mathcal{S} \cdot T)$$

$$@[\vec{S}] : (\mathcal{S} \cdot T) \quad &[\vec{S}] : (\mathcal{S} \cdot T)$$

$$*{[\vec{S}]} : (\mathcal{S} \cdot T) \quad !{[\vec{S}]} : (\mathcal{S} \cdot T)$$

$$(\mu \alpha.S) \bullet (\mu \beta.T) = \mu_\gamma.S[\gamma/\alpha] \cup T[\gamma/\beta]$$

where $\vec{S} \cdot T$ is the vector-scalor lifting of $\bullet$. All other cases are
undefined, e.g., $*{[\vec{S}]} \cdot T = \perp \perp$. 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.:

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

### 3.5 Examples

In our examples, we define effect handlers as recursive processes
via replicated input, of the form $\nu h.(\ast h(c, x), P \mid \tilde{R}(c, V))$ for
some effect channel $c$ and value arguments $\tilde{V}$. The process $P$ tends
to contain an output on $\tilde{R}$ 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:

$$\langle F \cdot \tau \rangle :: [F] = \otimes [\text{put }] [\text{get }] [F]$$

The interpretation of alternation effects $\oplus$ here is the free encoding
via subfecting (Section 3.3). The effect handler $H$ is defined
similarly to the variable agent in the introduction, where $H(\text{eff}) =
\text{eff} : \alpha. S \vdash \alpha, \text{get } \vdash [\text{get }] \cdot [\text{put }]$. 

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 $F = P\{rd, \tau, \text{get } \vdash \perp\}$, with
$(\mathcal{F}, \cup, \otimes)$ for sequencing and $\oplus = \cup$, $\tau^* = F$. 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),
except from the effect interpretation $[\text{eff}]$ which is instead given
in terms of a recursive type over a selection:

$$[F] = \mu \alpha.S \text{ where } \forall (\text{rd } \tau) \in F \Rightarrow \oplus \vdash \text{[rd]} : \Box[\text{[get]}] \cdot S < S$$

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, $\langle [\text{rd } \tau, \text{get } \tau]\rangle = \mu \alpha. \oplus [\text{[rd]} : \Box[\text{get}]] \cdot \alpha, \text{get } \vdash [\text{get}]. \text{This interpretation of effects into session types is well-typed with respect}

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 Ciçek et al. [12] and Daniel-
sons, via the Thunk annotated monad) [13]. The sequential part of
this effect system is given by the ordered monoid $\langle\mathbb{N}, +, 0, \leq\rangle$ with
$\oplus = \max$. For recursion, the domain is extended to $\mathbb{N} \cup \{\omega\}$ such
that $n^+ = \omega$ which is greatest element w.r.t $\leq$ and absorbing w.r.t
$+$. Thus, $\text{FPFCF}$ can be instantiated for this effect system with
the additional constant $0 \vdash \text{tick } : 1$. unit.
Resource counting effects are embedded via the following handler, terminator, and operation encoding:

\[ H(c) = \mathtt{thr} . \{ \mathtt{tick}(c) . \mathtt{stop}(0) \} \] \[ T(c) = \nu d . (c(d), d) \prec \mathtt{stop} \] \[ \mathtt{tick}[c] = \nu d . (e \mathtt{tick}(c) . e \mathtt{stop}(d) . d < \mathtt{tick} . d ! \mathtt{unit} . \mathtt{stop}(c) \]  

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

\[
[0] = \text{end} \quad [n+1] = ![\cdot][\text{tick} : [\cdot][\text{unit}]] . [n]
\]

\[
[\omega] = \mu a . ![\cdot][\text{tick} : [\cdot][\text{unit}]] . a
\]

The handler has type \( c : * ![\cdot][\text{tick} : [\cdot][\text{unit}]] \). Type soundness of this encoding requires a small extension to definition of session type duality for this encoding:

\[
\forall n. \text{dual} ![\cdot][\text{tick} : [\cdot][\text{unit}]] \leq ![\cdot][\text{tick} : [\cdot][\text{unit}]]
\]

We extend the duality function to a symmetric relation \( \sim \) giving:

\[
\forall \varphi, \psi . \varphi \sim \psi \quad \text{iff} \quad \forall \omega . \varphi(\omega) = \psi(\omega)
\]

and \( \sim \) 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 barbed congruence over our session calculus.

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

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

\[
\exists P, d, y . \| V \|_{\mathtt{c}} \equiv \nu d . (e \mathtt{tick}(c) . r ! \langle y \rangle \mathtt{stop}(c) . P)
\]

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

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

\[
\nu \varphi . (\nu \varphi (e \mathtt{tick}(c) . e \mathtt{stop}(d) . d < \mathtt{tick} . d ! \mathtt{unit} . \varphi)) \equiv \nu \varphi . (e \mathtt{tick}(c) . e \mathtt{stop}(d) . d < \mathtt{tick} . d ! \mathtt{unit} . \varphi)
\]

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

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

\[
\begin{array}{ll}
\nu \varphi . (\nu \varphi (e \mathtt{tick}(c) . e \mathtt{stop}(d) . d < \mathtt{tick} . d ! \mathtt{unit} . \varphi)) & \equiv \nu \varphi . (e \mathtt{tick}(c) . e \mathtt{stop}(d) . d < \mathtt{tick} . d ! \mathtt{unit} . \varphi) \\
\end{array}
\]

The main theorem of this section is then equalational soundness of the encoding with respect to \( \beta\eta \)-equality of \( \mathtt{FPCF} \).

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

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

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]):

\[
\Gamma \vdash M : \tau, F \vdash \Gamma \vdash M : \tau, F, x : \tau \Rightarrow \Gamma \vdash M \equiv N : \tau, F
\]

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

Note, these results are for the global encoding. Correctness of any effect-specific embeddings of effect operations (e.g., \textit{get} and \textit{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 \( \mathtt{FPCF} \) for parallel composition, written \( M \parallel N \). Unrestricted effects in parallel branches may cause race conditions, e.g. \textit{put } (get + 1) \textit{if put } (get + 2) \textit{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:

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

\[
\Gamma \vdash \text{unit} : F \land \Gamma \vdash \text{unit} : G \\ M \parallel N : F \land G
\]

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

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 \text{unit} \parallel \text{unit} \rightarrow \text{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:

\[
\nu_r.(x?c, \nu_l?d) \rightarrow \{r \rightarrow d!l(x), \text{fl}(c, x), \nu_r : d?l(y), \text{fl}(c, y), \text{stop} : 0\}
\]

This handler has the following recursive session type for eff:

\[
\mathit{eff} : \mu \alpha. ?[S] \mathit{α} \quad \text{where} \quad S = \langle \mathit{rd} : ![\tau], \mathit{wr} : ![\tau] \rangle, \text{stop} : \text{end}
\]

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

\[
\langle \mathit{get} \rangle!e.o.e = e?l(c).\nu_e.(c).\mathit{rd}!l(x).\mathit{wr}!l(y).
\]

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

\[
\Gamma \parallel \text{end} : \langle\mathit{rd}\langle\pi\rangle\rangle \rightarrow F = \nu_e.\langle\mathit{rd}\langle\pi\rangle\rangle \rightarrow \nu_e.\langle\mathit{rd}\langle\pi\rangle\rangle \rightarrow F
\]

The parallel composition operator is then defined:

\[
\langle \mathit{rd} \parallel \mathit{wr} \rangle \parallel [S] = \nu_e.\langle \mathit{rd}\langle\pi\rangle\rangle \rightarrow \nu_e.\langle \mathit{rd}\langle\pi\rangle\rangle \rightarrow F
\]

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

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

\[
\!*\![\mathit{rd} : ![\tau]] < \!*\![S_1] \quad \!*\![\mathit{wr} : ![\tau]] < \!*\![S]\]
\]

Thus, the interpretation of parallel effect annotations is 

\[
[F \parallel G] = \!*\![S]\big(n \text{ max } m\big)
\]

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 \(\forall \mathit{ndual} \!*\![S]\big(n\text{ max } m\big)\).

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

---

**Theorem 3** (Correspondence for parallel effects). \(\forall M, F, \tau, \Gamma, \). \n
- (sound) \(\forall N. \Gamma \vdash M : \tau, F \land (M \rightarrow N) \land F \subseteq G \Rightarrow \exists P. (\Gamma \parallel M : \tau, F, P) \land P \equiv (\Gamma \parallel N : \tau, G, P)\)

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

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

\(\Gamma \vdash M \equiv N : \tau, F \Rightarrow (\Gamma \parallel M : \tau, F, P) \equiv (\Gamma \parallel N : \tau, F, P)\)

For this extended encoding, equational completeness (w.r.t observational equivalence (12)) does not hold. This is because any process \(P \equiv (\Gamma \parallel N : \tau, F, P)\) 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 \(*\). 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 1

<table>
<thead>
<tr>
<th>FPCF (\pi_S)</th>
<th>core (\beta\eta)</th>
<th>if &lt;</th>
<th>eq. 10</th>
<th>(\forall)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\bullet)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\ast)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\oplus)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\otimes)</td>
<td>(\checkmark)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\parallel)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\oplus)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\ominus)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\ominus)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\ominus)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\mathcal{F})</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The \(\oplus\) or \(\ominus\) line represents the restricted version of case which has \(F \ominus F = F\) and \(F \ominus G = \perp\) when \(F \neq G\). The \(\oplus\) or \(\ominus\) line represents \(\ast\) constructed via subeffecting with the free alternation operator \(\oplus\) (Section 3.3). The \(\mathcal{F}\) line represents effects with varying types, such as effects for first-class references, which are discussed in Section 9.1. Note that the requirements here are only on the general encoding. Specific instantiations may use all/any of these features, e.g. Example 4 used the duality relation extension of equation 10.

**Linear control-flow effects**

The examples given in Section 3.5 can be characterised by linearity in their control flow (equivalent to algebraic effect handlers [3, 46] that are linear in their continuation). Effects such as exceptions (which interrupt control-flow) and non-determinism (with branching control-flow effects) may plausibly be captured via encodings that explicitly include their continuation, e.g. \([\text{let } x = \text{choose } V_1 V_2 \text{ in } M]\). 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/session types via alternation \(\oplus\) and replication via recursion and \(\ominus\). 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 session 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 $\uplus$ replaces select $\uplus$ and branch $\&$, with $\tau ::= \text{nat} \mid \text{unit} \mid [S]$ and $\mathcal{S}, T ::= ![\tau].S \mid ?[\tau].S \mid e!\tau].S \mid \text{end} \mid S + S + \mu\alpha.S \mid \alpha \mid \ominus S$

where $\ominus S$ is an intermediate representation for the session type of channels being composed via $\ominus$ (balanced composition). We assume equi-representative equations on $S$, e.g., $\mu\alpha.S = S[\mu\alpha.S/\alpha]$

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

- $(\mathcal{F} \bullet \emptyset)$ where $I = \emptyset$ is the empty mapping, is pointwise sequential composition of session types (via $\circ$, Section 3.4) where $\forall e : S \equiv \mathcal{T} \in \mathcal{G}$ then $e : S ; T \in (\mathcal{F} \circ \mathcal{G})$ but with additional equations for balancing composition triggered by $\ominus$ types e.g., $\Delta \ominus (c : \ominus S) = \perp$ if $\exists \tau : S \notin \Delta$ amongst other cases (the technical report [41] gives the full definition).

- $\uplus$ is set union $\cup$ if $\Diamond | \Diamond \geq \perp$ satisfies the following condition:

$$\forall e : S \in \Diamond \Rightarrow e : S \in \Diamond, \forall e \notin \dom(\Diamond)$$

i.e., a channel is either in $\Diamond_1$ or $\Diamond_2$, or in both but with the same session type, otherwise $\Diamond_1 \uplus \Diamond_2 = \perp$.

- $\ominus$ is a preorder where $\forall \Diamond, c, e$ where $e \notin \dom(\Diamond)$:

$$\Delta \ominus (\Delta, c : \text{end}) \equiv (\Delta, c : S) \ominus (\Delta, c : S + T)$$

$$\Delta \ominus (\Delta, c : \ast(S)) \equiv (\Delta, c : S + T)$$

(15)

- $\diamondsuit = \ominus$, for parallel effects, takes the union of two mappings if they are balanced in their channels, otherwise $\perp$;

- $F^\ast$ is defined $\forall e : S \in F$ then $e : \mu\alpha.(\ominus e + S \ominus \alpha) \in F^\ast$.

We extend $\mathcal{FPCF}$ values with channel values $C$ ranged over by $c, d, e$ and their dual endpoints $\overline{c}, \overline{d}, \overline{e}$. 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 $\ominus$ operation above is not the least-upper bound with respect to $\ominus$. Defining subeffecting as subset inclusion amongst other cases (15) above would give least-upper bounds with the union-like behaviour of $\ominus$, however this is unsound: arbitrary session types could be introduced without a corresponding implementation. This shows the need for separate $\ominus$ and $\ominus$.

**Sending, receiving and restriction operations** We add to the constants of $\mathcal{FPCF}$ operations that send and receive values, send and receive channels, and restrict channels:

- $\text{send}_{c,r} : Ch c \rightarrow \tau \quad \{c : ![\tau]\} \rightarrow \text{unit}$
- $\text{rsend}_{c,d,s} : Ch c \rightarrow Ch d \quad \{c : ![s], d : \ominus s\} \rightarrow \text{unit}$
- $\text{chSend}_{c,d,s} : Ch c \rightarrow Ch d \quad \{c : ![s], d : \ominus s\} \rightarrow \text{unit}$
- $\text{recv}_{c,r} : Ch c \rightarrow \tau \quad \{c : ![\tau]\} \rightarrow \tau$
- $\text{chRecv}_{c,d,s,r} : Ch c \rightarrow Ch d \rightarrow \tau \quad \{c : ![s], d : \ominus s\} \rightarrow \tau$
- $\text{new}_{c,d,s} : Ch c \rightarrow Ch d \rightarrow \tau \quad \{c : ![s], d : \ominus s\} \rightarrow \tau$

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 $\text{send}$ and $\text{recv}$ operations correspond to session send/receive prefixes with effect types describing the single action on their channel. For $\text{chSend}$, 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 $\ominus$ operator; $\text{rsend}$ is identical to $\text{chSend}$, but with the $!$ session type. The $\text{chRecv}$ 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 $F \bullet \{d : s\}$. From this, a computation is returned with $F$ channels where $d, c \notin F$. This typing is important for the typability of replicated input. The new operation is similarly higher-order, where the resulting effect is the effect of the parameter function, but with the session types $c : s, \overline{c} : \overline{s}$ deleted from the environment since they are in scope only for the parameter function.

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

- $\langle \text{send} c V, s \rangle \rightarrow (\text{unit}, enq s \tau V)$
- $\langle \text{rsend} c V, s \rangle \rightarrow (\text{unit}, enq s \tau V)$
- $\langle \text{chSend} c d s, \tau \rangle \rightarrow (\text{unit}, enq s \tau d)$
- $\langle \text{recv} c, s \rangle \rightarrow (\text{unit}, enq s \tau d)$  
- $\langle \text{chRecv} c d s, \tau \rangle \rightarrow (\text{unit}, enq s \tau d)$
- $\langle \text{new} c d s, \tau \rangle \rightarrow (\text{unit}, enq s \tau d)$

For send, $\text{rsend}$, and $\text{chSend}$, the value or channel is added to the end of the queue belonging to the opposite end-point $\overline{c}$. For recv, a value is dequeued from the endpoint $c$ queue (if it exists in $s$); $\text{chRecv}$ 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 $\text{chRecv}$. This semantics allows asynchronous communication. Section 7.2 defines a stable reduction relation that characterises the reductions of our encoding, which are only the synchronous subset.

The reduction of new exposes a key difference between the two calculi. In processes, $\nu x \cdot P$ introduces channel names $c \in \tau$ in the scope of $P$. In $\mathcal{FPCF}$, new encodes restriction but new $(\lambda x.\mu y.M)\cdot c$ 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:

- $\langle \text{new} (\lambda x.\mu y.M)\cdot c, \tau / \overline{y}, \overline{c}, \overline{\tau} \rangle$  

That is, if $x, y$ are not in the store, then $x, y$ are $\alpha$-renamed to names $c, \tau$ 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:

- $\langle \text{new} (\lambda x.\mu y.M)\cdot c, \tau / \overline{y}, \overline{c}, \overline{\tau} \rangle \rightarrow \langle \text{new} (\lambda x.\mu y.M)\cdot \sigma, \tau / \overline{y}, \overline{c}, \overline{\tau} \rangle$

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

From this semantics, $\equiv$ can be extended with an $\eta$-expansion for new over pure values: $\text{new}(\lambda x.y.V) \equiv V$ and distributivity $P \parallel \text{new}(\lambda x.y.M) \equiv \text{new}(\lambda x.y.P \parallel Q)$ if $x, y \notin \text{fv}(P)$.
7.1 Embedding the session calculus into FPCF

Encoding of types

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

Lemma 5 (Typability). Let \( \Gamma; \Delta \vdash P \) then \[ \exists M. \ [\Gamma; \Delta] P \equiv [\Gamma] M : \text{unit}, [\Delta] \]

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

Definition 15 (Session types to effect sessions annotations).

\[
\begin{align*}
\text{[end]} & = \text{end} & \text{[\text{let}]} & = \text{let} x = v \text{ rec } c \text{ in } [P] \\
\text{[c?(x).P]} & = \text{send } c \text{ [V]} ; [P] & \text{[\lambda d. \text{let} k = \text{chRec } c \text{ in } k (\lambda d.P)]} & = [\text{rec } (\lambda f. \lambda x. \text{let } k = \text{chRec } c \text{ in } (k (\lambda d.[P])) \text{ if } f \text{ unit }] \text{ unit} \\
\text{[\text{rec } (\lambda f. \lambda x. \text{let } k = \text{chRec } c \text{ in } (k (\lambda d.[P])) \text{ if } f \text{ unit }] \text{ unit}} & = \text{send } c d ; [P] & \text{[\text{if } c? \text{ [S].T } \in \Delta]} & = \text{end} & \text{[\text{select } (\text{where } c \rightarrow d, P)} \] \\
\end{align*}
\]

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

\[
\begin{align*}
\Delta + c!(d).P & = \text{recv } c d ; [P] & \text{if } c : *! [S] \in \Delta \\
\end{align*}
\]

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

Weakening (introducing either \( c : \text{end} \) or \( c : *! [S] \) is encoded via subeffecting: if \([\Gamma; \Delta] P \equiv [\Gamma] M : \text{unit}, [\Delta] \) then \([\Gamma] \equiv [\Gamma] \text{ unit } \) \( [\Delta] \equiv [\Delta] \) \( c : \text{end} \) (and similarly for output).

Value encoding is fairly direct as PCF has the same constructors as the session calculus (modulo pred): \([\text{unit}]) = \text{unit}, [\text{[V]}]) = \text{[V]} \), \( [0] = 0 \), \([\text{[pred } V]\) = \text{case } \( V \) of \( 0 \rightarrow 0, (\text{[zero } x]\) \rightarrow x \), and \( v \equiv v \) 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).

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

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

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

\[ (\beta \Rightarrow) \quad [\text{T send } c \text{V } v \text{recv } c \text{V }], s \Rightarrow [\text{T } \text{unit } [V], s] \]

\[ (\text{ch}\beta \Rightarrow) \quad [\text{T chSend } c \text{ d } \text{chRec } c \text{ V }], s \Rightarrow [\text{T } \text{unit } [\text{ch } c \text{ d } \text{v}], s] \]

\[ (\text{new} \Rightarrow) \quad [\text{M } c/c, \text{ V }], s \Rightarrow [\text{new } c/c, \text{ V }], s] \]

\[ (\equiv \Rightarrow) \quad M \equiv N \Rightarrow [\text{M } N ], s \Rightarrow [\text{N } N ], s] \]

where \( \equiv \) is the union of two finite maps with a left-bias, \( c \rightarrow [c \rightarrow t] \) \( c \rightarrow s \) is only the pure \( \beta \)-reductions (from Section 2.1) are allowed. A similar rule to \( (\text{ch}\beta \Rightarrow) \) is provided where \( \text{send } \) replaces \( \text{chSend} \).

Let \( \Delta \) be closed if \( \forall c \in \text{dom } \Delta \) then \( [c] \in \text{dom } \Delta \). 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 \( [\Gamma; \Delta] P \) where \( \Delta \) is closed and balanced, and \( \text{wf}(\Delta, s) \), then \([P] \theta, s \equiv \text{new } (\text{N}, s') \equiv [\text{P } \theta, s] \Rightarrow (\text{N}, s') \) where \( \theta \) is a substitution of free channel variables for channel values of the corresponding name in the store \( s \).

Further, \([P] \theta, s \equiv (\text{N}, s') \Rightarrow (\text{M}, s''), ([P] \theta, s) \Rightarrow (\text{M}, s'') \land (\text{N}, s') \Rightarrow (\text{M}, s'') \) and \( \land (\text{N}, s') \Rightarrow (\text{M}, s'') \) holds. Thus, under the image of our encoding and session typing, stable reductions (synchrony) and the operational semantics coincide.

Theorem 5 (Operational correspondence). Then \( \forall T, \Delta, P \).

- (sound) \( \forall Q, \Gamma; \Delta \vdash P \Rightarrow Q \land \text{wf}(\Delta, s) \Rightarrow \exists s'. M, \theta, [P] \theta, s \Rightarrow (\text{M}, s') \land M \equiv [Q] \)

- (complete) \( \forall s', \Gamma; \Delta \vdash P \Rightarrow [P] \theta, s \Rightarrow (\text{M}, s') \land \Delta \land \text{balanced } \Rightarrow \exists Q, P \Rightarrow Q \land \text{M } [Q] \)

For soundness, session calculus \( \beta \)-reductions correspond to \((\beta \Rightarrow)\) and \((\text{ch}\beta \Rightarrow)\), combined with \((\equiv \Rightarrow)\) for replication, branching, and selection. For \( (\Rightarrow) \) is used. Reduction under new corresponds to (new). Reduction under parallel composition and extension of reduction along structural congruence corresponds to \((\equiv \Rightarrow)\). Completeness similarly relates stable and session calculus reductions.
class Effect (m :: ef -> * -> *) where
  type Unit m :: ef
  type Plus m (f :: ef) (g :: ef) :: ef
  return :: a -> m (Unit m) a
  (>>>) :: m f a -> (a -> m g b) -> m (Plus m f g) b

instance Effect Process where
  type Plus Process f g = SeqUnion f g
  type Unit Process = '[]

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

Figure 4. Effect-graded monad and Process instances

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

8. Implementation

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

Effect systems in Haskell

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

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

$$\Gamma \vdash e_1 :: m F \sigma \quad \Gamma, x : \sigma \vdash e_2 :: m G \tau \quad \Gamma \vdash (do \ x \leftarrow e_1; e_2) :: (m (\text{Plus} m F G)) \tau$$

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

Session effects

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

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

The first parameter s is a type-level finite map modelling an environ-
ment of session type information. This is of the form $^\tau [c :-> s, d :-> t, ...]$ describing an environment where a channel
c has session type s, a channel d has session type t, and so on. Session environments are composed sequentially via the SeqUnion

type-level function which models $\cdot$ from Section 7.

Session effects are modelled by the Session data type:

```haskell
data Session = forall a . a :! Session -- send
              | forall a . a :? Session -- receive
              | Session :+: Session -- alternation
              | Bal Session | End -- ‘balanced’ & end
              | Fix Session Session -- $\cdot$ prefix
```

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

```haskell
send :: Chan c -> t -> Process ’c ::-> t :! End ()
recv :: Chan c -> Process ’c ::-> t :? End t
```

where Chan c is a channel named c (a type-level symbol). Chan-
nels 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 dual-
ity, 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:

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

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

```haskell
chRecv :: Chan c ->
  Process ’c ::-> (Delg (e :? d) :) ? End
  (Chan d -> Process e t) -> Process (e :: Channel d) t
```

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

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

Example 5. The following simple example corresponds to a
session calculus term $\nu c.(\nu d.(c!?(d.P))) \mid \tau?(x).x!(Ping)$:

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

server c = do { k <- chRecv c; k \(\langle x -> send x Ping \rangle)
  process = new \(\langle c’, c \rangle -> (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 ’(Chan "c" ::> (Delg (Mag :! 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 \( E = \{ (rd \rho \tau, wr \rho \tau, alloc \rho \tau \mid \forall \rho, \tau \} \), where \( \rho \) are regions, with \((F, U, \emptyset), \oplus = U, \) and \( F^* = F \). We add reference types \( ref, \tau \) tagged with their region and type, and constants:

\[
\text{get} : ref, \tau \to (rd \rho \tau) \to \tau, \quad \text{put} : ref, \tau \to (wr \rho \tau) \to \tau, \quad \text{unit} : \tau \to \tau
\]

The idea behind the encoding is that each mutable store has its own handler. Thus, when \text{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 state handler for the encoding) forwards requests to the state handlers by means of the reference channels. The central handler is defined as:

\[
s?b(c).c(\tau).c \to \{ alloc : \tilde{H}(c), \quad act : e\tilde{c}(r).c \to \{ rd : \tau \to rd.e(x).c(x).\tilde{H}(e(c)), \quad wr : \tau \to wr.e\tilde{c}(y).r(y).\tilde{H}(e(c)) \}\}
\]

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

\[
\{ new \}_{\tau}^{\rho \nu \alpha} = \nu c.q.ea.(\{ M \}_{\rho \nu \alpha}^{\tau} \tilde{H}(x).Var(\tau, x) \mid r!e(x) \} e\tilde{a}(c).c(x) \cdot \oplus < alloc \cdot e\tilde{a}(c))
\]

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

\[
\{ get \}_{\tau}^{\rho \nu \alpha} = \nu q.ea.(\{ M \}_{\rho \nu \alpha}^{\tau} \tilde{H}(x).Var(\tau, x) \mid r!e(x) \} e\tilde{a}(c).c(x) \cdot \oplus < act \cdot e\tilde{a}(c))
\]

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

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

\[
c : \mu \alpha \cdot \nu y.p.[\tilde{H}(y)] \cdot \{ alloc : \alpha, \quad act : \forall \tau. x\tilde{a}(\tau).c(x) \cdot \oplus < [ref, \tau].\tilde{a}(\tau) \cdot [rd][\tilde{a}(\tau).c(x)] \}
\]

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

\[
([rd \rho \tau])F = [\rho] \cdot \oplus \cdot [act \cdot [ref, \tau] \cdot [rd \cdot [\tilde{a}(\tau).c(x)] \cdot [F]]]
\]

\[
([wr \rho \tau])F = [\rho] \cdot \oplus \cdot [act \cdot [ref, \tau] \cdot [wr \cdot [\tilde{a}(\tau).c(x)] \cdot [F]]]
\]

\[
([alloc \rho \tau])F = [\rho] \cdot \oplus \cdot [alloc \cdot [F]]
\]

where \([ref, \tau] = \mu \alpha \cdot \nu y.p.[\tilde{H}(y)] \cdot \{ alloc : \alpha, \quad act : \forall \tau. x\tilde{a}(\tau).c(x) \cdot \oplus < [ref, \tau].\tilde{a}(\tau) \cdot [rd][\tilde{a}(\tau).c(x)] \}, \text{i.e., the type of an effect channel for interacting with a handler.}

9.2 Monadic metalanguage for effects

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

\[
\Gamma \vdash M : T \tau \quad \Gamma \vdash x : \sigma \vdash N : T(\sigma \tau) \quad \Gamma \vdash (M \uplus N) : (T \tau)
\]

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

Our encoding from FPCF to the session calculus can be re-worked for metaFPCF. Since the monad language 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 (\text{let} and \text{unit}) above. The encoding of (let) above would be the same as the encoding for \text{let} \( x = M \in N \) shown at the start of Section 3. The encoding for (unit) is similar to values in our calculus (variables and constants) with \( \langle (M) \rangle \cdot \rho^{\tau} = ei?\langle c \rangle. \forall \tau \langle c \rangle \mid [M] \), where \( [M] \) is any other (sound) typed encoding of PCF, perhaps building on the many possible encodings in the literature (e.g., [33, 49, 51], with various trade-offs).

However, we cannot take one of these existing pure typed-encodings and use it unmodified in combination with the encoding of (let) and (unit). Consider a pure encoding of the \( \lambda \)-calculus part of metaFPCF via \([-]_{\cdot}^{\cdot} \) defined for functions as:

\[
[\lambda x. M]_{\rho \nu \alpha} = \nu d. rd!d!\tilde{d}.(p, q).p?d(x).[M]_{\rho d}
\]

This is the same as in Section 3, but with the effect channel carriers erased. The interpretation of the monadic fragment is written \( \langle (\cdot) \cdot \rangle^{\cdot} \). Consider then the following encoding of a term:

\[
\text{let } f = \langle \lambda x. \text{put } x \rangle ; f \cdot 0^{\rho \nu \alpha} = \nu v.q.ei?\langle c \rangle. \forall \tau \langle c \rangle \mid

\nu y.q(\tilde{F}).y?\langle p, r \rangle .p?d(x) .[\text{put } x]_{\rho \nu \alpha} \mid \forall \tau \langle F \rangle . f \cdot 0^{\rho \nu \alpha}
\]

Whilst the function body is effectful, the encoding here tries to apply the pure embedding to \text{put}. However \text{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 \( T_{\alpha}A \) 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].

\[1\] Filinski calls this Effect-PCF [18], but this is not be confused with FPCF.
sion of the session calculus (without \(*\)!/*? types but only recursive types) [42]. This paper greatly expanded [42], extending the encoding to the higher-order setting of PCF with non-sequential control flow, and considering the reverse encoding and implementation.

**Communication effects** Effect systems have previously been used to describe communication effects for CML [28, 38]. These e.g. system can be translated to our of FPCF 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 \(\rho_1[\tau_1] | \rho_2[\tau_2] | \text{spawn}\{E\}\) 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).

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 lack of congruence under IO-subtyping can correct 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 \(\pi\)-calculi have investigated correctness of encodings in order to examine semantic consequences of proposed typing systems. Our work follows this tradition, studying properties of typed calculi and their encodings.

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

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

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