Higher-Order Processes, Functions, and Sessions: A Monadic Integration

Bernardo Toninho[1,2], Luis Caires[2], and Frank Pfenning[1]

[1] Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA
[2] CITI and Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Lisboa, Portugal

Abstract. In prior research we have developed a Curry-Howard interpretation of linear sequent calculus as session-typed processes. In this paper we uniformly integrate this computational interpretation in a functional language via a linear contextual monad that isolates session-based concurrency. Monadic values are open process expressions and are first class objects in the language, thus providing a logical foundation for higher-order session typed processes. We illustrate how the combined use of the monad and recursive types allows us to cleanly write a rich variety of concurrent programs, including higher-order programs that communicate processes. We show the standard metatheoretic result of type preservation, as well as a global progress theorem, which to the best of our knowledge, is new in the higher-order session typed setting.

1 Introduction

In prior work, we have developed a Curry-Howard interpretation of an intuitionistic linear sequent calculus, where linear propositions correspond to session types [11], sequent proofs to process expressions, and cut reduction to synchronous concurrent computation [4]. This π-calculus based system supports input and output of channels along channels, choice and selection, replicated input, generation of new channels, and message forwarding. Its logical origin led to straightforward generalizations to support data input and output [19] as well as polymorphism [20, 6], incorporated as type input and output. This leaves open the question of how to fully and uniformly incorporate the system into a complete functional calculus to support higher-order, message-passing concurrent computation. In this paper we make a proposal for such an integration and explore its expressive power. We feel that the latter is particularly important, since it is not a priori clear how significantly our session-based (typed) communication restricts the π-calculus, or how easy it is to fully combine functional and concurrent computation while preserving the ability to reason about programs in the two paradigms.

Besides all the constructs of the π-calculus mentioned above, our language includes recursive types and, most importantly, a *contextual monad* to encapsulate open concurrent computations, which can be passed in functional computation but also *communicated* between processes in the style of *higher-order processes*, providing a uniform symmetric integration of both higher-order functions and processes. We allow the construction of recursive processes, which is a common motif in applications. As the examples demonstrate, the various features combine smoothly, allowing concise implementations of diverse examples such as streams and stream transducers, higher-order programs with process passing, polymorphic stacks and process networks for binary counting, among others we omit due to length constraints.

The rest of the paper is organized as follows: Section 2 sets up the necessary background for the presentation of our language. Sections 3.1 through 3.9 present the constructs of our language, intermixed with examples illustrating some of the key features of the constructs. Section 4 details two extended examples: a process implementation of stacks and a process network implementation of a binary counter. Section 5 discusses the metatheory of our language and we present some concluding remarks in Section 6.

2 Processes, Session Types and Functional Computation

In this section we introduce the preliminaries and notation necessary for the introduction of our language. We begin with the notion of session-typed process. We conceive of a process P as *offering* a specified service A along a channel c. The *session type* A prescribes the communication pattern along channel a. We have written this as $P :: c : A$. The process P may *use* services offered by other processes, each with their specified session types, leading to a *linear sequent*:

$$c_1{:}A_1, \ldots, c_n{:}A_n \vdash P :: c{:}A$$

where each of the channels c_i must be used linearly in P and in accordance with its session type A_i. We will abbreviate linear channel contexts with Δ. Note that the order in which the channels are listed is irrelevant, and they can be renamed consistently in the whole sequence as long as all channel names remain distinct. As we will see, to cover the full generality of our language, our typing judgment must account not only for linear channels Δ, but also shared channels which we maintain in a context Γ, and (functional) variables, which we maintain in a context Ψ, thus making our process typing judgment: $\Psi; \Gamma; \Delta \vdash P :: c{:}A$.

Our goal is to combine session typed processes and functional computation, to enable potentially sophisticated reasoning about concurrent programs by exploiting our Curry-Howard foundations. One of the first issues that arises is how to treat the channels c and c_i of the process typing judgment: One solution is to map channels to ordinary functional variables, forcing the entire language to be linear, which is a significant departure from typical approaches. Moreover, even if linear variables are supported, it is unclear how one should restrict their occurrences so they are properly localized with respect to the structure of running processes.

Our solution is to encapsulate processes in a *contextual monad* so that each process is bundled with all channels (both linear and shared – the latter we cover in Section 3.9) it uses and the one that it offers. This is a linear counterpart to the contextual comonad presented in [15].

In our presentation in the following sections, we specify execution of concurrent programs in the form of a *substructural operational semantics* [17], for convenience of presentation. We rely on the following predicates: the linear proposition $\mathsf{exec}\,P$ denotes the state of a linear process expression P; $!\mathsf{exec}\,P$ denotes the state of a persistent process (which must always be a replicating input); and $!\mathsf{eval}\,M\,V$ expresses that the functional term M evaluates to value V without using linear resources.

The rules that make up our substructural operational semantics, for those unfamiliar with this style, can be seen as a form of multiset rewrite rules [7] where the pattern to the left of the $\multimap$ arrow describes a state which is consumed and transformed into the one to the right. Existentials are used to generate names. Names or predicates marked with ! are not linear and thus not consumed as part of the rewrite (aptly modeling replication). The use of connectives from linear logic in this style of presentation, namely $\multimap$ to denote the state transformation, $\otimes$ to combine linear propositions from the context, and ! to denote persistence should not be confused with our session type constructors.

3 Combining Sessions and Functions

In this section we provide first an overview of the constructs of our language and then some details on each. The types of the language are separated into a functional part and a concurrent part, which are mutually dependent on each other. In types, we refer to functional type variables t, process type variables X, labels l_j, and channel names a. We briefly note the meaning of the each session type from the perspective of a provider.

$\tau, \sigma ::= \tau \to \sigma \mid \ldots \mid \forall t.\, \tau \mid \mu t.\, \tau \mid t$	(ordinary functional types)
$\quad\mid\ \{a{:}A \leftarrow \overline{a_i{:}A_i}\}$	process offering A along channel a, using channels a_i offering A_i

A, B, C	$::= \tau \supset A$	input value of type τ and continue as A
	$\mid\ \tau \wedge A$	output value of type τ and continue as A
	$\mid\ A \multimap B$	input channel of type A and continue as B
	$\mid\ A \otimes B$	output fresh channel of type A and continue as B
	$\mid\ \mathbf{1}$	terminate
	$\mid\ \&\{\overline{l_j : A_j}\}$	offer choice between l_j and continue as A_j
	$\mid\ \oplus\{\overline{l_j : A_j}\}$	provide one of the l_j and continue as A_j
	$\mid\ !A$	provide replicable service A
	$\mid\ \mu X.A \mid X$	recursive process type
a	$::= c \mid\ !u$	linear and shared channels

The functional types consist of functions types, polymorphic and recursive types and some additional standard constructs such as base types, products and sums that we have omitted for brevity. The novelty here is the contextual monadic type $\{a{:}A \leftarrow \overline{a_i{:}A_i}\}$, denoting the type of a process expression offering session A along channel a, using the channels a_i at types A_i. Channels a can either be linear (denoted by c or d) or shared (denoted by $!u$ or $!v$).

At the level of session types we have value input $\tau \supset A$ and value output $\tau \wedge A$, referring back to the functional layer. We also have recursive process types, absent from our prior work, which allows us to write some interesting concurrent programs. Next we summarize the terms M, N and process expressions P, Q of the language. Many of the process expressions have a continuation, which is separated from the first action by a semicolon (';'). To explicate the binding structure we have indicated the scope of variables that are bound using subscripts. An overlined expression abbreviates an indexed sequence.

M, N	$::= \lambda x{:}\tau.\, M_x \mid M\, N \mid \texttt{fix}\, x.\, M_x \mid \ldots$	(usual functional constructs)
	$\mid\ a \leftarrow \{P_{a,\overline{a_i}}\} \leftarrow a_1, \ldots, a_n$	process providing a, using $a_1, \ldots, a_n$
P, Q	$::= a \leftarrow M \leftarrow a_1, \ldots, a_n; P_a$	compose process computed by M in parallel with P_a, communicating along fresh channel a
	$\mid\ x \leftarrow \texttt{input}\ c; Q_x$	input a value x along channel c
	$\mid\ \_ \leftarrow \texttt{output}\ c\ M; P$	output value of M along channel c
	$\mid\ d \leftarrow \texttt{input}\ c; Q_d$	input channel d along channel c
	$\mid\ \_ \leftarrow \texttt{output}\ c\ (d \leftarrow P_d); Q$	output a fresh channel d along c
	$\mid\ \texttt{close}\ c$	close channel c and terminate
	$\mid\ \_ \leftarrow \texttt{wait}\ c; P$	wait for closure of c
	$\mid\ \texttt{output}\ c\ !(d \leftarrow P_d)$	output a replicable d along c and terminate
	$\mid\ !u \leftarrow \texttt{input}\ c; Q_{!u}$	input shared channel u along c
	$\mid\ \texttt{case}\ c\ \texttt{of}\ \overline{l_j \Rightarrow P_j}$	branch on selection of l_j along c
	$\mid\ \_ \leftarrow c.l_j; P$	select label l_j along c
	$\mid\ c \leftarrow \texttt{copy}\ !u; P_c$	spawn a copy of $!u$ along c
	$\mid\ \texttt{fwd}\ c_1\ c_2$	forward between c_1 and c_2

The functional part of the language contains ordinary λ-abstraction, application, recursion, and the usual constructors for sum, product, and recursive types, omitted here for brevity. The main construct of interest is the internalization of process expressions through the contextual monadic construct $a \leftarrow \{P\} \leftarrow a_1, \ldots a_n$, denoting a process P using channels a_i to provide along a.

Among the process expressions, the analogue of the monadic *bind* construct is $a \leftarrow M \leftarrow a_1, \ldots, a_n; P_a$. It denotes the composition of the monadic object M (using channels a_i), spawning a new process that provides along a fresh channel for a, that will run in parallel with P_a. Typing enforces that if a is shared, all channels a_i must also be shared (otherwise we could violate linearity).

Using linear and persistent composition of monadic objects, we subsume direct process composition and provide a more uniform way of integrating composition of processes and functional computation. Monadic composition ultimately reduces to ordinary process composition during the computation.

Since we only provide a fixpoint operator at the functional level, writing recursive processes can only be done by writing a recursive function that returns an object of monadic type. We will see this pattern in our examples.

We write $\Psi = (x_1{:}\tau_1, \ldots, x_n{:}\tau_n)$ for the context declaring ordinary (functional) variables, $\Delta = (c_1{:}A_i, \ldots, c_n{:}A_n)$ and $\Gamma = (!u_1{:}B_1, \ldots, !u_n{:}B_n)$ for linear and shared channels, respectively. The order of declarations in all three forms of contexts is irrelevant, but all variables or channel names must be distinct.

$\Psi \Vdash M : \tau$	term M has type τ
$\Psi; \Gamma; \Delta \vdash P :: c{:}A$	process P offers A along c
$\Delta = \mathsf{lin}(\overline{a_i{:}A_i})$	Δ consists of the linear channels c_i in $\overline{a_i{:}A_i}$
$\Gamma = \mathsf{shd}(\overline{a_i{:}A_i})$	Γ consists of the shared channels $!u_i$ in $\overline{a_i{:}A_i}$

3.1 The Contextual Monad

We first detail our contextual monad, for now restricted to offering a service of type A along a *linear* channel c. It is embedded in the functional language with type $\{c{:}A \leftarrow \overline{a_i{:}A_i}\}$ and value constructor $c \leftarrow \{P_{c,\overline{a_i}}\} \leftarrow a_1, \ldots, a_n$.

A monadic value denotes a runnable process offering along channel c and using channels $a_1, \ldots, a_n$, serving both as a way of referring to processes in the functional layer and as a way of communicating processes in the process layer. The typing rule for the monad is:

$$\frac{\Delta = \mathsf{lin}(a_i{:}A_i) \quad \Gamma = \mathsf{shd}(a_i{:}A_i) \quad \Psi; \Gamma; \Delta \vdash P :: c{:}A}{\Psi \Vdash c \leftarrow \{P_{c,\overline{a_i}}\} \leftarrow \overline{a_i{:}A_i} : \{c : A \leftarrow \overline{a_i{:}A_i}\}} \;\{\,\}I$$

The monadic *bind* operation implements process composition. In the simplest case, $c \leftarrow M; Q_c$ composes the process underlying the monadic value M (which offers along c) with Q_c (which uses c and offers d). More generally, composition can refer to monadic values that use multiple channels: $c \leftarrow M \leftarrow a_1, \ldots, a_n; Q_c$. When writing code we often omit the semicolon, instead writing the continuation starting on the next line. The typing rule for the monadic bind is:

$$\frac{\Delta = \mathsf{lin}(\overline{a_i{:}A_i}) \quad \Gamma \supseteq \mathsf{shd}(\overline{a_i{:}A_i}) \quad \Psi \Vdash \mathtt{M} : \{c{:}A \leftarrow \overline{a_i{:}A_i}\} \quad \Psi; \Gamma; \Delta', c{:}A \vdash Q_c :: d{:}D}{\Psi; \Gamma; \Delta, \Delta' \vdash c \leftarrow M \leftarrow \overline{a_i}; Q_c :: d{:}D} \;\{\,\}E$$

The shared channels need not all be used, because shared channels are not linear. On the other hand, linear names c_i must exactly match all names in Δ, enforcing linearity. The operational semantics for executing a monadic bind are:

$$\begin{aligned} &\mathsf{exec}\,(c \leftarrow M \leftarrow \overline{a_i}\;;\, Q_c) \otimes !\mathsf{eval}\, M\, (c \leftarrow \{P_{c,\overline{a_i}}\} \leftarrow \overline{a_i}) \\ &\quad \multimap \{\exists c'.\, \mathsf{exec}\,(P_{c',\overline{a_i}}) \otimes \mathsf{exec}\,(Q_{c'})\} \end{aligned}$$

Executing a bind evaluates M to a value of the appropriate form, which must contain a process expression P. We then create a fresh channel c' and execute of $P_{c',\overline{a_i}}$ itself, in parallel with $Q_{c'}$. In the value of M, the channels c and $\overline{a_i}$ are all bound names, so we rename them implicitly to match the interface of M in the monadic composition.

3.2 Value Communication ($\wedge$ and $\supset$)

Communicating a *value* of the functional language (as opposed to communicating a channel, which is slightly different, see Section 3.5) is expressed at the type level as $\tau \wedge A$ and $\tau \supset A$, corresponding to offering to send and receive values of type τ, respectively. Note that τ is not a session type, although we can communicate session-typed terms by using a monadic type. The language construct for such an output is $\_ \leftarrow \texttt{output}\ c\ M; P$ with the typing rules:

$$\frac{\Psi \Vdash M : \tau \quad \Psi; \Gamma; \Delta \vdash P :: c : A}{\Psi; \Gamma; \Delta \vdash \_ \leftarrow \texttt{output}\ c\ M\ ;\ P :: c : \tau \wedge A}\ \wedge R$$

$$\frac{\Psi \Vdash M : \tau \quad \Psi\ ;\ \Delta, c{:}A \vdash P :: d : D}{\Psi; \Gamma; \Delta, c{:}\tau \supset A \vdash \_ \leftarrow \texttt{output}\ c\ M\ ;\ P :: d : D}\ \supset L$$

Theoretically, these are just trivial reformulations of the usual rules of the session-based process calculus, for example, as in [5]. We have therefore labeled them with their names from the linear sequent calculus.

When a process in the context provides a value output along a channel c, we can *input* it along c and bind a value variable x, written as $x \leftarrow \texttt{input}\ c\ ;\ Q$. The same construct applies when we wish to define a session that offers to input a value. The typing rules are:

$$\frac{\Psi, x{:}\tau; \Gamma; \Delta, c{:}A \vdash Q_x :: d : D}{\Psi; \Gamma; \Delta, c{:}\tau \wedge A \vdash x \leftarrow \texttt{input}\, c\ ;\ Q_x :: d : D}\ \wedge L$$

$$\frac{\Psi, x{:}\tau\ ;\ \Gamma; \Delta \vdash Q_x :: c : A}{\Psi\ ;\ \Gamma; \Delta \vdash x \leftarrow \texttt{input}\, c\ ;\ Q_x :: c : \tau \supset A}\ \supset R$$

At this point we have *two* typing rules each for `input` and `output`. This is because $\mathsf{input}\, c$ either provides a service along $c : \tau \supset A$ or uses a service offered along $c : \tau \wedge A$, and dually for output. A type-checker can always tell whether a process provides or uses a channel, so there is no ambiguity. The rule that governs the semantics for these constructs is:

$$\begin{array}{l}\mathsf{exec}\,(\_ \leftarrow \mathsf{output}\ c\ M\ ;\ P) \otimes \mathsf{exec}\,(x \leftarrow \mathsf{input}\ c\ ;\ Q_x) \otimes \mathsf{!eval}\, M\, V \\ \quad \multimap \{\mathsf{exec}\,(P) \otimes \mathsf{exec}\,(Q_V)\}\end{array}$$

In accord with the call-by-value semantics of the functional language, the term that is to be output must be reduced to a value V, after which an input and an output can *synchronize*, both continuations proceed and the bound variable x is instantiated with the appropriate value.

3.3 Forwarding and Termination

In the underlying proof theory of the linear sequent calculus, we can satisfy an offer $c{:}A$ by using a channel $d{:}A$ of identical type through the identity rule:

$$\frac{}{\Psi; \Gamma; d{:}A \vdash \texttt{fwd}\, c\, d :: c{:}A}\ id$$

In the monadic formulation it is natural to write d (which is consumed) on the left, and c (which is provided) on the right. Operationally, the construct just forwards inputs or outputs along c to d and vice versa. Note that there is no process continuation here, since the offer of A along c has been satisfied in full by $d{:}A$. It therefore only appears as the last line in a monadic expression. The semantics of forwarding are:

$$\mathsf{exec}\,(\mathsf{fwd}\, c\, d) \multimap \{c = d\}$$

The rule applies a global substitution of d for c in the current context representing the state of all processes. In a spatially distributed situation, this cannot be directly implemented. One strategy is to send d along c, tagged as a forwarded channel. In essence, the process offering along c tells its client that it should now interact with the process offering d and then terminates. For this to work, the client must be able to discriminate such a message. Fortunately, since channels are session-typed and have only two endpoints, this does not require a broadcast or a complex protocol.

The process type $\mathbf{1}$, the multiplicative unit of linear logic, maps to termination. The corresponding process constructor is `close` c with typing rule:

$$\frac{}{\Psi; \Gamma; \cdot \vdash \mathsf{close}\, c :: c : \mathbf{1}}\ \mathbf{1}R$$

According to the rules of linear logic, the linear channel context must be empty. Thus, communication along all channels that a process uses must be properly terminated before the process itself terminates. Conversely, if we are using a channel of type $\mathbf{1}$ we can *wait* for its underlying process to terminate with the *wait* construct: $\_ \leftarrow \texttt{wait}\ c\ ;\ P$.

$$\frac{\Psi; \Gamma; \Delta \vdash P :: d : D}{\Psi; \Gamma; \Delta, c{:}\mathbf{1} \vdash \_ \leftarrow \texttt{wait}\ c\ ;\ P :: d : D}\ \mathbf{1}L$$

The substructural operational semantics rule for these constructs is:

$$\mathsf{exec}\,(\mathsf{close}\, c) \otimes \mathsf{exec}\,(\_ \leftarrow \mathsf{wait}\, c\ ;\ P) \multimap \{\mathsf{exec}\,(P)\}$$

Termination is straightforward. When we wait upon a channel that is being closed (as the name implies, `wait` is blocking – and so is `close`), the two operations are consumed and the continuation is executed.

3.4 Example: Streams

We want to produce an infinite stream of integers, starting at a given number n and counting up. This requires a coinductive type, defined as a recursive type (we distinguish between functional and session type definitions with `type` and `stype`, respectively).

```
stype intStream = int /\ intStream
```

In order to produce such a stream, we write a recursive function producing a process expression:

```
nats : int -> {c:intStream}
c <- nats x =
{ _ <- output c x
  c’ <- nats (x+1)
  fwd c c’ }
```

This an example of a function definition. We take some liberties with the syntax of these definitions for readability. In particular, we list interface channels on the left-hand side of the definition. In this formulation, every recursive call starts a new process with a new channel c'. Both for conciseness of notation and efficiency we provide a short-hand: if a tail-call of the recursive function provides a new channel which is then forwarded to the original offering channel, we can reuse the name directly, making the last line of the function above simply `c <- nats (x+1)`.

It looks as if, for example, calling `nats 0` might get into an infinite loop. However, communication in our language is synchronous, so the output will block until a matching consumer inputs the numbers.

We can now construct a stream transducer. As an example, we write a filter that takes a stream of integers and produces a stream of integers, retaining only those satisfying a given predicate $q : \mathsf{int} \to \mathsf{bool}$:

```
filter : (int -> bool) -> { d:intStream <- c:intStream }
d <- filter q <- c =
{ x <- input c
  case q x
    of true  => _ <- output d x
                d <- filter q <- c
     | false => d <- filter q <- c }
```

The `filter` function is recursive, but not a valid coinductive definition unless we can show that filter will be true for infinitely many elements of the stream.

3.5 Linear Channel Communication ($\otimes$ and $\multimap$)

An essential aspect of the π-calculus is the ability to pass channels among processes. This operation belongs to the process layer, since the functional layer can not track the proper linear use of such channels. In a session-typed system we enable processes to send and receive *fresh* communication channels, along which some particular session will be carried out. The types that capture this behavior are $A \otimes B$, denoting a channel which offers to output a fresh channel of type A and continue as B; and $A \multimap B$, which is the type for a channel that offers to input a fresh channel of type A in order to provide a continuation of type B (we will use `*` for $\otimes$ when writing programs). The programming construct that achieves this is: $\_ \leftarrow \texttt{output}\ c\ (d \leftarrow P)\ ;\ Q$ which outputs a fresh channel along c and spawns process P which offers some behavior along the fresh channel (bound in P as d). All available channels will be used in exactly one of the two processes P and Q. The typing rules are:

$$\frac{\Psi;\Gamma;\Delta \vdash P_d :: d : A \quad \Psi;\Gamma;\Delta' \vdash Q :: c : B}{\Psi;\Gamma;\Delta,\Delta' \vdash \_ \leftarrow \texttt{output}\ c\ (d \leftarrow P_d)\ ;\ Q :: c : A \otimes B}\ \otimes\mathsf{R}$$

$$\frac{\Psi;\Gamma;\Delta \vdash P_d :: d : A \quad \Psi;\Gamma;\Delta', c{:}B \vdash Q :: e : E}{\Psi;\Gamma;\Delta,\Delta', c{:}A \multimap B \vdash \_ \leftarrow \texttt{output}\ c\ (d \leftarrow P_d)\ ;\ Q :: e : E}\ \multimap\mathsf{L}$$

Two rules apply for this form of output: offering an output and interacting with the environment that contains a session of $\multimap$ type. Note how in both rules the left premise ensures that process P indeed provides A along d, whereas the right premise types the continuation, where c is now offered (resp. used) as B.

Correspondingly, the construct to input fresh channels is $d \leftarrow \texttt{input}\ c\ ;\ R_d$.

$$\frac{\Psi;\Gamma;\Delta, d{:}A, c{:}B \vdash R_d :: e : E}{\Psi;\Gamma;\Delta, c{:}A \otimes B \vdash d \leftarrow \texttt{input}\ c\ ;\ R_d :: e : E}\ \otimes\mathsf{L}$$

$$\frac{\Psi;\Gamma;\Delta, d{:}A \vdash R_d :: c : B}{\Psi;\Gamma;\Delta \vdash d \leftarrow \texttt{input}\ c\ ;\ R_d :: c : A \multimap B}\ \multimap\mathsf{R}$$

The semantics for these constructions is defined as:

$$\begin{array}{l}\mathsf{exec}\,(\_ \leftarrow \mathsf{output}\,c\,(d \leftarrow P_d)\ ;\ Q) \otimes \mathsf{exec}\,(d \leftarrow \mathsf{input}\,c\ ;\ R_d) \\ \quad \multimap \{\exists d'.\, \mathsf{exec}\,(P_{d'}) \otimes \mathsf{exec}\,(Q) \otimes \mathsf{exec}\,(R_{d'})\}\end{array}$$

When an input and an output along the same channel meet, a fresh channel d' is generated and passed to the continuation of the input and the process Q which is spawned and now offers along that channel, resulting in three parallel processes: the continuation of the input $R_{d'}$, the offering process $P_{d'}$ and the continuation of the output Q (where d' cannot occur by construction).

3.6 Choice and Branching ($\&$ and $\oplus$)

A common idiom when writing concurrent programs is to offer alternative behavior, where a client selects which behavior the server executes. In our system this is embodied by the labelled choice type $\&\{l_1{:}A_1, \ldots, l_k{:}A_k\}$, where the l_i are labels allowing other processes to select behaviors. Dually, it is also common for clients to be able to branch on alternative behavior, decided by the server.

From the server perspective, this is usually referred to as internal choice, in opposition to external choice which corresponds to choices made by the client, and is represented by the type $\oplus\{l_1{:}A_1, \ldots, l_k{:}A_k\}$.

To offer a choice to a client along a channel c we use a `case` construct: `case` c `of` $l_1 \Rightarrow P_1, \ldots, \Rightarrow l_k \Rightarrow P_k$ Such a construct waits for a selection of a label l_j on channel c, after which it will continue as the process P_j. Similarly, a client that is interacting with a server that offers an *internal* choice must branch on the possible outcomes of the server choice, which is also represented by the case construct. The typing rules are:

$$\frac{\Psi; \Gamma; \Delta \vdash P_1 :: c : A_1 \quad \ldots \quad \Psi; \Gamma; \Delta \vdash P_k :: c : A_k}{\Psi; \Gamma; \Delta \vdash \texttt{case}\ c\ \texttt{of}\ \overline{l_j \Rightarrow P_j} :: c : \&\{\overline{l_j : A_j}\}}\ \&\mathsf{R}$$

$$\frac{\Psi; \Gamma; \Delta, c{:}A_1 \vdash P_1 :: d : D \quad \ldots \quad \Psi; \Gamma; \Delta, c{:}A_k \vdash P_k :: d : D}{\Psi; \Gamma; \Delta, c{:} \oplus \{\overline{l_j : A_j}\} \vdash \texttt{case}\ c\ \texttt{of}\ \overline{l_j \Rightarrow P_j} :: d : D}\ \oplus\mathsf{L}$$

In linear logic terminology, the types $\&$ and $\oplus$ are additive. This means that the linear channels available to the processes in the premises are the same. Note how, in the first rule, the channel c to the right of the turnstyle arrow is the same in all the premises, denoting that after the selection takes place, the selected behavior will be carried out along the same channel. In the second rule, each branch is typed in a context where the channel c has committed to a choice.

To perform a selection of a label l_j on a channel, or to make a particular internal choice l_j along c, we use the process construct $\_ \leftarrow c.l_j\ ;\ P$. After performing such a selection, c will offer the behavior assigned to l_i. The typing rules for this construct are:

$$\frac{\Psi; \Gamma; \Delta, c{:}A_j \vdash P :: d : D}{\Psi; \Gamma; \Delta, c{:} \& \{\overline{l_j : A_j}\} \vdash \_ \leftarrow c.l_j\ ;\ P :: d : D}\ \&\mathsf{L}$$

$$\frac{\Psi; \Gamma; \Delta \vdash P :: c : A_j}{\Psi; \Gamma; \Delta \vdash \_ \leftarrow c.l_j\ ;\ P :: c : \oplus\{\overline{l_j : A_j}\}}\ \oplus\mathsf{R}$$

Finally, the operational semantics rule is:

$$\mathsf{exec}\,(\_ \leftarrow c.l_j\ ;\ P) \otimes \mathsf{exec}\,(\mathsf{case}\ c\ \mathsf{of}\ \overline{l_j \Rightarrow Q_j}) \multimap \{\mathsf{exec}\,(P) \otimes \mathsf{exec}\,(Q_j)\}$$

Essentially, choices and selection block until both can be found along the same channel, after which the synchronization takes place and the continuation of the selection P and the corresponding selected process Q_i are executed concurrently.

3.7 Example: An App Store

Our contextual monad allows us to write functions that produce processes, but it also enables us to write process expressions that, through communication and composition of monadic values, communicate and execute actual processes. This contrasts with previous work where only purely functional values could be communicated in the functional layer [19] (also in the language of [20], while there is no distinction between functional and process expressions, it is not completely clear how one can in effect communicate suspended processes).

To clarify this, consider an App Store service that sells applications to its customers. These applications are not necessarily functional, in that they may communicate with the outside world. We can model such a service using monadic types as follows, using `Choice {...}` as our concrete syntax for $\&\{\ldots\}$:

```
stype AppStore = Choice {weather: {c:Weather <- d:API, e:GPS } /\ 1
                         travel: {c:Travel <- d:API } /\ 1
                         game: {c:Game <- d:API } /\ 1}
```

The type above describes a simplified App Store service, which offers three different applications to its customers (for simplicity, assume they are free): a weather forecast application, a travel information application, and a game. Upon selection from the client, the store will send to it the corresponding application. All the applications depend on a proprietary API that is not present locally in clients and is accessed remotely. Furthermore, the weather forecast application also makes use of a GPS connection to locate the user. These restrictions and dependencies are made precise by the contextual regions of the monadic types. The code for a client that downloads the weather application and runs it is given below:

```
ActivateGPS : unit -> {g:GPS}
WeatherClient : unit -> {c:Weather <- a:AppStore,d:API}
c <- WeatherClient() <- a:AppStore, d:API =
{ _ <- a.weather
  w <- input a
  _ <- wait a
  g <- ActivateGPS()
  c <- w <- d, g }
```

The client requires an existing connection with the AppStore service and the connection with the API, which we assume is established by some other means. The client then performs the appropriate selection and download from the store. To run the application, it first makes use of a local function that activates its GPS module, supplying a channel handle `g` which is then used to fulfill the required dependencies of the weather application. This simple example shows how cleanly we can integrate *communication* and *execution* of open, process expressions into our functional language. It is straightforward to extend this example to more complex communication interfaces.

3.8 Example: A List Process

We exemplify the usage of branching by defining a type for a process implementation of a list. The process can either behave as the empty list (i.e. offer no behavior) or as a list with a head and a tail, modelled by the output of the head element of the list, followed by the output of a *fresh* channel consisting of the handle to the tail list process. The type employs data polymorphism in the elements maintained in the list. We write `Or {...}` as concrete syntax for $\oplus\{\ldots\}$ and `t => s` as concrete syntax for $\tau \supset A$.

```
stype List t = Or {nil: 1, cons: t => (List t * 1)}
```

We can now define two functions, `Nil` and `Cons`: the first produces a process that corresponds to the empty list and the second, given a value `v` of type `t` will produce a process that expects to interact with a channel `l` denoting a list of `t`'s, such that it will implement, along channel `c`, a new list with `v` as its head.

```
Nil : unit -> {c:List t}
c <- Nil () =
{ _ <- c.nil
  close c
}
```

```
Cons : t -> {c:List t <- l:List t}
c <- Cons v <- l =
{ _ <- c.cons
  _ <- output c v
  _ <- output c (l' <- fwd l l')
  close c }
```

Note that the `Cons` function, after sending the `cons` label along channel `c` and outputting `v`, it will output a fresh channel `l'` that is meant to represent the tail of the list, which is actually present along channel `l`. Thus, the function will also spawn a process that will forward between `l` and `l'`.

3.9 Sharing and Replication (!)

All the process type constructors we have described thus far have been purely linear, in the sense that they represent behavior that must take place exactly once. *Shared channels* $!u$ allow behavior to be replicated.

Shared channels appear in our language in two roles: we can *bind* a monadic expression to a shared channel, provided the monadic expression does not depend on linear channels; and we can *use* a channel of type $!A$, which we make precise shortly. A monadic expression that does not depend on any linear channels can be bound to a persistent channel $!u \leftarrow M \leftarrow !u_1, \ldots, !u_n \ ; P$. where M is an expression of monadic type, conforming to the linearity restriction mentioned above. Type-theoretically, this is reminiscent of the $\mathsf{cut}^!$ principle:

$$\frac{\Gamma \subseteq \overline{!u_i{:}B_i} \quad \Psi \Vdash M : \{!u{:}A \leftarrow \overline{!u_i{:}B_i}\} \quad \Psi \ ; \Gamma, u{:}A; \Delta \vdash Q_{!u} :: c : C}{\Psi; \Gamma; \Delta \vdash !u \leftarrow M \leftarrow \overline{!u_i} \ ; Q_{!u} :: c : C} \{\,\}E^!$$

The idea is that the monadic process underlying M will be *replicated* as many times as uses of u take place. The semantics for this form of bind are:

$$\begin{array}{l} \mathsf{exec}\,(!u \leftarrow M \leftarrow \overline{!u_i} \ ; Q_{!u}) \otimes !\mathsf{eval}\, M\,(c \leftarrow \{P_{c,\overline{!u_i}}\} \leftarrow \overline{!u_i}) \\ \quad \multimap \{\exists u.\, !\mathsf{exec}\,(c \leftarrow \mathsf{input}\,!u\,; P_{c,\overline{!u_i}}) \otimes \mathsf{exec}\,(Q_{!u})\} \end{array}$$

A persistent bind forces the evaluation of the monadic object and then spawns a replicating process that will input on the generated (shared) channel u a fresh channel c. Each such channel c well be used for communication between replicas of P and its clients. This process is spawned in parallel with the continuation $Q_{!u}$ which can trigger replications of P as needed.

Using a shared channel is accomplished by $c \leftarrow \texttt{copy}\ !u\ ;\ P$. The `copy` construct triggers the creation of a new process that implements the behavior ascribed to $!u$ along a fresh *linear* channel that is bound to c. The typing rule for `copy` explicates this concept:

$$\frac{\Psi; \Gamma, u{:}A; \Delta, c{:}A \vdash Q_c :: d : D}{\Psi; \Gamma, u{:}A; \Delta \vdash c \leftarrow \texttt{copy}\ !u\ ;\ Q_c :: d : D}\ \textsf{copy}$$

And the semantics are:

$$!\textsf{exec}\,(d \leftarrow \textsf{input}\,!u\ ;\ P_d) \otimes \textsf{exec}\,(c \leftarrow \textsf{copy}\,!u\ ;\ Q_c) \multimap \{\exists c'.\,\textsf{exec}(P_{c'}) \otimes \textsf{exec}(Q_{c'})\}$$

Note that the process performing an input along $!u$ persists, since the proposition $!\textsf{exec}$ is persistent in the metalanguage of SSOS.

We internalize sharing at the process type level as $!A$, the type of a linear channel that can be promoted to a shared channel of type A. The constructs below may appear somewhat complex, but arise entirely from a Curry-Howard interpretation of intuitionistic linear logic. We provide a channel $c{:}!A$ of such a type with the construct $\texttt{output}\ c\ !(d \leftarrow P_d)$. Note that there is no continuation, and the subterm is preceded by a '!'. To use a channel of type $!A$, we input the fresh shared channel $!u \leftarrow \texttt{input}\ c\ ;\ Q_u$.

The idea is that we will output along c a *fresh* channel $!u'$ which will be of a shared nature. It is along $!u'$ that subsequent interactions will take place, and thereafter all communication along c has terminated. The process expression P_d will then implement some behavior that will be *replicated* whenever (and only if) the fresh shared channel u' that was output is used. The typing rules, corresponding to the $!R$ and $!L$ from dual intuitionistic linear logic:

$$\frac{\Psi; \Gamma; \cdot \vdash P_d :: d : A}{\Psi; \Gamma; \cdot \vdash \texttt{output}\ c\ !(d \leftarrow P_d) :: c : !A}\ \textsf{!R} \qquad \frac{\Gamma, u{:}A; \Delta \vdash Q_u :: d : D}{\Gamma; \Delta, c{:}!A \vdash\ !u \leftarrow \texttt{input}\ c\ ;\ Q_u :: d : D}\ \textsf{!L}$$

Note that P_d may not depend on any linear channels, so that it can be replicated as needed. The semantics for these forms of input and output are:

$$\begin{gathered}\textsf{exec}\,(\textsf{output}\,c\,!(d \leftarrow P_d)) \otimes \textsf{exec}\,(!u \leftarrow \textsf{input}\,c\ ;\ Q_{!u}) \\ \multimap \{\exists !u'.\ !\textsf{exec}\,(d \leftarrow \textsf{input}\,!u'\ ;\ P_d) \otimes \textsf{exec}\,(Q_{!u'})\}\end{gathered}$$

4 Extended Examples

In this section we cover two slightly larger examples that showcase some of the expressiveness of our language. We will first show how to define two different implementations of stacks using monadic processes and how our Curry-Howard basis allows for simple and elegant programs. Secondly, we will describe the implementation of a binary counter as a network of communicating processes.

Stacks We begin by defining the type for a stack session type:

```
stype Stack t =  Choice {push: t => Stack t
                         pop: Or {none: unit /\ Stack t
                                  some:t /\ Stack t}
                         dealloc: 1}
```

The `Stack` recursive type denotes a channel along which a process offers the choice between three operations: `push`, which will then expect the input of an element that will be the new top of the stack; `pop`, which outputs either the top element or unit (if the stack is empty); and `dealloc`, which fully deallocates the stack and thus has type **1**.

We present two distinct implementations of type `Stack`: `stack1` makes use of functional lists to maintain the stack; the second implementation `stack2`, more interestingly, uses the list processes from Section 3.8 to implement the stack as a network of communicating processes.

```
stack1 : list t -> {c:Stack t}
c <- stack1 nil =                     | c <- stack1 (v::l) =
{ case c of                             { case c of
  push    => v <- input c                 push    => v' <- input c
             c <- stack1 (v::nil)                    c  <- stack1 (v'::v::l)
  pop     => _ <- c.none                  pop     => _  <- c.some
             _ <- output c ()                        _  <- output c v
             c <- stack1 nil                         c  <- stack1 l
  dealloc => close c }                    dealloc => close c }
```

The code above consists of a function, taking a list and producing a monadic object indexed by the given list. We define the function by branching on the structure of the list, as usual. We can, for instance, create an empty stack by calling `stack1` with the empty list.

As mentioned above, our second implementation makes use of the list processes of Section 3.8. We begin by defining a function `deallocList`, whose purpose is to fully terminate a process network implementing a list. This means recursively consuming the list session and terminating it:

```
deallocList : unit -> {c:1 <- l:List t}
  c <- deallocList () <- l =
  {  case l of
       nil  => _ <- wait l
               close c
       cons => v  <- input l
               l' <- input l
               _  <- wait l
               c  <- deallocList () <- l' }
```

We define our second stack implementation by making use of the `Cons`, `Nil` and `deallocList` functions. The function `stack2` below produces a monadic stack process with an underlying process network implementing the list itself:

```
stack2 : unit -> {c:Stack t <- l:List t}
 c <- stack2 () <- l =
 { case c of
   push    => v  <- input c
              l' <- Cons v <- l
              c  <- stack2 <- l'
   pop     => case l of
              nil  => _  <- wait l
                      _  <- c.none
                      _  <- out c ()
                      l' <- Nil ()
                      c  <- stack2 () <- l'
              cons => v  <- input l
                      l' <- input l
                      _  <- wait l
                      _  <- c.some
                      _  <- out c v
                      c  <- stack2 () <- l'
  dealloc => c <- deallocList () <- l }
```

The monadic process specified above begins by offering the three stack operations: `push`, `pop` and `dealloc`. The first inputs along the stack channel the element that is to be pushed onto the stack and calls on the `Cons` function to produce a monadic process that appends to the list session l the new element, binding the resulting list process to l' and making a recursive call. The `pop` case needs to branch on whether or not the list session l encodes the empty list. If such is the case (`nil`), it waits for the termination of l, signals that the stack is empty and calls upon the `Nil` function to reconstruct an empty list for the recursive call; if not (`cons`), it inputs the element from the list session and the continuation list l'. It then outputs the received element and proceeds recursively. Finally, `deallocList` calls out to the list deallocation function.

Bit Counter Network As above, we begin with the interface type:

```
stype Counter = Choice {inc: Counter
                        val: nat /\ Counter
                        halt: 1}
```

The `Counter` session type provides three operations: an `inc` operation, which increments its internal state; a `val` operation, which just outputs the counter's current value and a `halt` operation which terminates the counter.

One way to implement such a counter is through a network of communicating processes, each storing a single bit of the bit string that encodes the value of the counter. We do this by defining two mutually recursive functions `epsilon` and `bit`. The former encodes the empty bit string, abiding to the counter interface. Notably, in the `inc` branch, a new bit process is spawned with value 1. To do this, we make a recursive call to the epsilon function, bound to channel d, and then simply call the `bit` function with argument 1, also providing it with the channel d. The `bit` function encodes an actual bit element of the bit string. It takes a number as an argument which is the 1 or 0 value of the bit and constructs a process expression that provides the counter interface along channel c by having access to the process encoding the previous bit in the string along channel d.

```
bit : nat -> {c:Counter <- d:Counter}   epsilon : unit -> {c:Counter}
  c <- bit b <- d =                       c <- epsilon () =
  {  case c of                            { case c of
       inc  => case b of                     inc  => d <- epsilon ()
                 0 => c <- bit 1 <- d                c <- bit 1 <- d
                 1 => _ <- d.inc
                      c <- bit 0 <- d
       val  => _ <- d.val                    val  => _ <- output c 0
               n <- input d                          c <- epsilon ()
               _ <- output c (2*n+b)
               c <- bit b <- d
       halt => _ <- d.halt                   halt => close c }
               _ <- wait d
               close c }
```

A bit b outputs the counter value by polling the previous bit for its counter value n and then outputting $2n + b$. This invariant ensures an adequate binary encoding of the counter. Termination triggers the cascade termination of all bits by sending a termination message to the previous bit, waiting on the channel and then terminating. The increment case simply recurses with value 1 if the bit is 0; otherwise it sends an increment message to the previous bit to encode the carry and recurses with value 0.

5 Metatheory

For the functional part of the language, we presuppose a standard call-by-value semantics. We could also use call-by-name, but for communication across channels, especially if distributed, one would not want to pass potentially large computations and the data structures they still rely on. If the functional language is overlaid with a termination checker (for examples, along the lines of Abel's proposal [1]), then the two should semantically coincide in any case. Since this is standard, we focus on the interesting new constructs: the monad, and the process expressions contained in them.

From the perspective of the functional language, an encapsulated process expression is a value and is not executed. Instead, functional programs can be used to construct concurrent programs which can be executed at the top-level, or with a special built-in construct such as *run*, which would have type

```
run : {c:1} -> unit
```

Not accidentally, this is analogous to Haskell's I/O monad [16], even if our language is call-by-value.

We now summarize the expected preservation and progress theorems. In order to state the type preservation theorem we must be able to talk about the types and channels during the execution of the processes, not just for a process expression P. An elegant way to accomplish this is to annotate each $\mathsf{exec}\, P$ with the channel c along which P offers its output and its type A. This exploits the observation that every process offers a service along exactly one channel, and for every channel there is exactly one process providing a service along it. This extended form is written $\mathsf{exec}\, P\, c\, A$. The rules given above can be updated in a straightforward fashion, and the original rules can be recovered by erasure. The annotations fix the role of every channel in a communication as either offered or used, and we can check if the whole process state Ω is well-typed according to a signature of (linear and shared) channels Σ. We write $\models (\Sigma \,;\, \Omega) :: c_0 : \mathbf{1}$ if process state Ω uses channels in Σ accordingly and offers $\mathbf{1}$ along an initial channel c_0 that is offered but not used anywhere. Initially, we have a closed process expression P_0 and $\models (\cdot \,;\, \mathsf{exec}\, P_0\, c_0\, \mathbf{1}) :: c_0 : \mathbf{1}$. Overall, a pair consisting of the currently available channels and the process state evolves via multiset rewriting [7] to another pair, potentially containing new channels and the new process state.

Theorem 5.1 (Type Preservation).

(i) If $\cdot \Vdash M : \tau$ *and* $!\mathsf{eval}\, M\, V$ *then* V *is a value and* $\cdot \Vdash V : \tau$.
(ii) If $\models (\Sigma \,;\, \Omega) :: c_0 : \mathbf{1}$ *and* $(\Sigma \,;\, \Omega) \longrightarrow^* (\Sigma' \,;\, \Omega')$ *then* $\models (\Sigma' \,;\, \Omega') :: c_0 : \mathbf{1}$.

Type preservation is fairly straightforward to prove, given the strong logical foundations of our language. We require the typical substitution property for the functional portion of the language. As for processes, the proof requires us to relate typing derivations of process expressions to typings of the global executing process state. This turns out to be easy, since substructural operational semantics breaks down the global state into its local process expressions.

Theorem 5.2 (Progress). *Assume for every term* M *such that* $\cdot \Vdash M : \tau$ *there exists a value* V *with* $!\mathsf{eval}\, M\, V$*. Then for every well-typed process state* $\models (\Sigma \,;\, \Omega) :: c_0 : \mathbf{1}$*, either* $\Omega = (!\Omega'', \mathsf{exec}\,(\mathsf{close}\, c_0)\, c_0\, \mathbf{1})$ *where* $!\Omega''$ *consists of propositions of the form* $!\mathsf{exec}\, P$*, or* $(\Sigma \,;\, \Omega) \longrightarrow (\Sigma' \,;\, \Omega')$ *for some* Σ' *and* Ω'.

Progress is, as usual, slightly harder to prove. Once we account for the internal transitions of processes and functional evaluation, we note that in a well-typed state Ω, persistent processes (which always perform a replicating input) can never block. Due to linear well-typing of the state, we can therefore restrict attention to the remaining $k+1$ processes that *offer* communication along $k+1$ channels, but *using* only k channels since c_0 does not have a match. Now we perform an induction on k. If P_0 is blocked on c_0, it must have the form stated in the theorem (by inversion on its typing) and we are done. If not, it must be blocked on some other channel, say, c_1. Now the process P_1 offering c_1 is either blocked on c_1, in which case it can communicate with P_0 and we can make a transition, or it must be blocked on some other c_2. We proceed in this way until we must come to P_k, which must be blocked on c_k and can communicate with P_{k-1} since no other linear channel c_{k+1} remains on which it could be blocked.

It is easy to modify the operational semantics to employ a small-step semantics for the functional layer, which ensures progress for the full language without relying on termination of functional computation.

We now return to consideration of what we have called the *linear contextual monad*. In general, a monad consists of a type constructor M supporting two operations usually called *return* and *bind*. The return operation allows for any value in the language to be made into a monadic object, whereas bind is a form of composition. These operations are expected to satisfy certain equational laws. Specifically, return is both a left and right unit for bind, and bind is associative.

Using our monadic introduction and composition constructs, we can reproduce similar laws. First, if we consider the process expression: $c \leftarrow \{c \leftarrow P_c\}; Q_c$, it is straightforward to see, using our semantics, that it behaves as just both P_c and Q_c executing in parallel. This is a form of left identity, and captures the computational effects of binding. Secondly, we can reconstruct a right identity law by observing that the term: $\{c \leftarrow (d \leftarrow M; \texttt{fwd}\ c\ d)\}$ always behaves like M. This is reminiscent of an η-conversion law, but one must note that in the presence of non-termination, evaluating the expression M might not terminate, whereas the monadic expression is always a value. However, any context that uses both expressions will not be able to distinguish them, regardless of non-termination. Finally, it is easy to see that our composition construct is associative.

Taking the category theory perspective, these constructions are somewhat reminiscent of the work on Arrows [12], itself a special case of Relative Monads [2], which are monadic constructions for functors that are not endomorphic. A closer work, also rich in category theoretical foundations is that of Benton [3], where two functors F and G are defined, forming an adjunction between intuitionistic and linear functional calculi. Our construction is in essence a contextual variant of G, albeit with some slight differences, specifically the fact that we employ a let-style elimination and we bridge a functional and a process calculus, instead of two functional calculi.

6 Related Work and Conclusion

Our language is similar to the higher-order session typed calculi of [14]. However, our logical foundation makes the system substantially simpler, and the contextual monad allows for a cleaner integration of higher-order communication, which they accomplish by passing λ-abstractions. Furthermore, we obtain a global progress result, which is not present in [14].

A language with similar goals to ours is Wadler's GV [20], which is itself based on a session-typed functional language created by Gay and Vasconcelos [10]. GV is also a session-typed functional language, essentially consisting of a simply typed, linear λ-calculus extended with primitives for session-typed communication. A point of divergence of GV and our language is that GV is itself linear, whereas we base our functional language in a traditional λ-calculus equipped with a linear contextual monad that isolates communication and linear typing. Naturally, making the whole language linear avoids the need for the monad, since it becomes possible to write functions that, for instance, take a data value and a channel and send the piece of data along the channel (this is so because essentially all terms in GV can be translated to session-typed linear processes). On the other hand, the pervasively concurrent semantics means it is further from a practical integration of concurrency into an existing functional language. Another significant difference between the two approaches is that the underlying type theory of GV is classical, whereas ours is intuitionistic. Monads are intuitionistic in their logical form [9], which therefore makes the intuitionistic form of linear logic a particularly good candidate for a monadic integration of functional and concurrent computation based on a Curry-Howard correspondence. We believe our natural examples demonstrate this clearly. Prior work by Mazurak and Zdancewic [13] indicates that control operators may be a better candidate than a contextual monad for classical linear logic, if the functional character of the underlying language is to be fully preserved.

The language F^* [18] shares similar goals and ideas, but it is aimed at security properties and distributed computation, while we aim at concurrency. Instead of linear types, F^* uses affine types and its concurrency primitives are not based on a Curry-Howard correspondence. The various language levels, including communication, are separated not by a monad but through a complex kinding system that controls their interaction. Our language design aims to be a stepping stone towards full dependent verification (as traditional in type theory) and allowing for dynamic verification. F^* makes several interesting contributions with respect to this tradeoff, in particular the use of value-dependent types.

Finally, there are a number of language features we have given short thrift here in order to concentrate on our essential contributions. One is the possibility of asynchronous communication. Work by DeYoung et al. [8] shows that this is consistent with a Curry-Howard approach although some programs we wrote here (like infinite stream producers) would have to be rewritten to account for the change in the operational semantics. Polymorphism [20, 6] for process expressions is largely orthogonal and manageable as long as types are explicitly passed.

Future Work. Our main goal for future work is the generalization of the system we have presented here to a full dependent type theory that integrates reasoning about both functional and concurrent computation. Dependent types in the purely functional setting are a well understood concept, however the generalization to our language is far from straightforward since we move to a setting where session types can be indexed not only by purely functional terms, but also by session typed processes through the monadic type from the functional language. Similarly, dependent types in the functional layer share this feature. This means that type equality (crucial for type conversion in dependent type theories), which ultimately reduces to term equality, requires a suitable notion of *process equality*. While we obviously want a decidable equality, it is not clear what other criteria this notion of equality should obey. Moreover, reasoning about processes is typically done both inductively and coinductively, so to be able to internalize this reasoning in the language we require a primitive notion of coinductive reasoning, as well as a proper theory of inductive and coinductive definitions applied to our session typed setting. We plan to tackle these challenges in future work.

Acknowledgments. Support for this research was provided by the Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program, under grants SFRH / BD / 33763 / 2009 and INTERFACES NGN-44 / 2009, and CITI; and by the Army Research Office under Award No. W911NF-09-1-0273

References

1. Abel, A.: Type-based termination, inflationary fixed-points, and mixed inductive-coinductive types. In: Proceedings of FICS'2012. pp. 1–11 (2012)
2. Altenkirch, T., Chapman, J., Uustalu, T.: Monads need not be endofunctors. In: FOSSACS. pp. 297–311 (2010)
3. Benton, P.N.: A mixed linear and non-linear logic: Proofs, terms and models (preliminary report). In: Comp. Sci. Log. pp. 121–135. Springer-Verlag (1994)
4. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: 21st Int. Conf. Concur. Theory. pp. 222–236. LNCS 6269 (2010)
5. Caires, L., Pfenning, F., Toninho, B.: Towards concurrent type theory. In: Types in Language Design and Implementation. pp. 1–12 (2012)
6. Caires, L., Pérez, J.A., Pfenning, F., Toninho, B.: Relational parametricity for polymorphic session types. Tech. Rep. CMU-CS-12-108, Carnegie Mellon Univ. (2012)
7. Cervesato, I., Scedrov, A.: Relating state-based and process-based concurrency through linear logic. Information and Computation 207(10), 1044–1077 (2009)
8. DeYoung, H., Caires, L., Pfenning, F., Toninho, B.: Cut reduction in linear logic as asynchronous session-typed communication. In: Computer Science Logic (2012)
9. Fairtlough, M., Mendler, M.: Propositional lax logic. Information and Computation 137(1), 1–33 (Aug 1997)
10. Gay, S., Vasconcelos, V.T.: Linear type theory for asynchronous session types. J. Funct. Programming 20(1), 19–50 (2010)
11. Honda, K.: Types for dyadic interaction. In: 4th Int. Conf. Concur. Theory. pp. 509–523. LNCS 715 (1993)
12. Hughes, J.: Generalising monads to arrows. Sci. of Comp. Prog. 37, 67–111 (1998)
13. Mazurak, K., Zdancewic, S.: Lolliproc: to concurrency from classical linear logic via curry-howard and control. In: ICFP. pp. 39–50 (2010)
14. Mostrous, D., Yoshida, N.: Two session typing systems for higher-order mobile processes. In: TLCA07. pp. 321–335. Springer-Verlag (2007)
15. Nanevski, A., Pfenning, F., Pientka, B.: Contextual modal type theory. Transactions on Computational Logic 9(3) (2008)
16. Peyton Jones, S.L., Wadler, P.: Imperative functional programming. In: Principles of Prog. Lang. pp. 71–84. POPL '93 (1993)
17. Pfenning, F., Simmons, R.J.: Substructural operational semantics as ordered logic programming. In: Logic in Comp. Sci. pp. 101–110 (2009)
18. Swamy, N., Chen, J., Fournet, C., Strub, P.Y., Bhargavan, K., Yang, J.: Secure distributed programming with value-dependent types. In: ICFP. pp. 266–278 (2011)
19. Toninho, B., Caires, L., Pfenning, F.: Dependent session types via intuitionistic linear type theory. In: Prin. Pract. Decl. Program. pp. 161–172 (2011)
20. Wadler, P.: Propositions as sessions. In: ICFP. pp. 273–286 (2012)