Lightweight Functional Session Types

Sam Lindley J. Garrett Morris
The University of Edinburgh
{Sam.Lindley,Garrett.Morris}@ed.ac.uk

Abstract

Session types describe communication protocols, capturing both the type and the order of messages. Recently, we presented a semantics for a core session-typed linear λ-calculus, GV, and proved that it enjoys a number of desirable properties beyond type soundness, including deadlock freedom, determinism (and hence race freedom), and termination.

In this paper, we modularly extend GV with practical features. We begin by introducing FST (System F with Session Types), an extension of GV, with features including polymorphism, row typing (to support extensible records, variants, and session types), and a subkinding system (to integrate linear and unlimited types). FST preserves all of GV's desirable properties. We then consider further extensions of FST with recursion and recursive types, which preserve all of the properties save for termination. Finally, we consider an additional extension of FST with access points (a much more expressive mechanism for initiating communication among threads) in return for giving up deadlock freedom, determinism (and indeed race freedom), and termination.

Building on the formal development, we discuss the design of Session Links, a session-typing extension of the Links web programming language and a full implementation of FST.

1. Introduction

Concurrency has become a critical aspect of modern programs, and thus a central problem in program correctness. Concurrency and communication add a new dimension to assuring program behavior: communication protocols specify not just what form messages between participants must take, but also impose ordering constraints on the transmission of those messages. For example, consider the SMTP protocol. All SMTP messages are strings, so SMTP can be described as the exchange of arbitrary strings. However, such a description includes many violations of the protocol. Describing correct SMTP participants requires capturing two additional aspects. First, individual SMTP messages must be well-formed; for example, each recipient is identified by a message `RCPT TO: [email]` from the client, where `[email]` is a valid email address. Second, messages must be received in the right order: the sender must be identified before the recipients, which in turn must precede the message body.

Honda [13] proposed session types as a means to capture protocols between two participants sharing a communication channel. A session type captures the messages observable on that channel; for example, a simplified version of the client side of the SMTP protocol could be expressed as

$$!\mathit{Sender}.!\mathit{Recipient}.!\mathit{Body}.\mathsf{End}$$

where $!A.S$ denotes sending a value of type A and then continuing as S. Session type systems are necessarily substructural: if programs could duplicate or discard channels, there would be no guarantee that the observable behavior matched the protocol even if the channels were otherwise used according to their types.

Recently we introduced a semantics [20] for GV, a core linear λ-calculus extended with session types due to Wadler [32], and inspired by prior work of Gay and Vasconcelos [11]. We gave direct proofs of deadlock freedom, determinism, and termination for GV. We also gave semantics-preserving translations between GV and Wadler's process calculus CP [32], showing a strong connection between GV's small-step operational semantics and cut elimination in classical linear logic. In this paper, we demonstrate that we can build practical languages based on the primitives and properties we used with GV. We begin with a language, FST, that extends GV with polymorphism and row typing, allowing abstraction over types, record and variant structure, and subkinding, integrating linear and unlimited data types. We show that FST, while more expressive, is still deadlock-free, deterministic, and terminating. We then consider several extensions of FST. Adding recursion and recursive session types permits the definition of long-running services and repeated behavior; while the resulting system is no longer terminating, it preserves deadlock freedom and determinism. Access points provide a more flexible mechanism for session initiation. Introducing access points results in a system that is not deadlock-free, deterministic, or terminating; but it still satisfies subject reduction and a weak form of type progress.

***A First Look at* FST** Before giving a formal account of the syntax and type system of FST, we give a simple example of programming in FST. The SMTP protocol is complex, and a complete description of the protocol would require many details irrelevant to the focus of this paper. Instead, we will use a desktop calculator as a running example. Despite its simplicity, it will motivate many of the features of FST.

We begin with the process that implements the calculator. We specify it as a function of one channel, c, on which it will communicate with a user of the calculator.

$$\begin{aligned} \mathit{calc} = \lambda c.\mathsf{offer}\, c\, \{ & \\ \mathit{Add}\, c \to\ & \mathsf{let}\, \langle x, c\rangle = \mathsf{receive}\, c\, \mathsf{in} \\ & \mathsf{let}\, \langle y, c\rangle = \mathsf{receive}\, c\, \mathsf{in} \\ & \mathsf{send}\, \langle x + y, c\rangle \\ \mathit{Neg}\, c \to\ & \mathsf{let}\, \langle x, c\rangle = \mathsf{receive}\, c\, \mathsf{in} \\ & \mathsf{send}\, \langle -x, c\rangle \} \end{aligned}$$

On receiving a channel c, the function *calc* offers a choice of two behaviors, labeled *Add* and *Neg* on c. In the *Add* case, it then expects to read two values from c and send their sum along c. The *Neg* case is similar. The session type of channel c encodes these interactions, so the type of *calc* is

$$calc : \&\{Add : ?Int.?Int.!Int.\mathsf{End}, Neg : ?Int.!Int.\mathsf{End}\} \to^{\bullet} \langle\rangle$$

where the session type $!T.S$ denotes sending a value of type T followed by behavior S, $?T.S$ denotes reading a value of type T followed by behavior S, and $\&\{\ell : S, \ldots, \ell_n : S_n\}$ denotes offering an n-ary choice, with the behavior of the i^{th} branch given by S_i.

Next, we consider clients of the calculator process. For example, here is a function that uses the calculator service:

$$user = \lambda c.\ \mathsf{let}\ c = \mathsf{select}\ Add\ c\ \mathsf{in}\ \mathsf{receive}\ (\mathsf{send}\ \langle 6, \mathsf{send}\ \langle 7, c\rangle\rangle)$$

Like *calc*, *user* is specified as a function of a channel on which it communicates with the calculator. It begins by selecting the *Add* behavior; this is the dual of the choice offered by *calc*. The remainder of its behavior is unsurprising. We could give the channel a type dual to that provided by the calculator:

$$user : \oplus\{Add : !Int.!Int.?Int.\mathsf{End}, Neg : !Int.?Int.\mathsf{End}\} \to^{\bullet} Int$$

However, this over-specifies the behavior of *user* by specifying the behavior of the *Neg* branch, which is unused by *user*. In FST, we can use row polymorphism to abstract over the irrelevant labels in a choice, as follows:

$$user : \forall \rho^{\circ,\pi}.\ \oplus\{Add : !Int.!Int.?Int.\mathsf{End}; \rho\} \to^{\bullet} Int$$

This type denotes that *user* can accept an argument that offers arbitrary choices, so long as it includes the *Add* branch with suitable behavior. FST includes explicit type abstractions and type annotations on bound variables; we will omit both in these examples to improve their legibility. Session Links, our concrete implementation of FST, is able to reconstruct the types and type abstractions omitted in the examples using a fairly standard Hindley-Milner-style type inference algorithm.

The fork primitive creates a new child thread and a channel through which it can communicate with its parent thread. We can compose the calculator and the user together as follows

$$\mathsf{let}\ c = \mathsf{fork}\ calc\ \mathsf{in}\ user\ c$$

yielding the number 13.

Contributions. The paper proceeds as follows:

- Section 2 introduces FST, a linear variant of System F, incorporating polymorphism, row-typing, subkinding, and session types.
- Section 3 gives a small-step semantics for FST, incorporating explicit buffers to provide asynchronous communication primitives. We characterize deadlock and show that well-typed FST programs enjoy type soundness, deadlock freedom, progress, determinism, and termination.
- Section 4 explores extensions of our core calculus with recursion, recursive types, and access points. We demonstrate the expressivity of access points, giving encodings of state cells, nondeterministic choice, and recursion. We argue that adding recursion and recursive types preserves type soundness, determinism, and deadlock freedom, and that even in the presence of access points we can show a relaxed form of progress.
- Section 5 describes Session Links, a practical implementation of FST in Links, a functional language for web programming, and discusses our adaptation of the existing Links syntax and type inference mechanisms to support linearity and session types.

Section 6 discusses related work and Section 7 concludes.

2. The Core Language

The calculus we present in this section, FST (F with Session Types), is a call-by-value linear variant of System F with subkinding, row types, and session types. It combines a variant of GV, our session-typed linear λ-calculus [20], with the row typing and subkinding of our previous core language for Links [18], and the similar approach to subkinding for linearity of Mazurak and Zdancewic's lightweight linear types [22].

As our focus is the semantics of a programming language for session types rather than its logical underpinnings, we make some simplifications with respect to our earlier work [20]. Specifically, we have a single unlimited self-dual type of closed channels, and we omit the operation for linking channels together.

2.1 Syntax

To avoid duplication and keep the concurrent semantics of FST as simple, we strive to implement as much as possible in the functional core of FST, and limit the session typing constructs to the essentials. The only session type constructors are for output, input, and closed channels, and no special typing rules are needed for the primitives, which are specified as constants. Other features such as choice and selection can be straightforwardly encoded using features of the functional core.

Types and Kinds. The syntax of types and kinds is given below.

Ordinary Types	A, B	$::= A \to^{Y} B \mid \langle R\rangle \mid [R] \mid \forall \alpha^{K(Y,Z)}.A \mid \alpha \mid \overline{\alpha} \mid S$
Session Types	S	$::= !A.S \mid ?A.S \mid \mathsf{End} \mid \sigma \mid \overline{\sigma}$
Row Types	R	$::= \cdot \mid \ell : P; R \mid \rho \mid \overline{\rho}$
Presence Types	P	$::= \mathsf{Abs} \mid \mathsf{Pre}(A) \mid \theta \mid \overline{\theta}$
Types	T	$::= A \mid R \mid P$
Type Variables	$\alpha, \sigma, \rho, \theta$	
Labels	ℓ	
Label Sets	$\mathcal{L}$	$::= \{\ell_1, \ldots, \ell_k\}$
Kinds	J	$::= K(Y, Z)$
Primary Kinds	K	$::= Type \mid Row_{\mathcal{L}} \mid Presence$
Linearity	Y	$::= \bullet \mid \circ$
Restriction	Z	$::= \pi \mid \star$

The function type $A \to^{Y} B$ takes an argument of type A and returns a value of type B and has linearity Y. The record type $\langle R\rangle$ has fields given by the labels of row R. The variant type $[R]$ admits tagged values given by the labels of row R. The polymorphic type $\forall \alpha^{K(Y,Z)}.A$ is parameterized over the type variable α of kind $K(Y, Z)$.

The input type $?A.S$ receives an input of type A and proceeds as the session type S. Dually, the output type $!A.S$ sends an output of type A and proceeds as the session type S. The type End terminates a session; it is its own dual. We let σ range over session type variables and the dual of session type variable σ is $\overline{\sigma}$.

Row Types. Records and variants are defined in terms of row types. A row type represents a mapping from labels to ordinary types. A row type includes a list of distinct labels, each of which is annotated with a presence type. The presence type indicates whether the label is present with type A ($\mathsf{Pre}(A)$), absent (Abs), or polymorphic in its presence (θ).

Row types are either *closed* or *open*. A closed row type ends in $\cdot$. An open row type ends in a *row variable* ρ or its dual $\overline{\rho}$; the latter are only meaningful for session-kinded rows.

The mapping from labels to ordinary types represented by a closed row type is defined only on the labels that are explicitly listed in the row type, and cannot be extended. In contrast, the row variable in an open row type can be instantiated in order to extend the row type with additional labels. As usual, we identify rows up to reordering of labels.

$$\ell_1 : P_1; \ell_2 : P_2; R = \ell_2 : P_2; \ell_1 : P_1; R$$

Furthermore, absent labels in closed rows are redundant:

$$\ell : \mathsf{Abs}; \ell_1 : P_1, \ldots; \ell_n : P_n; \cdot = \ell_1 : P_1, \ldots; \ell_n : P_n; \cdot$$

Duality. The duality operation is lifted to session types, session row types, and session presence types in the standard way:

$$\begin{array}{rcl} \overline{?A.S} & = & !A.\overline{S} \\ \overline{!A.S} & = & ?A.\overline{S} \\ \overline{\mathsf{End}} & = & \mathsf{End} \\ \overline{\overline{\alpha}} & = & \alpha \end{array} \qquad \begin{array}{rcl} \overline{\cdot} & = & \cdot \\ \overline{\ell : P; R} & = & \ell : \overline{P}; \overline{R} \\ \overline{\overline{\rho}} & = & \rho \end{array} \qquad \begin{array}{rcl} \overline{\mathsf{Abs}} & = & \mathsf{Abs} \\ \overline{\mathsf{Pre}(S)} & = & \mathsf{Pre}(\overline{S}) \\ \overline{\overline{\theta}} & = & \theta \end{array}$$

Kinds. Types are classified into kinds. Ordinary types have kind *Type*. Row types R have kind $Row_{\mathcal{L}}$ where $\mathcal{L}$ is a set of labels not allowed in R. Presence types have kind *Presence*.

The three primary kinds are refined with a simple subkinding discipline, similar to the system described in our previous work on Links [18] and the system of Mazurak et al. on lightweight linear types [22]. A primary kind K is parameterized by a *linearity* Y and a *restriction* Z. The linearity can be either unlimited ($\bullet$) or linear ($\circ$). The restriction can be session typed (π) or unconstrained ($\star$). The interpretation of these parameters on row and presence kinds is pointwise on the ordinary types contained within the row or presence types inhabiting those kinds. For instance, the kind $Row_{\mathcal{L}}(\circ, \pi)$ is inhabited by row types whose fields are linear session types and the kind $Presence(\bullet, \star)$ is inhabited by presence types whose fields are unlimited unconstrained ordinary types.

We often omit the primary kind, assuming a default of *Type*. For instance, we write $\alpha^{(\bullet,\star)}$ instead of $\alpha^{Type(\bullet,\star)}$. We write $A \to B$ as an abbreviation for $A \to^{\bullet} B$.

Subkinding. The two sources of subkinding are the linearity and restriction parameters.

$$\frac{}{\vdash \bullet \leq \circ} \qquad \frac{}{\vdash \pi \leq \star}$$

$$\frac{\vdash Y \leq Y'}{\vdash K(Y,Z) \leq K(Y',Z)} \qquad \frac{\vdash Z \leq Z'}{\vdash K(Y,Z) \leq K(Y,Z')}$$

Our notion of linearity corresponds to usage, not alias freedom. Thus, any unlimited type can be used linearly, but not vice versa.

Kind and Type Environments.

$$\begin{array}{lll} \text{Kind Environments} & \Delta & ::= \cdot \mid \Delta, \alpha : K(Y,Z) \\ \text{Type Environments} & \Gamma & ::= \cdot \mid \Gamma, x : A \end{array}$$

Kind environments map type variables to kinds. Type environments map term variables to types.

Terms. The syntax of terms and values is given below.

$$\begin{array}{lll} \text{Terms} & L, M, N & ::= x \mid c \\ & & \mid \lambda^{Y} x^{A}.M \mid L\,M \\ & & \mid \Lambda\alpha^{J}.\,V \mid M\,T \\ & & \mid \langle\rangle \mid \langle \ell = M; N\rangle \\ & & \mid \mathsf{let}\,\langle\rangle \leftarrow M\,\mathsf{in}\,N \\ & & \mid \mathsf{let}\,\langle \ell = x; y\rangle \leftarrow M\,\mathsf{in}\,N \\ & & \mid (\ell\,M)^{R} \mid \mathsf{case}\,L\,\{\ell\,x \to M; y \to N\} \\ & & \mid \mathsf{case}_{\perp}\,L \\ \text{Values} & V, W & ::= x \\ & & \mid \lambda^{Y} x^{A}.M \\ & & \mid \Lambda\alpha^{K(Y,Z)}.\,V \\ & & \mid \langle\rangle \mid \langle \ell = V; W\rangle \\ & & \mid (\ell\,V)^{R} \\ \text{Constants} & c & ::= \mathsf{send} \mid \mathsf{receive} \mid \mathsf{fork} \end{array}$$

We let x range over term variables and c range over constants. Lambda abstractions $\lambda^{Y} x^{A}.M$ are annotated with linearity Y. Type abstractions $\Lambda\alpha^{J}.\,V$ are annotated with kind J. Note that the body of a type abstraction is restricted to be a syntactic value in the spirit of the ML value restriction (in order to avoid problems with polymorphic linearity and with polymorphic session types). Records are introduced with the unit record $\langle\rangle$ and record extension $\langle \ell = M; N\rangle$ constructs. They are eliminated with the binding form $\mathsf{let}\,\langle \ell = x; y\rangle \leftarrow M\,\mathsf{in}\,N$, which binds the value labeled by ℓ to x and the remainder of the record to y. Conventional projections $M.\ell$ are definable using this form, but note that because projection discards the remainder of the record, its applicability to records with linear components is limited. Variants are introduced with the injection $\ell\,M$ and eliminated using the $\mathsf{case}\,L\,\{\ell\,x \to M; y \to N\}$ and $\mathsf{case}_{\perp}\,L$ constructs.

Concurrency. The concurrency features of FST are provided by a collection of special constants. The term $\mathsf{send}\,\langle V, W\rangle$ sends V along channel W, returning the updated channel. The term $\mathsf{receive}\,W$ receives a value V along channel W, and returns a pair of V and the updated channel. The term $\mathsf{fork}\,(\lambda x.M)$ returns one end of a channel and forks a new process M in which x is bound to the other end of the channel.

Notation. We use the following abbreviations:

$$\begin{array}{rcl} \mathsf{let}\,x = M\,\mathsf{in}\,N & \stackrel{\text{def}}{=} & (\lambda x.N)\,M \\ M; N & \stackrel{\text{def}}{=} & \mathsf{let}\,x = M\,\mathsf{in}\,N, \quad x \text{ fresh} \\ \ell : A & \stackrel{\text{def}}{=} & \ell : \mathsf{Pre}(A) \\ \ell : P & \stackrel{\text{def}}{=} & \ell : P(\langle\rangle) \\ \langle A_1, \ldots, A_k\rangle & \stackrel{\text{def}}{=} & \langle 1 : A_1; \ldots; k : A_k; \cdot\rangle \\ \overrightarrow{\ell} & \stackrel{\text{def}}{=} & \ell_1, \ldots, \ell_k \\ \overrightarrow{\ell : P} & \stackrel{\text{def}}{=} & \ell_1 : P_1, \ldots, \ell_k : P_k \end{array}$$

We interpret n-ary record and case extension at the type and term levels in the obvious way. For instance

$$\langle \overrightarrow{\ell : P}; R\rangle \stackrel{\text{def}}{=} \langle \ell_1 : P_1; \ldots; \ell_n : P_n; R\rangle$$

and

$$\begin{array}{rcl} \mathsf{case}\,L\,\{\cdot\} & \stackrel{\text{def}}{=} & \mathsf{case}_{\perp}\,L \\ \mathsf{case}\,L\,\{z \to N\} & \stackrel{\text{def}}{=} & \mathsf{let}\,z = L\,\mathsf{in}\,N \\ \mathsf{case}\,L\,\{\ell\,x \to N; \chi\} & \stackrel{\text{def}}{=} & \mathsf{case}\,L\,\{\ell\,x \to N; z \to \mathsf{case}\,z\,\{\chi\}\} \end{array}$$

where we let χ range over sequences of cases:

$$\chi ::= \cdot \mid z \to N \mid \ell\,x \to N; \chi$$

We often write cases on separate lines without the ; delimiter.

We write $\mathrm{fv}(M)$ for the free variables of M. We write $\mathrm{ftv}(T)$ for the free type variables of a type T and $\mathrm{ftv}(\Gamma)$ for the free type

$\boxed{\Delta \vdash T : K(Y,Z)}$

Ordinary Types

$$\textsc{Function}\ \frac{\Delta \vdash A : \mathit{Type}(Y,\star) \qquad \Delta \vdash B : \mathit{Type}(Y',\star)}{\Delta \vdash A \to^{Y''} B : \mathit{Type}(Y'',\star)}$$

$$\textsc{Record}\ \frac{\Delta \vdash R : \mathit{Row}_\emptyset(Y,\star)}{\Delta \vdash \langle R \rangle : \mathit{Type}(Y,\star)} \qquad \textsc{Variant}\ \frac{\Delta \vdash R : \mathit{Row}_\emptyset(Y,\star)}{\Delta \vdash [R] : \mathit{Type}(Y,\star)}$$

$$\textsc{ForAll}\ \frac{\Delta, \alpha : K(\bullet, Z) \vdash A : \mathit{Type}(Y,\star)}{\Delta \vdash \forall \alpha^{K(Y',Z)}.A : \mathit{Type}(Y,\star)}$$

Session Types

$$\textsc{Input}\ \frac{\Delta \vdash A : \mathit{Type}(Y,\star) \quad \Delta \vdash S : \mathit{Type}(Y',\pi)}{\Delta \vdash ?A.S : \mathit{Type}(\circ,\pi)} \qquad \textsc{Output}\ \frac{\Delta \vdash A : \mathit{Type}(Y,\star) \quad \Delta \vdash S : \mathit{Type}(Y',\pi)}{\Delta \vdash !A.S : \mathit{Type}(\circ,\pi)}$$

$$\textsc{End}\ \frac{}{\Delta \vdash \mathsf{End} : \mathit{Type}(\bullet,\pi)}$$

Row Types

$$\textsc{EmptyRow}\ \frac{}{\Delta \vdash \cdot : \mathit{Row}_{\mathcal{L}}(Y,Z)} \qquad \textsc{ExtendRow}\ \frac{\Delta \vdash R : \mathit{Row}_{\mathcal{L} \uplus \{\ell\}}(Y,Z) \quad \Delta \vdash P : \mathit{Presence}(Y,Z)}{\Delta \vdash (\ell : P; R) : \mathit{Row}_{\mathcal{L}}(Y,Z)}$$

Presence Types

$$\textsc{Absent}\ \frac{}{\Delta \vdash \mathsf{Abs} : \mathit{Presence}(Y,Z)} \qquad \textsc{Presence}\ \frac{\Delta \vdash A : \mathit{Type}(Y,Z)}{\Delta \vdash \mathsf{Pre}(A) : \mathit{Presence}(Y,Z)}$$

Type Variables

$$\textsc{TyVar}\ \frac{\alpha : K(Y,Z) \in \Delta}{\Delta \vdash \alpha : K(Y,Z)} \qquad \textsc{DualTyVar}\ \frac{\alpha : K(Y,\pi) \in \Delta}{\Delta \vdash \overline{\alpha} : K(Y,\pi)}$$

Subkinding

$$\textsc{Upcast}\ \frac{\vdash J \leq J' \qquad \Delta \vdash T : J}{\Delta \vdash T : J'}$$

Figure 1. Kinding Rules

variables of type environment Γ. We write $\mathrm{dom}(\Gamma)$ for the domain of type environment Γ.

2.2 Typing and Kinding Judgments

The kinding rules are given in Figure 1. The kinding judgment $\Delta \vdash A : K(Y,Z)$ states that in kind environment Δ, the type A has kind $K(Y,Z)$. Type variables in the kind environment are well-kinded. The rules for forming function, record, variant, universally quantified, and presence types follow the syntactic structure of types. Because of the subkinding relation, a record is linear if any of its fields are linear, and similarly for variants. Recall that $\mathit{Row}_{\mathcal{L}}$ is the kind of row types whose labels cannot appear in $\mathcal{L}$. (To be clear, this constraint applies equally to absent and present labels; it is a constraint on the form of *row types*. In contrast, $\ell : \mathsf{Abs}$ in a row type is a constraint on *terms*.) An empty row has kind

$\boxed{\Delta \vdash \Gamma : Y}$

$$\textsc{LEmpty}\ \frac{}{\Delta \vdash \cdot : Y} \qquad \textsc{LExtend}\ \frac{\Delta \vdash \Gamma : Y \qquad \Delta \vdash A : K(Y,Z)}{\Delta \vdash (\Gamma, x : A) : Y}$$

$\boxed{\Delta \vdash \Gamma = \Gamma_1 + \Gamma_2}$

$$\textsc{CEmpty}\ \frac{}{\Delta \vdash \cdot = \cdot + \cdot} \qquad \textsc{C-}\bullet\ \frac{\Delta \vdash A : \mathit{Type}(\bullet,\star) \qquad \Delta \vdash \Gamma = \Gamma_1 + \Gamma_2}{\Delta \vdash \Gamma, x : A = (\Gamma_1, x : A) + (\Gamma_2, x : A)}$$

$$\textsc{C-}\circ\textsc{-Left}\ \frac{\Delta \vdash A : \mathit{Type}(\circ,\star) \qquad \Delta \vdash \Gamma = \Gamma_1 + \Gamma_2}{\Delta \vdash \Gamma, x : A = (\Gamma_1, x : A) + \Gamma_2}$$

$$\textsc{C-}\circ\textsc{-Right}\ \frac{\Delta \vdash A : \mathit{Type}(\circ,\star) \qquad \Delta \vdash \Gamma = \Gamma_1 + \Gamma_2}{\Delta \vdash \Gamma, x : A = \Gamma_1 + (\Gamma_2, x : A)}$$

Figure 2. Linearity of Contexts and Context Splitting

$\mathit{Row}_{\mathcal{L}}(Y,Z)$ for any label set $\mathcal{L}$, linearity Y, and restriction Z. The side-conditions $\ell \notin \mathcal{L}$ in ExtendRow ensures that row types have distinct labels. A row type can only be used to build a record or variant if it has kind $\mathit{Row}_\emptyset$. This constraint ensures that any absent labels in an open row type must be mentioned explicitly.

In Figure 2 we define two auxiliary judgments that for use in the typing rules. The linearity judgment $\Delta \vdash \Gamma : Y$ is the pointwise extension of the kinding judgment restricted to the linearity component of the kind. It states that in kind environment Δ, each type in environment Γ has linearity Y. The type environment splitting judgment $\Delta \vdash \Gamma = \Gamma_1 + \Gamma_2$ states that in kind environment Δ, the type environment Γ can be split into type environments Γ_1 and Γ_2. Contraction of unlimited types is built into this judgment.

The typing rules are given in Figure 3. The typing judgment $\Delta; \Gamma \vdash M : A$ states that in kind environment Δ and type environment Γ, the term M has type A. We assume that Γ and A are well-kinded with respect to Δ. If Δ and Γ are empty (that is, M is a closed term), then we will often omit them, writing $\vdash M : A$ for $\cdot; \cdot \vdash M : A$.

We assume a signature Σ mapping constants to their types. The definition of Σ on the basic concurrency primitives is given in Figure 4.

The Extend rule is strict in the sense that it requires a label to be absent from a record before the record can be extended with the label. The Case rule refines the type of the value being matched so that in the type of the variable bound by the default branch, the non-matched label is absent.

Selection and Choice. Traditional accounts of session types include types for selection and choice. Following our previous work [20], inspired by Kobayashi [15], we encode selection and choice using variant types.

$$\begin{aligned} \oplus\{R\} &\stackrel{\text{def}}{=} ![\overline{R}].\mathsf{End} \\ \&\{R\} &\stackrel{\text{def}}{=} ?[R].\mathsf{End} \\ \mathsf{select}\ \ell\ M &\stackrel{\text{def}}{=} \mathsf{fork}\,(\lambda x.\mathsf{send}\,\langle \ell\, x, M\rangle) \\ \mathsf{offer}\ L\,\{\chi\} &\stackrel{\text{def}}{=} \mathsf{let}\,\langle x, z\rangle = \mathsf{receive}\,L\,\mathsf{in}\,\mathsf{case}\,x\,\{\chi\} \end{aligned}$$

In the implementation of Session Links we support selection and choice in the source language. This is primarily for programming convenience. One might imagine desugaring these using the rules above, and then potentially rediscovering them in the back-end for performance reasons.

$\boxed{\Delta;\Gamma \vdash M : A}$

$$\textsc{Var}\ \frac{\Delta \vdash \Gamma : \bullet}{\Delta;\Gamma, x : A \vdash x : A} \qquad \textsc{Const}\ \frac{\Sigma(c) = A}{\Delta;\Gamma \vdash c : A}$$

$$\textsc{LinLam}\ \frac{\Delta;\Gamma, x : A \vdash M : B}{\Delta;\Gamma \vdash \lambda^{\circ} x^{A}.M : A \to^{\circ} B} \qquad \textsc{UnlLam}\ \frac{\Delta \vdash \Gamma : \bullet \quad \Delta;\Gamma, x : A \vdash M : B}{\Delta;\Gamma \vdash \lambda^{\bullet} x^{A}.M : A \to^{\bullet} B}$$

$$\textsc{App}\ \frac{\Delta \vdash \Gamma = \Gamma_1 + \Gamma_2 \quad \Delta;\Gamma_1 \vdash L : A \to^{Y} B \qquad \Delta;\Gamma_2 \vdash M : A}{\Delta;\Gamma \vdash L\,M : B}$$

$$\textsc{PolyLam}\ \frac{\Delta, \alpha :: K(\bullet, Z);\Gamma \vdash V : A \qquad \alpha \notin \mathrm{ftv}(\Gamma)}{\Delta;\Gamma \vdash \Lambda\alpha^{K(Y,Z)}.\,V : \forall\alpha^{K(Y,Z)}.A}$$

$$\textsc{PolyApp}\ \frac{\Delta;\Gamma \vdash M : \forall\alpha^{K(Y,Z)}.A \qquad \Delta \vdash T :: K(Y,Z)}{\Delta;\Gamma \vdash M\,T : A[\alpha := T]} \qquad \textsc{Unit}\ \frac{\Delta \vdash \Gamma : \bullet}{\Delta;\Gamma \vdash \langle\rangle : \langle\rangle}$$

$$\textsc{Extend}\ \frac{\Delta \vdash \Gamma = \Gamma_1 + \Gamma_2 \quad \Delta;\Gamma_1 \vdash M : A \qquad \Delta;\Gamma_2 \vdash N : \langle \ell : \mathsf{Abs}; R\rangle}{\Delta;\Gamma \vdash \langle \ell = M; N\rangle : \langle \ell : \mathsf{Pre}(A); R\rangle}$$

$$\textsc{LetUnit}\ \frac{\Delta \vdash \Gamma = \Gamma_1 + \Gamma_2 \quad \Delta;\Gamma_1 \vdash M : \langle\rangle \qquad \Delta;\Gamma_2 \vdash N : B}{\Delta;\Gamma \vdash \mathsf{let}\,\langle\rangle \leftarrow M\,\mathsf{in}\,N : B}$$

$$\textsc{LetRecord}\ \frac{\Delta \vdash \Gamma = \Gamma_1 + \Gamma_2 \quad \Delta;\Gamma_1 \vdash M : \langle \ell : \mathsf{Pre}(A); R\rangle \qquad \Delta;\Gamma_2, x : A, y : \langle R\rangle \vdash N : B}{\Delta;\Gamma \vdash \mathsf{let}\,\langle \ell = x; y\rangle \leftarrow M\,\mathsf{in}\,N : B}$$

$$\textsc{Inject}\ \frac{\Delta;\Gamma \vdash M : A}{\Delta;\Gamma \vdash (\ell\,M)^{R} : [\ell : \mathsf{Pre}(A); R]}$$

$$\textsc{Case}\ \frac{\Delta \vdash \Gamma = \Gamma_1 + \Gamma_2 \qquad \Delta;\Gamma_1 \vdash L : [\ell : \mathsf{Pre}(A); R] \quad \Delta;\Gamma_2, x : A \vdash M : B \qquad \Delta;\Gamma_2, y : [\ell : \mathsf{Abs}; R] \vdash N : B}{\Delta;\Gamma \vdash \mathsf{case}\,L\,\{\ell\,x \to M; y \to N\} : B}$$

$$\textsc{CaseZero}\ \frac{\Delta;\Gamma \vdash L : []}{\Delta;\Gamma \vdash \mathsf{case}_{\perp} L : B}$$

Figure 3. Typing Rules

$$\begin{aligned}
\Sigma(\mathsf{send}) &= \forall\alpha^{\circ,\star}.\forall\sigma^{\circ,\pi}.\langle\alpha, !\alpha.\sigma\rangle \to^{\bullet} \sigma \\
\Sigma(\mathsf{receive}) &= \forall\alpha^{\circ,\star}.\forall\sigma^{\circ,\pi}.?\alpha.\sigma \to^{\bullet} \langle\alpha, \sigma\rangle \\
\Sigma(\mathsf{fork}) &= \forall\sigma^{\circ,\pi}.\forall\alpha^{\bullet,\star}.(\sigma \to^{\circ} \alpha) \to^{\bullet} \overline{\sigma}
\end{aligned}$$

Figure 4. Type Schemas for Constants

$$\begin{array}{lll}
\text{Evaluation contexts} & E & ::= [\,] \mid E\,M \mid V\,E \mid E\,T \\
& & \quad \mid \langle \ell = E; N\rangle \mid \langle \ell = V; E\rangle \\
& & \quad \mid \mathsf{let}\,\langle\rangle = E\,\mathsf{in}\,N \\
& & \quad \mid \mathsf{let}\,\langle \ell = x; y\rangle = E\,\mathsf{in}\,N \\
& & \quad \mid \ell\,E \\
& & \quad \mid \mathsf{case}\,E\,\{\ell\,x \to M; y \to N\} \\
& & \quad \mid \mathsf{case}_{\perp}\,E \\
& F & ::= \phi E \\
\text{Configurations} & C, D & ::= \phi M \mid C \parallel D \mid (\nu x)C \\
& & \quad \mid x(\vec{V}) \leftrightsquigarrow y(\vec{W}) \\
\text{Configuration contexts} & G & ::= [\,] \mid G \parallel C \mid (\nu x)G \\
\text{Flags} & \phi & ::= \Diamond \mid \blacklozenge
\end{array}$$

Figure 5. Syntax of Configurations and Contexts

3. Semantics

We give an asynchronous small-step operational semantics for FST. Following Gay and Vasconcelos [11], whose calculus we call LAST (for Linear Asynchronous Session Types), we factor the semantics into functional and concurrent reduction relations, and introduce explicit buffers to provide asynchrony. For the functional fragment of the language, we give a standard left-to-right call-by-value semantics. The semantics of the concurrent portion of the language is given by a reduction relation on configurations of threads and buffers. This differs from our previous work on GV [20] by the introduction of buffers, allowing asynchrony between the sending and receiving of a message, and by using standard β-reduction instead of weak explicit substitutions. FST, like GV but unlike LAST, is deadlock-free, deterministic, and terminating.

3.1 Expressions, Configurations and Reduction

Figure 5 gives the syntax of evaluation contexts (which are standard), thread configurations, and configuration contexts. In addition to standard notions of name restriction and parallel composition, configurations include threads (ϕM) and buffers ($x(\vec{V}) \leftrightsquigarrow y(\vec{W})$). Our treatment of threads accounts for the functional nature of FST—as functional programs return values, we distinguish between the main thread ($\blacklozenge M$), which computes the return value, and its child threads ($\Diamond M$), which do not. Buffers mediate communication, allowing sending threads to continue immediately while queuing sent values for the receiving thread. A buffer $x(\vec{V}) \leftrightsquigarrow y(\vec{W})$ has two endpoints (x and y) and two queues of values ($\vec{V}$ and $\vec{W}$). The queue $\vec{V}$ holds those values to be read on endpoint x; correspondingly, writes to endpoint x are stored in $\vec{W}$.

Figure 6 gives reduction and equivalence relations for FST. We write $\mathrm{fv}(C)$ for the free variables of configuration C. The term reduction $\longrightarrow_{\mathrm{V}}$ is standard. Configuration equivalence reflects that $\parallel$ is associative and commutative, and allows name restriction to be floated outwards. We have two unusual equivalences: first, we reflect that buffers are symmetric, and second, we allow garbage collection of buffers once neither of their endpoints are in use by any threads. The reduction rules for send and receive are straightforward. In each case, the reduction involves a thread and a buffer. Threads performing send can always reduce, while threads performing receive can reduce only if there is a buffered value to be read. The rule for fork introduces a new buffer and spawns a child process.

FST programs are free of deadlock. However, our grammar clearly admits deadlocked configurations. For example, if we define

$$\begin{aligned}
M_{xy} = \mathsf{let}\,\langle z, x\rangle = \mathsf{receive}\,x\,\mathsf{in} \\
\mathsf{let}\,y = \mathsf{send}\,\langle z, y\rangle\,\mathsf{in}\,\langle\rangle
\end{aligned}$$

Term reduction

$$
\begin{array}{lrcl}
(\to.\beta) & (\lambda x.M)\,V & \longrightarrow_{\mathrm{V}} & M\{V/x\} \\
(\forall.\beta) & (\Lambda\alpha.V)\,T & \longrightarrow_{\mathrm{V}} & V\{T/\alpha\} \\
(\textsc{Unit}.\beta) & \mathsf{let}\,\langle\rangle = \langle\rangle\,\mathsf{in}\,N & \longrightarrow_{\mathrm{V}} & N \\
(\textsc{Record}.\beta) & \mathsf{let}\,\langle \ell = x; y\rangle = \langle \ell = V; W\rangle\,\mathsf{in}\,N & \longrightarrow_{\mathrm{V}} & N\{V/x, W/y\} \\
(\textsc{Match}.\beta) & \mathsf{case}\;\ell\;V\;\{\ell\,x \to M; y \to N\} & \longrightarrow_{\mathrm{V}} & M\{V/x\} \\
(\textsc{Default}.\beta) & \mathsf{case}\;\ell\;V\;\{\ell'\,x \to M; y \to N\} & \longrightarrow_{\mathrm{V}} & N\{\ell\,V/y\}, \quad \text{if } \ell \neq \ell'
\end{array}
$$

$$\textsc{TermEval}\quad \frac{M \longrightarrow_{\mathrm{V}} M'}{E[M] \longrightarrow_{\mathrm{V}} E[M']}$$

Configuration equivalence

$$C \parallel D \equiv D \parallel C \qquad C_1 \parallel (C_2 \parallel C_3) \equiv (C_1 \parallel C_2) \parallel C_3 \qquad C \parallel (\nu x)D \equiv (\nu x)(C \parallel D),\ \text{if } x \notin \mathrm{fv}(C) \qquad G[C] \equiv G[D],\ \text{if } C \equiv D$$

$$x(\vec{V}) \leftrightsquigarrow y(\vec{W}) \equiv y(\vec{W}) \leftrightsquigarrow x(\vec{V}) \qquad (\nu xy)(x(\epsilon) \leftrightsquigarrow y(\epsilon)) \parallel C \equiv C \qquad \Diamond\langle\rangle \parallel C \equiv C$$

Configuration reduction

$$
\begin{array}{lrcl}
(\textsc{Send}) & F[\mathsf{send}\,\langle W', x\rangle] \parallel x(\vec{V}) \leftrightsquigarrow y(\vec{W}) & \longrightarrow & F[x] \parallel x(\vec{V}) \leftrightsquigarrow y(\vec{W}W') \\
(\textsc{Receive}) & F[\mathsf{receive}\,x] \parallel x(V'\vec{V}) \leftrightsquigarrow y(\vec{W}) & \longrightarrow & F[(V', x)] \parallel x(\vec{V}) \leftrightsquigarrow y(\vec{W}) \\
(\textsc{Fork}) & F[\mathsf{fork}\,(\lambda z.M)] & \longrightarrow & (\nu xy)(F[x] \parallel \Diamond M\{y/z\} \parallel x(\epsilon) \leftrightsquigarrow y(\epsilon)), \quad x, y \text{ fresh}
\end{array}
$$

$$\textsc{TermConfig}\quad \frac{M \longrightarrow_{\mathrm{V}} M'}{G[M] \longrightarrow G[M']} \qquad \textsc{ConfigConfig}\quad \frac{C \longrightarrow C'}{G[C] \longrightarrow G[C']}$$

Figure 6. Reduction Rules and Equivalences for Terms and Configurations

then the configuration

$$(\nu xx'yy')(M_{xy} \parallel x(\epsilon) \leftrightsquigarrow x'(\epsilon) \parallel y(\epsilon) \leftrightsquigarrow y'(\epsilon) \parallel M_{y'x'})$$

is stuck. To rule out such configurations, we introduce a typing relation on configurations, given in Figure 7. The primary typing relation $\Delta; \Gamma \vdash^{\phi} C$ indicates that a configuration is well-typed. The rules for threads ensure that only the main thread has a non-trivial return value. In order to define name restriction, we extend the type system inside configuration contexts. Name restriction introduces a new channel of type $S^\sharp$; intuitively, this represents both endpoints of the channel, and will be split into S and $\overline{S}$ endpoints at a later composition. Note that channel types are not permitted in terms (the MAIN and CHILD rules have implicit side-conditions asserting that Γ is not allowed to contain a channel type), so configurations such as $(\nu x)(\blacklozenge x)$ are ruled out by the type system.

The composition rules impose two restrictions. First, a configuration contains at most one main thread $\blacklozenge M$, which is ensured by leaving $\blacklozenge + \blacklozenge$ undefined. Second, at most one channel may be shared across a composition of threads. Note that only channels of type $S^\sharp$ can be shared, with the dual types S and $\overline{S}$ in the subconfigurations.

The buffer typing rule requires that values stored in the buffer match the expectations of the endpoints and that the endpoints are dual. These properties are captured by two auxiliary judgments. First, $\Delta; \Gamma \vdash \vec{V} : \vec{T}$ expresses that the queue of values V have types T; the environment is necessary as channels themselves can appear as values in buffers. Second, the quotient $S/\vec{T}$ denotes the "remainder" of session type S after sending values of types $\vec{T}$. Note that, because the endpoint types must be dual and $S/\vec{T}$ is defined only if $S = !T.S'$, at most one of the queues can be non-empty in a well-typed buffer.

We conclude our summary of the semantics with a subject reduction (type preservation) theorem.

Lemma 1. *If $\Delta; \Gamma \vdash M : A$ and $M \longrightarrow_V N$, then $\Delta; \Gamma \vdash N : A$.*

The proof is by induction on M; the cases are standard. We now lift this result to configurations. We must account for the possibility that a configuration reduction advances the session type of one of the channels in the environment. We write $\Gamma \longrightarrow \Gamma'$ to denote that environment Γ can reduce to environment Γ' (see Figure 8), and $\longrightarrow^?$ for the reflexive closure of $\longrightarrow$.

Theorem 2. *If $\Delta; \Gamma \vdash C$ and $C \longrightarrow C'$, then there is a Γ' such that $\Gamma \longrightarrow^? \Gamma'$ and $\Delta; \Gamma' \vdash C'$.*

The proof is by induction the derivation of $C \longrightarrow C'$.

Our notion of typing is not preserved by configuration equivalence. For example, suppose that $\Delta; \Gamma \vdash C_1 \parallel (C_2 \parallel C_3)$ where $x \in \mathrm{fv}(C_1)$, $y \in \mathrm{fv}(C_2)$, $x, y \in \mathrm{fv}(C_3)$; we would have that $\Delta; \Gamma \vdash C_2 \parallel (C_1 \parallel C_3)$, but that $\Delta; \Gamma \nvdash (C_1 \parallel C_2) \parallel C_3$. Nonetheless, typing modulo equivalence is preserved by reduction.

Theorem 3. *If $\Delta; \Gamma \vdash C$, $C \equiv C'$, and $C' \longrightarrow D'$, then there is some Γ' and D such that $\Gamma \longrightarrow^? \Gamma'$, $\Delta; \Gamma' \vdash D$, and $D \equiv D'$.*

Our asynchronous semantics only admits communication between a thread and a buffer: a thread can send a value to a buffer or receive a value from a buffer. However, our type system does not distinguish between channels used in threads and channels that appear as endpoints of buffers. Thus, the following configuration is well-typed, but cannot reduce:

$$(\nu x)(\blacklozenge\,(\mathsf{receive}\,x) \parallel \Diamond\,(\mathsf{send}\,\langle\langle\rangle, x\rangle; \langle\rangle))$$

There are a number of ways to address this issue. For example, we could distinguish between buffer endpoints and channels used in threads (but this would complicate the typing rules). Instead, we introduce the notion of a *well-buffered* configuration, defined in Figure 9. Configuration C is well-buffered with respect to variables $\mathcal{X}$ ($\mathrm{wb}_{\mathcal{X}}(C)$) if every variable $x \in \mathcal{X}$ occurs as the end point of exactly one buffer in C. It is straightforward to show that well-buffering is preserved by reduction:

Theorem 4. *If $\mathrm{wb}_{\mathcal{X}}(C)$ and $C \longrightarrow C'$ then $\mathrm{wb}_{\mathcal{X}}(C')$.*

The judgment $\Delta; \Gamma \vdash_{\mathrm{wb}} C$ states that well-typed configuration C is well-buffered with respect to all of its free variables of channel type.

3.1.1 Asynchrony

Our semantics for GV [20] is synchronous, whereas the semantics we have presented for FST is asynchronous. The synchronous se-

$\boxed{\Delta;\Gamma \vdash C}$

$$\textsc{Main}\ \frac{\Delta;\Gamma \vdash M : A}{\Delta;\Gamma \vdash^{\blacklozenge} \blacklozenge M} \qquad \textsc{Child}\ \frac{\Delta;\Gamma \vdash M : \langle\rangle}{\Delta;\Gamma \vdash^{\Diamond} \Diamond M}$$

$$\textsc{Buffer}\ \frac{\Delta \vdash \Gamma = \Gamma_1 + \Gamma_2 \quad \Delta;\Gamma_1 \vdash \vec{V} : \vec{A} \quad \Delta;\Gamma_2 \vdash \vec{W} : \vec{B} \quad S/\vec{A} = \overline{(S'/\vec{B})}}{\Delta;\Gamma, x : S, y : S' \vdash x(\vec{V}) \leftrightsquigarrow y(\vec{W})}$$

$$\textsc{NewSession}\ \frac{\Delta;\Gamma, x : S^\sharp \vdash^{\phi} C}{\Delta;\Gamma \vdash^{\phi} (\nu x)C} \qquad \textsc{Compose-1}\ \frac{\Delta \vdash \Gamma = \Gamma_1 + \Gamma_2 \quad \Delta;\Gamma_1, x : S \vdash^{\phi} C \quad \Delta;\Gamma_2, x : \overline{S} \vdash^{\phi'} C'}{\Delta;\Gamma, x : S^\sharp \vdash^{\phi+\phi'} C \parallel C'}$$

$$\textsc{Compose-0}\ \frac{\Delta \vdash \Gamma = \Gamma_1 + \Gamma_2 \quad \Delta;\Gamma_1 \vdash^{\phi} C \quad \Delta;\Gamma_2 \vdash^{\phi'} C'}{\Delta;\Gamma \vdash^{\phi+\phi'} C \parallel C'}$$

$\boxed{\phi + \phi = \phi}$

$\Diamond + \Diamond = \Diamond \qquad \Diamond + \blacklozenge = \blacklozenge \qquad \blacklozenge + \Diamond = \blacklozenge \qquad \blacklozenge + \blacklozenge$ undefined

$\boxed{\Delta;\Gamma \vdash \vec{V} : \vec{A}}$

$$\frac{\Delta \vdash \Gamma : \bullet}{\Delta;\Gamma \vdash \varepsilon : \varepsilon} \qquad \frac{\Delta \vdash \Gamma = \Gamma_1 + \Gamma_2 \quad \Delta;\Gamma_1 \vdash V : A \quad \Delta;\Gamma_2 \vdash \vec{A} : \vec{A}}{\Delta;\Gamma \vdash V\vec{V} : A\vec{A}}$$

$\boxed{S/\vec{T} = S}$

$$S/\epsilon = S \qquad !A.S/A\vec{A} = S/\vec{A}$$

$\boxed{\Delta \vdash T}$ $\quad \cdots \quad$ $\textsc{Sharp}\ \frac{\Delta \vdash S : \mathit{Type}(Y, \pi)}{\Delta \vdash S^\sharp : \mathit{Type}(\circ, \pi)}$

Figure 7. Configuration Typing

$\boxed{S \longrightarrow S}$ $\quad \boxed{\Gamma \longrightarrow \Gamma}$

$$\overline{(!A.S)^\sharp \longrightarrow S^\sharp} \qquad \overline{(?A.S)^\sharp \longrightarrow S^\sharp} \qquad \frac{S \longrightarrow S'}{\Gamma, x : S \longrightarrow \Gamma, x : S'}$$

Figure 8. Session Type Reduction

$$\overline{\mathrm{wb}_{\{x,y\}}(x(\vec{V}) \leftrightsquigarrow y(\vec{W}))} \qquad \overline{\mathrm{wb}_{\emptyset}(\phi M)}$$

$$\frac{\mathrm{wb}_{\mathcal{X}_1}(C_1) \quad \mathrm{wb}_{\mathcal{X}_2}(C_2)}{\mathrm{wb}_{\mathcal{X}_1 \uplus \mathcal{X}_2}(C_1 \parallel C_2)} \qquad \frac{\mathrm{wb}_{\mathcal{X} \uplus \{x\}}(C)}{\mathrm{wb}_{\mathcal{X}}((\nu x)C)}$$

$$\frac{\Delta;\Gamma \vdash^{\blacklozenge} C \quad \mathrm{wb}_{\{x \mid x:S^\sharp \in \Gamma\}}(C)}{\Delta;\Gamma \vdash_{\mathrm{wb}} C}$$

Figure 9. Well-buffered Configurations

mantics for GV is convenient in theory, particularly for drawing out the connections with cut-elimination in linear logic. In practice an asynchronous semantics is often more appropriate, and indeed our Session Links implementation is asynchronous. It is perfectly natural to define synchronous and asynchronous variants of GV and FST.

It is straightforward to show that the asynchronous semantics of GV admits all of the behaviors of the synchronous semantics: we simulate each synchronous reduction by a write to a buffer followed by a read from the same buffer. Furthermore, given that both the asynchronous semantics and the synchronous semantics are deterministic, and the asynchronous semantics can always simulate the synchronous semantics, it is clear that asynchronous reduction to a value will always produce the same result as synchronous reduction to a value. The same arguments apply for synchronous versus asynchronous variants of FST. Of course, if we lose determinism, such as when we add access points (§4.2), then this line of reasoning no longer holds.

3.2 Deadlock

Deadlock freedom is a prerequisite for ensuring conventional notions of progress (although, we will later consider a weaker notion of progress, suitable for more expressive systems that do admit deadlock). We will give a characterization of deadlock that allows us to show that typed configurations are deadlock-free, and because of preservation, that well-typed FST terms can never deadlock. Recall the example of a deadlocked configuration in the previous section: given the definition

$$M_{xy} = \mathsf{let}\ \langle z, x\rangle = \mathsf{receive}\ x\ \mathsf{in}\ \mathsf{let}\ y = \mathsf{send}\ \langle z, y\rangle\ \mathsf{in}\ \langle\rangle$$

the configuration

$$(\nu x x' y y')(M_{xy} \parallel x(\epsilon) \leftrightsquigarrow x'(\epsilon) \parallel y(\epsilon) \leftrightsquigarrow y'(\epsilon) \parallel M_{y'x'})$$

is deadlocked. In this case, the individual threads (M_{xy} and $M_{y'x'}$) are well-typed. However, the configuration is stuck because sending on x is blocked on receiving on y, while sending on y' is blocked on receiving on x'. In combination with the buffers connecting x to x' and y to y', we can see that these dependencies form a cycle. Our formalization of deadlock is based on such cycles of dependencies; Carbone and Debois consider a similar criterion for deadlock freedom in session-typed π-calculi [8]. We say that a term M is blocked on channel x, if M is about to send or receive on x:

$$\mathrm{blocked}(x, M) \iff M = E[\mathsf{receive}\, x] \vee M = E[\mathsf{send}\ \langle V, x\rangle]$$

We say that y depends on x in C if y appears in some thread blocked by x, or if y depends on some channel z which depends on x. Formally:

$$\begin{aligned} \mathrm{depends}(x, y, x(\vec{V}) \leftrightsquigarrow y(\vec{W})) & \\ \mathrm{depends}(x, y, \phi M) \iff \mathrm{blocked}(x, M) \wedge y \in \mathrm{fv}(M) & \\ \mathrm{depends}(x, y, C) \iff C \equiv G[C_1 \parallel C_2] \wedge & \\ \exists z.\mathrm{depends}(x, z, C_1) \wedge \mathrm{depends}(z, y, C_2) & \end{aligned}$$

We define deadlocked configurations as those with cyclic dependencies:

$$\mathrm{deadlocked}(C) \iff C \equiv G[C_1 \parallel C_2] \wedge \exists x, y.\ \mathrm{depends}(x, y, C_1) \wedge \mathrm{depends}(y, x, C_2)$$

We can now show that well-typed terms are not deadlocked. The crucial observation is graph theoretic. In a well-typed configuration, each composition partitions the available channels such that at most one channel is shared between the two subconfigurations. A cycle cannot have such a partitioning: it must have at least one composition where the subconfigurations share at least two channels. That is, if we think of the threads and buffers as the nodes in

a graph, and the edges as indicating channels, then if the graph can be partitioned where no partition has more than one cut edge, then it must be cycle free.

Theorem 5. *If* $\Delta;\Gamma \vdash_{\mathrm{wb}} C$, *then* $\neg\mathrm{deadlocked}(C)$.

Subject reduction ensures that well-typed FST terms can never reduce to deadlocked configurations.

Corollary 6.
If $\Delta;\Gamma \vdash M : A$ *and* $\phi M \longrightarrow^{\star} C$, *then* $\neg\mathrm{deadlocked}(C)$.

3.3 Progress and Canonical Forms

FST, just like GV [20], is not a pure language due to the concurrency primitives. Thus, a fully evaluated closed term will always yield a value, but if that value contains channels, then there may be blocked threads present in the final configuration.

We let H range over buffers and child threads, which we will subsequently refer to as *configuration leaves*. We begin by defining a notion of canonical form for configurations; in particular, this form identifies the main thread and (for closed configurations) the blocking channel for each child thread.

Definition 7. *A process C is in canonical form if there is a sequence of variables $x_1, \ldots, x_n$, a sequence of configuration leaves $H_1, \ldots, H_n$, and some term M, such that:*

$$C = (\nu x_1)(H_1 \parallel (\nu x_2)(H_2 \parallel \cdots \parallel (\nu x_{n-1})(H_{n-1} \parallel \blacklozenge M)\ldots))$$

Lemma 8. *If* $\Delta;\Gamma \vdash_{\mathrm{wb}} C$, *then there is some* $C' \equiv C$ *such that* $\Delta;\Gamma \vdash_{\mathrm{wb}} C'$ *and* C' *is in canonical form.*

This is established by a counting argument, relying on the acyclicity of well-typed configurations (established in the previous section). In the case of processes that share no channels, a channel of type End can be introduced to assure canonical form. It is intuitive to see that, if C is a stuck configuration in canonical form, each thread ϕM_j in C must be blocked either on one of the preceding ν-bound variables or on one of the free variables of the configuration.

Theorem 9. *Let* $\Delta;\Gamma \vdash_{\mathrm{wb}} C$, *with* $C \not\longrightarrow$ *and let* $C' = (\nu x_1)(H_1 \parallel (\nu x_2)(H_2 \parallel \cdots \parallel (\nu x_n)(H_n \parallel \phi N)\ldots))$ *be a canonical form of C. Then:*

1. *For* $1 \le i \le n$, *either H_i is a buffer, or* $H_i = \Diamond M_i$, *for some term M_i, and* $\mathrm{blocked}(x_j, M_i)$, *where* $j \le i$ *or* $\mathrm{blocked}(y, M_i)$ *for some* $y \in \mathrm{dom}(\Gamma)$*; and*
2. *Either N is a value or* $\mathrm{blocked}(y, N)$ *for some* $y \in \{x_i \mid 1 \le i \le n\} \cup \mathrm{dom}(\Gamma)$.

If we assume that C is a closed configuration, we can state a more precise result.

Theorem 10. *Let* $\vdash_{\mathrm{wb}} C$, *with* $C \not\longrightarrow$ *and let* $C' = (\nu x_1)(H_1 \parallel (\nu x_2)(H_2 \parallel \cdots \parallel (\nu x_n)(H_n \parallel \phi N)\ldots))$ *be a canonical form of C. Then:*

1. *For* $1 \le i \le n$, *either H_i is a buffer, or* $H_i = \Diamond M_i$, *for some term M_i, and* $\mathrm{blocked}(x_i, M_i)$*; and*
2. *N is a value.*

Intuitively, we know that the first thread must be blocked on the first ν-bound variable, x_1. The second thread cannot be blocked on x_1, as it could then reduce, so it must be blocked on a different ν-bound variable, and so on. Note that some of the x_i must appear in N; thus, as an immediate corollary, we have that closed configurations that do not return channels evaluate to values.

Corollary 11. *Let* $\vdash_{\mathrm{wb}} C$ *such that* $C \not\longrightarrow$*; if the value returned by C contains no channels, then* $C \equiv \phi V$ *for some value V.*

3.4 Determinism and Termination

FST is deterministic. In fact, FST enjoys a strong form of determinism, called the diamond property [5].

Theorem 12 (Diamond property)**.** *If* $\Delta;\Gamma \vdash C$, $C \equiv\!\!\longrightarrow\!\!\equiv D_1$, *and* $C \equiv\!\!\longrightarrow\!\!\equiv D_2$, *then there exists* D_3 *such that* $D_1 \equiv\!\!\longrightarrow\!\!\equiv D_3$, *and* $D_2 \equiv\!\!\longrightarrow\!\!\equiv D_3$.

FST is strongly normalizing.

Theorem 13 (Strong normalization)**.** *If* $\Delta;\Gamma \vdash C$, *then there are no infinite* $\equiv\!\!\longrightarrow\!\!\equiv$ *reduction sequences beginning from C.*

As FST satisfies the diamond property, strong normalization is in fact implied by weak normalization. A canonical approach to proving strong normalization directly is to construct a logical relations proof along the lines of Perez et al. [24] taking account of the impredicativity of FST. The approach we take instead is to define a translation into Fω, an existing strongly normalizing language, and show that terms in the image of the translation simulate reduction in FST.

The full details of the translation are beyond the scope of this paper. Here we give a brief sketch.

- The problem of defining a translation is simplified somewhat by observing that we can disregard linearity and subkinding in the target calculus (it is perfectly acceptable for the target language to admit terms and reductions that have no equivalent in the source language).
- The problem can be further simplified by switching to a synchronous semantics and observing that because of the diamond property synchronous reduction can simulate asynchronous reduction (§3.1.1).
- It is straightforward to simulate reductions of row-typed terms in Fω [18].
- The most interesting part of the translation is the simulation of concurrency. We do this by a CPS translation. The translation of send and receive is unsurprising. The translation of fork is unusual because it is type-directed, differing depending on whether the channel bound by the forked thread has input, output, or closed session type.
- Having defined the CPS translation on terms, it is straightforward to extend it to cover configurations. The interesting case is parallel composition whose CPS translation is much like that of fork.

4. Extensions

FST can be straightforwardly extended with additional features.

If we add a fixed point constant, then we lose termination, but deadlock freedom and determinism continue to hold. Another standard extension supported by Session Links is recursive types. While care is needed in defining the dual of a recursive session type, the treatment is otherwise quite standard. Negative recursive types allow a fixed point combinator to be defined, so again we lose termination, but deadlock freedom and determinism continue to hold.

The price we pay for the strong properties we obtain is that our model of concurrency is rather weak. For instance, it gives us no way of implementing a server with any notion of shared state. Drawing from LAST (and previous work on session-typed π-calculi), Session Links supports *access points*, which provide a much more expressive model of concurrency at the cost of introducing deadlock. Nevertheless, it is often possible to locally restrict code to a deadlock-free subset of Session Links.

4.1 Recursion

The grammar of session types we have presented so far is extremely limited: it cannot express repeated behavior. A typical approach to doing so is to add recursive session types. Continuing with the calculator example, we could write a recursive session type to allow multiple calculations as follows

$$\mathsf{rec}\,\sigma^{(\star,\pi)}.\&\{\mathit{Add} : ?\mathit{Int}.?\mathit{Int}.!\mathit{Int}.\sigma, \mathit{Neg} : ?\mathit{Int}.!\mathit{Int}.\sigma, \mathit{Stop} : \mathsf{End}\}$$

Here we have replaced each End with the recursion variable σ; consequently, we add a new branch to terminate the interaction.

We can straightforwardly extend FST with equi-recursive types. We add a kinding rule for recursive types

$$\textsc{Rec}\ \frac{\Delta, \alpha : \mathit{Type}(Y,Z) \vdash A}{\Delta \vdash \mathsf{rec}\,\alpha^{(Y,z)}.A}$$

and identify each recursive type with its unrolling:

$$\mathsf{rec}\,\alpha^{(Y,Z)}.A = A[\mathsf{rec}\,\alpha^{(Y,Z)}.A/\alpha]$$

It is well-known [6, 7] that recursive types complicate the definition of duality, particularly when the recursion variable appears as a carried type (that is, as A in $?A.S$ or $!A.S$). For example, consider the simple recursive session type $\mathsf{rec}\,\sigma^{\circ,\pi}.?\sigma.\sigma$. The dual of this type is not $\mathsf{rec}\,\sigma^{\circ,\pi}.!\sigma.\sigma$, as one would obtain by taking the dual of the body of the recursive type directly, but is $\mathsf{rec}\,\sigma^{\circ,\pi}.!\overline{\sigma}.\sigma$ instead. Previous solutions to this problem [6, 7] involve a new notion of substitution that only applies to the carried types in session types. We give an alternative, but equivalent, definition that relies on standard notions of substitution, as follows:

$$\overline{\mathsf{rec}\,\sigma^{X,\pi}.S} = \mathsf{rec}\,\sigma^{X,\pi}.\overline{(S[\overline{\sigma}/\sigma])}$$

Having added recursive types, one can of course encode a fixed point combinator. Alternatively, we can add a fixed point constant to FST, even without recursive types

$$\begin{aligned}\Sigma(\mathsf{fix}) \;=\; & \forall\alpha^{(\bullet,\star)}.\forall\beta^{(\bullet,\star)}.\\ & ((\alpha \to^{\bullet} \beta) \to^{\bullet} (\alpha \to^{\bullet} \beta)) \to^{\bullet} (\alpha \to^{\bullet} \beta)\end{aligned}$$

with the reduction rule

$$\mathsf{fix}\,(\lambda f.\lambda x.M)\; V \longrightarrow_{\mathsf{V}} M\{\mathsf{fix}\,(\lambda f.\lambda x.M)/f, V/x\}$$

Of course, these extensions allows us to write nonterminating programs, but it is straightforward to show that subject reduction, progress, deadlock freedom, and determinism continue to hold.

4.2 Access Points

Access points provide a more flexible mechanism for session initiation than the fork primitive. Intuitively, we can think of access points as providing a matchmaking service for processes. Processes may either accept or request connections at a given access point; accepting and requesting processes are paired nondeterministically. As a simple example, we can adapt our calculator example to synchronize on an access point instead of a fixed channel:

$$\begin{aligned}\mathit{calc}_{\mathsf{AP}} = \lambda a\, z.&\mathsf{let}\, c = \mathsf{accept}\, a\, \mathsf{in}\\ &\mathsf{offer}\, c\, \{\\ &\quad \mathit{Add}\, c \to \mathsf{let}\, \langle x, c\rangle = \mathsf{receive}\, c\, \mathsf{in}\\ &\qquad\qquad \mathsf{let}\, \langle y, c\rangle = \mathsf{receive}\, c\, \mathsf{in}\\ &\qquad\qquad \mathsf{let}\, c = \mathit{Send}\, \langle x + y, c\rangle\, \mathsf{in}\\ &\qquad\qquad\quad \mathit{calc}_{\mathsf{AP}}\, a\, z\\ &\quad \mathit{Neg}\, c \to \mathsf{let}\, \langle x, c\rangle = \mathsf{receive}\, c\, \mathsf{in}\\ &\qquad\qquad \mathsf{let}\, c = \mathit{Send}\, \langle -x, c\rangle\, \mathsf{in}\\ &\qquad\qquad\quad \mathit{calc}_{\mathsf{AP}}\, a\, z\\ &\quad \mathit{MP}\, c \to \mathsf{let}\, \langle x, c\rangle = \mathsf{receive}\, c\, \mathsf{in}\\ &\qquad\qquad\quad \mathit{calc}_{\mathsf{AP}}\, a\, (z + x)\\ &\quad \mathit{MR}\, c \to \mathsf{let}\, c = \mathsf{send}\, \langle z, c\rangle\, \mathsf{in}\\ &\qquad\qquad\quad \mathit{calc}_{\mathsf{AP}}\, a\, z\}\end{aligned}$$

We assume some form of recursive definition; we will show later in this section that access points are themselves sufficient to encode term-level recursion. We have also extended the calculator with a single memory register, as is not uncommon for desktop calculators. The $\mathit{calc}_{\mathsf{AP}}$ function takes two arguments: an access point a and the value of the register z. Note that because access points are unlimited, a client can make multiple requests on the same access point, and so we have no need to also build recursion into each interaction with the calculator. The behavior of an individual action is now

$$\begin{aligned}S = \&\{&\mathit{Add} : ?\mathit{Int}.?\mathit{Int}.!\mathit{Int}.\mathsf{End}, \mathit{Neg} : ?\mathit{Int}.!\mathit{Int}.\mathsf{End},\\ &\mathit{MP} : ?\mathit{Int}.\mathsf{End}, \mathit{MR} : !\mathit{Int}.\mathsf{End}\}\end{aligned}$$

and the type of $\mathit{calc}_{\mathsf{AP}}$, given the above, is:

$$\mathit{calc}_{\mathsf{AP}} : \forall\alpha.\mathsf{AP}(S) \to^{\bullet} \mathit{Int} \to^{\bullet} \alpha$$

We can define a client as before, but beginning with a call to request:

$$\begin{aligned}\mathit{user}_{\mathsf{AP}} = \lambda z.&\mathsf{let}\, c = \mathsf{request}\, a\, \mathsf{in}\\ &\mathsf{let}\, \langle x, c\rangle = \mathsf{receive}\, \mathsf{send}\, \langle 6\, \mathsf{send}\, \langle 7, c\rangle\rangle\, \mathsf{in}\, x\end{aligned}$$

Finally, to compose them we use new to create a fresh access point and spawn to spawn threads.

$$\begin{aligned}&\mathsf{let}\, a = \mathsf{new}\, \mathsf{in}\\ &\quad \mathsf{spawn}\, (\lambda^{\circ}\langle\rangle.\mathit{calc}_{\mathsf{AP}}\, a);\\ &\quad \mathit{user}_{\mathsf{AP}}\, a\end{aligned}$$

The result of evaluation is again 13.

In order to extend FST with access points, we replace the constant fork with four new constants:

$$\begin{aligned}\Sigma(\mathsf{spawn}) &= \forall\alpha^{\bullet,\star}.(\langle\rangle \to^{\circ} \alpha) \to^{\bullet} \langle\rangle\\ \Sigma(\mathsf{new}) &= \forall\sigma^{\circ,\pi}.\langle\rangle \to^{\bullet} \mathsf{AP}\ \sigma\\ \Sigma(\mathsf{accept}) &= \forall\sigma^{\circ,\pi}.\mathsf{AP}\ \sigma \to^{\bullet} \sigma\\ \Sigma(\mathsf{request}) &= \forall\sigma^{\circ,\pi}.\mathsf{AP}\ \sigma \to^{\bullet} \overline{\sigma}\end{aligned}$$

A process M is spawned with spawn M, where M is a thunk that returns an arbitrary unlimited value. The spawn operation can be defined in terms of fork

$$\mathsf{spawn}\, M \stackrel{\text{def}}{=} (\lambda x^{\mathsf{End}}.\langle\rangle)(\mathsf{fork}\, (\lambda x^{\mathsf{End}}.M\, \langle\rangle))$$

and vice versa:

$$\mathsf{fork}\, M \stackrel{\text{def}}{=} \mathsf{let}\, z = \mathsf{new}\, \langle\rangle\, \mathsf{in}\, \mathsf{spawn}\, (\lambda x.M\, (\mathsf{accept}\, z)); \mathsf{request}\, z$$

Session-typed channels are created through access points. A fresh access point of type $\mathsf{AP}\ S$ is created with new. Given an access point L of type $\mathsf{AP}\ S$ we can create a new server channel (accept L), of session type S, or client channel (request L), of session type $\overline{S}$. Processes can accept and request an arbitrary number of times on any given access point. Access points are synchronous in the sense that each accept will block until it is paired up with a corresponding request and vice-versa.

In order to model access points in the operational semantics we add a new kind of configuration

$$C ::= \cdots \mid z(\mathcal{X}, \mathcal{Y})$$

which associates each access point z with a set of server names $\mathcal{X}$ and a set of client names $\mathcal{Y}$, and we add an associated typing rule

$$\textsc{Access}\ \frac{\Delta \vdash \Gamma : \bullet}{\Delta; \Gamma, z : \mathsf{AP}\ S, \mathcal{X} : S, \mathcal{Y} : \overline{S} \vdash z(\mathcal{X}, \mathcal{Y})}$$

where we write $\mathcal{X} : A$ as shorthand for $x_1 : A, \ldots, x_k : A$, where $\mathcal{X} = \{x_1, \ldots, x_k\}$.

We introduce reduction rules for the constants and for access points themselves. The reduction rule for access points makes it

clear that the semantics is no longer deterministic.

$$\begin{array}{rcl} F[\mathsf{spawn}\ V] & \longrightarrow & F[\langle\rangle] \parallel V\,\langle\rangle \\ F[\mathsf{new}\ \langle\rangle] & \longrightarrow & (\nu z)(F[z] \parallel z(\emptyset,\emptyset)) \\ F[\mathsf{accept}\ z] \parallel z(\mathcal{X},\mathcal{Y}) & \longrightarrow & (\nu x)(F[x] \parallel z(\{x\} \uplus \mathcal{X},\mathcal{Y})) \\ F[\mathsf{request}\ z] \parallel z(\mathcal{X},\mathcal{Y}) & \longrightarrow & (\nu y)(F[y] \parallel z(\mathcal{X},\{y\} \uplus \mathcal{Y})) \\ z(\{x\} \uplus \mathcal{X},\{y\} \uplus \mathcal{Y}) & \longrightarrow & z(\mathcal{X},\mathcal{Y}) \parallel x(\varepsilon) \leftrightsquigarrow y(\varepsilon) \end{array}$$

We must also amend the rules for name restriction and parallel composition of configurations. This is where deadlock freedom breaks. First, we add an additional rule for name restriction of access points.

$$\textsc{NewAccess}\ \frac{\Delta;\Gamma,x : \mathsf{AP}\ S \vdash^{\phi} C}{\Delta;\Gamma \vdash^{\phi} (\nu x)C}$$

Second, we generalize the two parallel composition rules (Compose-1 and Compose-0) to a single rule (Compose-n) that allows an arbitrary number of channels to communicate across a boundary.

$$\textsc{Compose-}n\ \frac{\begin{array}{l}\Delta \vdash \Gamma = \Gamma_1 + \Gamma_2 \\ \Delta;\Gamma_1,x_1 : S,\ldots,x_n : S_n \vdash^{\phi} C \\ \Delta;\Gamma_2,x_1 : \overline{S_1},\ldots,x_n : \overline{S_n} \vdash^{\phi'} C'\end{array}}{\Delta;\Gamma,x_1 : S_1^\sharp,\ldots,x_n : S_n^\sharp \vdash^{\phi+\phi'} C \parallel C'}$$

The Compose-n rule is the functional analog of the rule for parallel composition in the linear π-calculus [16] and of a generalization of the Mix [12] and BiCut [2] rules in classical linear logic.

A weak form of progress holds in the presence of access points.

Theorem 14. *Let $\vdash C$, with $C \not\longrightarrow$, then every leaf of C is either: an access point, a buffer, a blocked thread, or a value.*

Adding access points exposes the difference between asynchronous and synchronous semantics. Referring back to the previous discussion of asynchrony (§3.1.1), here is an example of a term that reduces to a value according to our asynchronous semantics, but would deadlock under a synchronous semantics.

$$\begin{array}{l}\mathsf{let}\ z = \mathsf{new}\ \langle\rangle\ \mathsf{in} \\ \mathsf{let}\ z' = \mathsf{new}\ \langle\rangle\ \mathsf{in} \\ \mathsf{spawn}\ (\lambda\langle\rangle.\mathsf{let}\ x = \mathsf{accept}\ z\ \mathsf{in} \\ \qquad\qquad \mathsf{let}\ y = \mathsf{accept}\ z'\ \mathsf{in} \\ \qquad\qquad\quad \mathsf{send}\ \langle 0,x\rangle; \\ \qquad\qquad\quad \mathsf{let}\ \langle v,y\rangle = \mathsf{receive}\ y\ \mathsf{in}\ v) \\ \mathsf{let}\ x = \mathsf{accept}\ z'\ \mathsf{in} \\ \mathsf{let}\ y = \mathsf{accept}\ z\ \mathsf{in} \\ \quad \mathsf{send}\ \langle 0,x\rangle; \\ \quad \mathsf{let}\ \langle v,y\rangle = \mathsf{receive}\ y\ \mathsf{in}\ v\end{array}$$

With our asynchronous semantics, both sends happen followed by both receives, and the term reduces to the value 0. With a synchronous semantics both sends are blocked and the term is deadlocked.

Concurrent State. With access points we can implement concurrent state cells.

$$\begin{array}{rcl} \mathit{State}\ A & = & \mathsf{AP}\ (!A.\mathsf{End}) \\ \\ \mathit{newCell} & : & \forall\alpha^{(\bullet,\star)}.\langle\rangle \to \mathit{State}\ \alpha \\ \mathit{newCell}\ v & = & \mathsf{let}\ x = \mathsf{new}\ \langle\rangle\ \mathsf{in} \\ & & \quad \mathsf{spawn}\ (\lambda\langle\rangle.\mathsf{send}\ \langle v,\mathsf{accept}\ x\rangle); x \\ \mathit{put} & : & \forall\alpha^{(\bullet,\star)}.\mathit{State}\ \alpha \to \alpha \to \langle\rangle \\ \mathit{put}\ x\ v & = & \mathsf{let}\ \langle\_,\_\rangle = \mathsf{receive}\ (\mathsf{request}\ x)\ \mathsf{in} \\ & & \quad \mathsf{spawn}\ (\lambda\langle\rangle.\mathsf{send}\ \langle v,\mathsf{accept}\ x\rangle); \langle\rangle \\ \mathit{get} & : & \forall\alpha^{(\bullet,\star)}.\mathit{State}\ \alpha \to \alpha \\ \mathit{get}\ x & = & \mathsf{let}\ \langle v,\_\rangle = \mathsf{receive}\ (\mathsf{request}\ x)\ \mathsf{in} \\ & & \quad \mathsf{spawn}\ (\lambda\langle\rangle.\mathsf{send}\ \langle v,\mathsf{accept}\ x\rangle); v \end{array}$$

Nondeterminism. We can straightforwardly encode a nondeterministic choice by using an access point to generate a nondeterministic boolean value. Suppose that we have $\Delta;\Gamma \vdash M : T$ and $\Delta;\Gamma \vdash N : T$. The following term will nondeterministically choose between terms M and N:

$$\begin{array}{l}\mathsf{let}\ z = \mathsf{new}\ \langle\rangle\ \mathsf{in} \\ \quad \mathsf{spawn}\ (\lambda\langle\rangle.\mathsf{send}\ \langle \mathit{True},\mathsf{accept}\ z\rangle); \\ \quad \mathsf{spawn}\ (\lambda\langle\rangle.\mathsf{send}\ \langle \mathit{False},\mathsf{accept}\ z\rangle); \\ \quad \mathsf{let}\ \langle x,\_\rangle = \mathsf{receive}\ (\mathsf{request}\ z)\ \mathsf{in} \\ \qquad \mathsf{case}\ x\ \{\mathit{True} \to M; \mathit{False} \to N\}\end{array}$$

This thread does leave one thread waiting on $\mathsf{accept}\ z$. As z is cannot escpae this thread, it can be safely garbage collected; we have not included such a garbage collection equivalence, but it could be added easily.

Recursion. Recursion can in fact be encoded using access points. We have already seen that access points are expressive enough to simulate higher-order state. We can now use Landin's knot (backpatching) [17] to implement recursion. For instance, the following term loops forever:

$$\mathsf{let}\ x = \mathit{newCell}_{\langle\rangle\to\langle\rangle}\ (\lambda\langle\rangle.\langle\rangle)\ \mathsf{in}\ \mathit{put}\ \langle x,\lambda\langle\rangle.\mathit{get}\ x\ \langle\rangle\rangle; \mathit{get}\ x\ \langle\rangle$$

5. Links with Session Types

We have implemented Session Links, a session typed extension of the Links web programming language based on Fst. The source code is available online:

https://github.com/links-lang/links/tree/sessions

Links is a functional programming language for the web. From a single source program, Links generates code to run on all three tiers of a web application: the browser, the server, and the database. Links is a call-by-value language with support for ML-style type inference (extended with support for first-class polymorphism in a similar manner to GHC [31]). It incorporates a row-type system that is used for records, variants, and effects, and provides equirecursive types. Subkinding is used to distinguish base types from other types. This is important for enforcing the constraint that generated SQL queries must return a list of records whose fields are of base type [18].

In order to keep the presentation uniform and self-contained we use the concrete syntax of Fst rather than that of Links. However, all of the examples in the paper can be written directly in Links with essentially the same abstract syntax, modulo the fact that Links uses Hindley-Milner style type inference.

5.1 Design Choices

Before implementing session types for Links we considered a number of design choices. Linearity is crucial for implementing session types. Most existing functional languages (including vanilla Links) do not provide native support for linear types. We considered three broad approaches:

1. encode linearity using existing features of the programming language (as in Pucella and Tov's Haskell encoding of session types [26])
2. stratify the language so that the linear fragment of the language is separated out from the host language (as in Toninho et al's work [28])
3. bake linearity into the type system of the whole language (as in Last [11])

The appeal of the first approach is that it does not require any new language features, assuming the starting point is a language

with a sufficiently rich type system—that is able to conveniently encode parameterized monads [3], for instance. The second approach is somewhere in between. It allows a linear language to be embedded in an existing host language without disrupting the host language. The third approach requires linearity to pervade the whole of the type system, but opens up interesting possibilities for code reuse, for instance by using polymorphism over linearity [33] or using subkinding [22].

Given that we are in the business of developing our own programming language, over which we have full control, we decided to pursue the third option. We wanted to include most of the features of our language in the linear fragment, so we did not see a significant benefit in stratification. We did want to explore possibilities for code-reuse offered by the third approach.

Having chosen the third approach, we were presented with another choice regarding how to accommodate code reuse. Given that Links already supported subkinding [18] we elected to adopt the linear subkinding approach of Mazurak et al. [22].

An advantage of the LAST (and FST) approach to session typing is that channels are first class and hence support compositional programming. This is in contrast to the parameterized monad approach and approaches based on process calculi, in which channels are just names.

As an example, in FST with recursive types we can construct lists of channels, and, for instance, define a function to broadcast a value to a whole list of channels:

$$\begin{array}{rcl} \mathit{broadcast} & : & \forall \alpha^{\bullet,\star} \sigma^{\circ,\pi}.\alpha \to \mathit{LinList}\,(!\alpha.\sigma) \to \mathit{LinList}\,\sigma \\ \mathit{broadcast}\,v\,xs & = & \mathit{linMap}\,(\lambda x.\mathsf{send}\,\langle v, x\rangle)\,xs \end{array}$$

where *LinList A* is a linear list data type and *linMap* is the map operation over linear lists:

$$\begin{array}{rcl} \mathit{LinList}\,A & = & \mathsf{rec}\,\alpha^{(\circ,\star)}.[\mathit{Nil}; \mathit{Cons} : \langle A, \alpha\rangle] \\ \mathit{linMap} & : & \forall \alpha^{(\circ,\star)} \beta^{(\circ,\star)}.(\alpha \to \beta) \to \mathit{LinList}\,\alpha \to \mathit{LinList}\,\beta \\ \mathit{linMap}\,f\,xs & = & \mathsf{case}\,xs\,\{\mathit{Nil} \to \mathit{Nil} \\ & & \quad \mathit{Cons}\,\langle x, xs\rangle \to \mathit{Cons}\,\langle f\,x, \mathit{linMap}\,f\,xs\rangle\} \end{array}$$

An attendant drawback to having first-class channels is that one must explicitly rebind channels after each operation, often leading to verbose code. This is in contrast to the parameterized monad approach and approaches based on process calculi, which implicitly rebind channels after each communication.

In order to mitigate the need to explicitly rebind channels, we introduce process calculus style syntactic sugar inspired by previous work on the correspondence between classical linear logic and functional sessions [19, 20, 32]. To ease the job of writing a parser, we explicitly delimit process calculus style syntactic sugar with special brackets $\triangleleft - \triangleright$.

$$\begin{array}{rcl} \triangleleft x(y).Q \triangleright & \stackrel{\text{def}}{=} & \mathsf{let}\,\langle x, y\rangle = \mathsf{receive}\,x\,\mathsf{in}\,\triangleleft Q \triangleright \\ \triangleleft x[M].Q \triangleright & \stackrel{\text{def}}{=} & \mathsf{let}\,x = \mathsf{send}\langle M, x\rangle\,\mathsf{in}\,\triangleleft Q \triangleright \\ \triangleleft \ell\,x.Q \triangleright & \stackrel{\text{def}}{=} & \mathsf{let}\,x = \mathsf{select}\,\ell\,x\,\mathsf{in}\,\triangleleft Q \triangleright \\ \triangleleft \mathsf{offer}\,x\,\{\ell_i \to Q_i\}_i \triangleright & \stackrel{\text{def}}{=} & \mathsf{offer}\,x\,\{\ell_i(x) \to \triangleleft Q_i \triangleright\}_i \\ \triangleleft \{M\} \triangleright & \stackrel{\text{def}}{=} & M \end{array}$$

We let Q range over process calculus style terms. The desugaring of input, output, selection, and branching is direct. The $\{-\}$ brackets allow values to be returned from the tail of a process calculus expression. As an example, we can more concisely rewrite the simple calculator server of (§1) as follows:

$$\begin{array}{l} \mathit{sugarCalc} = \lambda c.\triangleleft\,\mathsf{offer}\,c\,\{ \\ \qquad \mathit{Add} \to c(x).c(y).c[x+y].\{\langle\rangle\} \\ \qquad \mathit{Neg} \to c(x).c[-x].\{\langle\rangle\}\}\triangleright \end{array}$$

In general, the syntactic sugar allows us to take advantage of a process-calculus style when a lot of rebinding is going on and then switch back to a functional style for compositional programming.

5.2 Type Reconstruction

Vanilla links provides automatic type reconstruction as in ML; we would hope to extend this approach to Session Links. Alas, as a consequence of the treatment of application, the types of higher-order functions in FST are not uniquely determined by their use; thus, type reconstruction in Session Links is necessarily incomplete. As an example, consider the following FST term, which implements function composition:

$$\Lambda\alpha_1, \alpha_2, \alpha_3.\lambda f^{\alpha_1 \to^{Y_1} \alpha_2}.\lambda g^{\alpha_2 \to^{Y_2} \alpha_3}.\lambda x^{\alpha_1}.f\,(g\,x).$$

This term is well-typed for arbitrary choices for Y_1 and Y_2, giving four distinct terms, each with distinct types, and Session Links type reconstruction must make an arbitrary choice among them. We could imagine repairing this by adding additional forms for quantification over linearity [33], or by introducing subtyping for the function types (as in LAST). Session Links does not have complete type inference; we prefer $\tau \to^{\bullet} \tau'$ to $\tau \to^{\circ} \tau'$ when computing the types of expressions. This has the advantage of working for existing functional code, but is arguably less general (as a value of type $\tau \to^{\circ} \tau'$ can be constructed from a value of type $\tau \to^{\bullet} \tau'$, but not vice versa).

Functions. We also observe that, even in cases where Session Links expressions have principal types, they are not always as general as we might hope. Consider the curried pair constructor; we have two ways to write this, corresponding to the following two FST terms:

$$\begin{array}{rcl} p & = & \Lambda\alpha^{\mathit{Type}(\circ,\star)}.\Lambda\beta^{\mathit{Type}(\circ,\star)}.\lambda^{\bullet}x^{\alpha}.\lambda^{\circ}y^{\beta}.\langle x, y\rangle \\ q & = & \Lambda\alpha^{\mathit{Type}(\bullet,\star)}.\Lambda\beta^{\mathit{Type}(\circ,\star)}.\lambda^{\bullet}x^{\alpha}.\lambda^{\bullet}y^{\beta}.\langle x, y\rangle \end{array}$$

We can distinguish between the corresponding Session Links terms, as Session Links distinguishes between $\lambda^{\bullet}$ and λ°. However, note that neither of these definitions is as generic as we might hope. The first accepts more types for α than the second, but the inner function is always linear, whether or not the instantiation of α requires it. The second constructs an unlimited inner function, but at the cost of requiring that α be unlimited. There is no FST term that can behave as either, depending on α. In future we plan to explore the use of kind polymorphism (and in particular polymorphism over linearity) to mitigate the issues described above.

Unlike general subtyping, implicit subkinding in Links does not significantly affect the implementation of type inference in the source language as there are no higher kinds. Aside from the issues discussed above, our limited use of subkinding seems pleasantly well-behaved in practice.

6. Related Work

Session types were originally proposed by Honda [13], and later extended by Takeuchi et al. [27] and by Honda et al. [14]. Honda's system relies on a substructural type system (in which channels cannot be duplicated or discarded). Polyadic linear π-calculus was introduced by Kobayashi et al. [16], who additionally describe a notion of linearized channels similar to the sequencing provided by session types. Kobayashi [15] uses variant types to express choice for linearized channels. Dardha et al. [10] show the correctness of this encoding and extend it to polymorphism and subtyping; Dardha [9] extends the encoding to recursive and replicated session types.

Abramsky [1] proposed a lambda calculus term language for intuitionistic linear logic. Vasconcelos et al. [30] develop a language that integrates session-typed communication primitives and

a linear functional language. Gay and Vasconcelos [11] extend the approach to describe asynchronous communication with statically-bounded buffers. Mazurak and Zdancewic's lolliproc language [21] extends a linear λ-calculus with control operators and is capable of expressing session types. Mazurak et al. [22] demonstrate the use of subkinding for unlimited types in a linear lambda calculus. Toninho et al. [28] give a stratified language including both an intuitionistic functional language and a linear session-typed process calculus. Their system has been implemented as the language SILL [25]. Pucella and Tov [26] give an implementation of session types in Haskell, built on an existing unsafe communication mechanism. Their approach relies on capturing the session state in a parameterized monad [3], and encoding session types and duality using the Haskell class system. They also present a core functional calculus with communication primitives, and prove the safety of their approach when treated as a library for this core language.

7. Conclusion and Future Work

We have presented an account of lightweight functional session types, extending our core session-typed linear λ-calculus [20] with: the row typing of our core language for Links [18], the subkinding for linearity of Mazurak and Zdancewic's lightweight linear types [22], and the asynchrony and access points of Gay and Vasconcelos's linear type theory for asynchronous session types [11]. We conclude by highlighting several areas of future work.

There is a large gap between variants of FST with and without access points. We would like to investigate abstractions that add some of the expressive power of access points, but are better behaved. In particular, it would be interesting to explore richer type systems for enforcing deadlock-freedom and race freedom, while allowing some amount of stateful concurrency. More immediately, it would also be natural to exploit the existing effect type system of Links to statically enforce desirable properties, for instance, by associating the use of access points with a particular effect type.

In this work, we have considered the extension of FST with general recursive types. We would also like to develop a theory of total recursive and corecursive session types based on Baelde's work on fixed point combinators for classical linear logic [4], and relate this to total recursive and corecursive data types and to Toninho et al.'s account of corecursive session types [29].

The connections between continuations and concurrency are well known. Indeed, our strong normalization argument relies on a CPS transformation. It would be interesting to investigate the exact relationship between GV, Mazurak and Zdancewic's lolliproc [21], and other calculi with first-class continuations such as Parigot's $\lambda\mu$-calculus [23].

References

[1] S. Abramsky. Computational interpretations of linear logic. *Theor. Comput. Sci.*, 111(1&2):3–57, 1993.

[2] S. Abramsky, S. J. Gay, and R. Nagarajan. Interaction categories and the foundations of typed concurrent programming. In *Deductive Program Design*. Springer, 1996.

[3] R. Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19(3-4):335–376, 2009.

[4] D. Baelde. Least and greatest fixed points in linear logic. *ACM Trans. Comput. Logic*, 13(1):2:1–2:44, Jan. 2012.

[5] H. P. Barendregt. *The Lambda Calculus Its Syntax and Semantics*, volume 103. North Holland, revised edition, 1984.

[6] G. Bernardi and M. Hennessy. Using higher-order contracts to model session types. *CoRR*, abs/1310.6176, 2013.

[7] V. Bono and L. Padovani. Typing copyless message passing. *Logical Methods in Computer Science*, 8(1), 2012.

[8] M. Carbone and S. Debois. A graphical approach to progress for structured communication in web services. In *ICE*, 2010.

[9] O. Dardha. Recursive session types revisited. In *BEAT*, volume 162 of *EPTCS*, pages 27–34, 2014.

[10] O. Dardha, E. Giachino, and D. Sangiorgi. Session types revisited. In *PPDP*. Springer, 2012.

[11] S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(01):19–50, 2010.

[12] J. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[13] K. Honda. Types for dyadic interaction. In *CONCUR*. Springer, 1993.

[14] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*. Springer, 1998.

[15] N. Kobayashi. Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*. Springer, 2002.

[16] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the π-calculus. In *POPL*. ACM, 1996.

[17] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[18] S. Lindley and J. Cheney. Row-based effect types for database integration. In B. C. Pierce, editor, *TLDI*. ACM, 2012.

[19] S. Lindley and J. G. Morris. Sessions as propositions. In *PLACES*, 2014.

[20] S. Lindley and J. G. Morris. A semantics for propositions as sessions. In *ESOP*. Springer, 2015. To appear.

[21] K. Mazurak and S. Zdancewic. Lolliproc: to concurrency from classical linear logic via Curry-Howard and control. In P. Hudak and S. Weirich, editors, *ICFP*. ACM, 2010.

[22] K. Mazurak, J. Zhao, and S. Zdancewic. Lightweight linear types in System F°. In A. Kennedy and N. Benton, editors, *TLDI*. ACM, 2010.

[23] M. Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In *LPAR*. Springer, 1992.

[24] J. A. Pérez, L. Caires, F. Pfenning, and B. Toninho. Linear logical relations and observational equivalences for session-based concurrency. *Inf. Comput.*, 239:254–302, 2014.

[25] F. Pfenning, L. Caires, B. Toninho, and D. Griffith. From linear logic to session-typed concurrent programming. Tutorial at POPL 2015, 2015.

[26] R. Pucella and J. A. Tov. Haskell session types with (almost) no class. In A. Gill, editor, *Haskell*. ACM, 2008.

[27] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE*. Springer, 1994.

[28] B. Toninho, L. Caires, and F. Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *ESOP*. Springer, 2013.

[29] B. Toninho, L. Caires, and F. Pfenning. Corecursion and non-divergence in session-typed processes. In *TGC*. Springer, 2014.

[30] V. T. Vasconcelos, S. J. Gay, and A. Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006.

[31] D. Vytiniotis, S. Weirich, and S. L. Peyton Jones. FPH: first-class polymorphism for Haskell. In J. Hook and P. Thiemann, editors, *ICFP*. ACM, 2008.

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

[33] D. Walker. Substructural Type Systems. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 1. MIT Press, 2005.