12

Lightweight Functional Session Types

Sam Lindley and J. Garrett Morris

University of Edinburgh, Edinburgh, UK

Abstract

Row types provide an account of extensibility that combines well with parametric polymorphism and type inference. We discuss the integration of row types and session types in a concurrent functional programming language, and how row types can be used to describe extensibility in session-typed communication.

12.1 Introduction

In prior work, we have developed a core linear λ-calculus with session types called GV [13]. GV is inspired by a functional language with session types developed by Gay and Vasconcelos [7], which we term LAST (for Linear Asynchronous Session Types), and by the propositions-as-types correspondence between session types and linear logic first introduced by Caires and Pfenning [4] and later adapted to the classical setting by Wadler [23]. We have given direct proofs of deadlock freedom, determinism, and termination for GV. We have also given semantics-preserving translations between GV and Wadler's process calculus CP, showing a strong connection between GV's small-step operational semantics and cut elimination in classical linear logic.

In this article, we demonstrate that we can build practical languages based on the primitives and properties of GV. We introduce a language, FST, that extends GV with polymorphism, row types, and subkinding, integrating linear and unlimited data types. FST, while more expressive, is still deadlock-free, deterministic, and terminating. We consider several extensions of FST. Recursion and recursive session types support the definition of long-running services and repeated behavior. Adding recursion and recursive session types results in a system that is no longer terminating, but is still deadlock free and deterministic. Access points support a more flexible mechanism for session initiation. Adding access points results in a system that is not deadlock-free, deterministic, or terminating, but that still satisfies subject reduction and a weak form of progress.

Outline. The article proceeds as follows. Section 12.2 presents some examples illustrating FST and its extensions. Section 12.3 gives a formal account of FST, a linear variant of System F, incorporating polymorphism, row-typing, subkinding, and session types.

Section 12.4 explores extensions of FST with recursion, recursive types, and access points, and demonstrates the expressivity of access points with encodings of state cells, nondeterministic choice, and recursion.

Section 12.5 describes 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 12.6 concludes.

In this version of the article, we focus on the FST type system, and omit the formal semantics and statements of correctness. An extended version including the formal semantics and correctness proofs is available online [15].

12.2 A First Look

Before giving a formal account of the syntax and type system of FST, we present some simple examples of programming in FST. We use a desktop calculator as a running example. Despite its simplicity, it will motivate the features of FST.

A One-Shot Calculator Server. We begin with a process that implements a calculator server. We specify it as a function of one channel, c, on which it will communicate with a user of the calculator.

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

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

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

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 $\&\{\ell : S, \dots, \ell_n : S_n\}$ denotes offering an n-ary choice, with the behavior of the i^{th} branch given by S_i.

Next, we consider a client for the calculator server:

$$\mathsf{user}_1\ c = \mathbf{let}\ c = \mathbf{select}\ \mathsf{Add}\ c\ \mathbf{in}\ \mathbf{let}\ \langle x, c\rangle = \mathbf{receive}\ (\mathbf{send}\ \langle 19, \mathbf{send}\ \langle 23, c\rangle\rangle)\ \mathbf{in}\ x$$

Like calc, the user_1 function is passed the channel on which it communicates with the calculator. It begins by selecting the Add behavior, which is compatible with the choice offered by calc. Its subsequent behavior is unsurprising. We could give the channel a type dual to that provided by the calculator:

$$\mathsf{user}_1 : \oplus\{\mathsf{Add} : !\mathsf{Int}.!\mathsf{Int}.?\mathsf{Int}.\mathsf{End}, \mathsf{Neg} : !\mathsf{Int}.?\mathsf{Int}.\mathsf{End}\} \to \mathsf{Int}$$

However, this type overspecifies the behavior of user_1 as the Neg branch is unused in the definition of user_1. In FST, we can use row polymorphism to abstract over the irrelevant labels in a choice, as follows:

$$\mathsf{user}_1 : \forall\rho. \oplus \{\mathsf{Add} : !\mathsf{Int}.!\mathsf{Int}.?\mathsf{Int}.\mathsf{End}; \rho\} \to \mathsf{Int}$$

This type specifies that the argument to user_1 may be instantiated to any session type that offers a choice of Add with a suitable behavior along with arbitrary other choices. FST includes explicit type abstractions and type annotations on bound variables; we omit both in the examples in order to improve readability. Our concrete implementation of FST in Links, is able to reconstruct omitted types and type abstractions using a fairly standard Hindley-Milner-style type inference algorithm.

We can plug the calculator server and the user together as follows

$$\mathbf{let}\ c = \mathbf{fork}\ \mathsf{calc}\ \mathbf{in}\ \mathsf{user}_1\ c$$

yielding the number 42. The **fork** primitive creates a new child process and a channel through which it can communicate with its parent process.

Recursive Session Types. The one-shot calculator server allows only one operation to be performed before the communication is exhausted. If we add support for recursive session types, then we can define a calculator that allows an arbitrary number of operations to be performed. In order to make the example more interesting, we define a calculator server with a memory.

$$
\begin{array}{l}
\mathsf{calc}_{\mathbf{rec}} : \mathsf{Int} \to (\mathbf{rec}\ \sigma.\&\{\mathsf{Add} : ?\mathsf{Int}.?\mathsf{Int}.!\mathsf{Int}.\sigma, \\
\qquad\qquad\qquad\qquad \mathsf{Neg} : ?\mathsf{Int}.!\mathsf{Int}.\sigma, \\
\qquad\qquad\qquad\qquad \mathsf{M}^{+} : ?\mathsf{Int}.\sigma, \\
\qquad\qquad\qquad\qquad \mathsf{MR} : !\mathsf{Int}.\sigma \\
\qquad\qquad\qquad\qquad \mathsf{Stop} : \mathsf{End}\}) \to \mathsf{End} \\
\mathsf{calc}_{\mathbf{rec}}\ m\ c = \mathbf{offer}\ c\ \{\mathsf{Add}\ c\ \to \mathbf{let}\ \langle x,c\rangle = \mathbf{receive}\ c\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ \langle y,c\rangle = \mathbf{receive}\ c\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{calc}_{\mathbf{rec}}\ m\ (\mathbf{send}\ \langle x+y,c\rangle) \\
\qquad\qquad\qquad\qquad\quad \mathsf{Neg}\ c\ \to \mathbf{let}\ \langle x,c\rangle = \mathbf{receive}\ c\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{calc}_{\mathbf{rec}}\ m\ (\mathbf{send}\ \langle -x,c\rangle) \\
\qquad\qquad\qquad\qquad\quad \mathsf{M}^{+}\ c\ \to \mathbf{let}\ \langle x,c\rangle = \mathbf{receive}\ c\ \mathbf{in}\ \mathsf{calc}_{\mathbf{rec}}\ (m+x)\ c \\
\qquad\qquad\qquad\qquad\quad \mathsf{MR}\ c\ \to \mathbf{let}\ c = \mathbf{send}\ \langle m,c\rangle\ \mathbf{in}\ \mathsf{calc}_{\mathbf{rec}}\ m\ c\} \\
\qquad\qquad\qquad\qquad\quad \mathsf{Stop}\ c \to c\}
\end{array}
$$

The idea is that selecting M^{+} adds a number to that currently stored in memory and MR reads the current value of the memory. A user must now explicitly select Stop in order to terminate communication with the server.

$$
\begin{array}{l}
\mathsf{user}_2 : \forall \rho \rho'. \oplus \{\mathsf{Add} : !\mathsf{Int}.!\mathsf{Int}.?\mathsf{Int}.\oplus\{\mathsf{Stop} : \mathsf{End}; \rho\}; \rho'\} \to \mathsf{Int} \\
\mathsf{user}_2\ c = \mathbf{let}\ \langle x,c\rangle = \mathbf{receive}\ (\mathbf{send}\ \langle 19, \mathbf{send}\ \langle 23, \mathbf{select}\ \mathsf{Add}\ c\rangle\rangle)\ \mathbf{in} \\
\qquad\qquad \mathbf{select}\ \mathsf{Stop}\ c; x
\end{array}
$$

With the row variables instantiated appropriately, we can plug user_2 and the recursive calculator together

$$\mathbf{let}\ c = \mathbf{fork}\ \mathsf{calc}_{\mathbf{rec}}\ 0\ \mathbf{in}\ \mathsf{user}_2\ c$$

again yielding 42.

The examples we have seen so far could be implemented using subtyping instead of row polymorphism. We now consider a function that cannot be implemented with subtyping. Suppose we wish to abstract over the memory add operation. We define a function that can be used to communicate with any calculator server that supports M^{+} and arbitrary other operations.

$$
\begin{array}{l}
\mathsf{mAdd} : \forall \rho.\mathsf{Int} \to (\mathbf{rec}\ \sigma. \oplus \{\mathsf{M}^{+} : !\mathsf{Int}.\sigma; \rho\}) \to (\mathbf{rec}\ \sigma. \oplus \{\mathsf{M}^{+} : !\mathsf{Int}.\sigma; \rho\}) \\
\mathsf{mAdd}\ n\ c = \mathbf{send}\ \langle n, \mathbf{select}\ \mathsf{M}^{+}\ c\rangle
\end{array}
$$

The key feature of this function is that the row variable ρ appears both contravariantly (inside the second argument) and covariantly (inside the return type) in the type of mAdd. Thus, in a system with subtyping but without row typing, one would have to explicitly instantiate ρ, ruling out an extensible calculator server implementation. Let us use mAdd to define a client that invokes multiple calculator operations.

```
user₃ :
   $\forall \rho\rho'\rho''.$
      $\oplus\{\mathsf{M}^+ : !\mathsf{Int}.\oplus\{\mathsf{M}^+ : !\mathsf{Int}.\oplus\{\mathsf{MR} : ?\mathsf{Int}.\oplus\{\mathsf{Stop} : \mathsf{End};\rho\};\rho'\};\rho''\}\} \to \mathsf{Int}$
user₃ $c$ = let $c$ = select MR (mAdd 19 (mAdd 23 $c$)) in
           let $\langle x, c\rangle$ = receive $c$ in
           select Stop $c$;$x$
```

We can plug user₃ and the recursive calculator together as before

$$\textbf{let } c = \textbf{fork } \mathsf{calc}_{\mathbf{rec}}\ 0 \textbf{ in } \mathsf{user}_3\ c$$

again yielding 42.

Access Points. A key limitation of the examples we have seen so far is that they allow only one user to connect to a calculator server at a time. 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 non-deterministically. We now adapt our calculator server to synchronize on an access point instead of a fixed channel:

```
calc_AP : $\forall\alpha.\mathsf{Int} \to \mathsf{AP}$ ($\&\{$Add : ?Int.?Int.!Int.End,
                          Neg : ?Int.!Int.End,
                          $\mathsf{M}^+$ : ?Int.End,
                          MR : !Int.End$\}) \to \alpha$
calc_AP $m$ $a$ = let $c$ = accept $a$ in
              offer $c$ {
                 Add $c$ →let $\langle x, c\rangle$ = receive $c$ in
                          let $\langle y, c\rangle$ = receive $c$ in
                          let $c$ = send $\langle x+y, c\rangle$ in calc_AP $m$ $a$
                 Neg $c$ →let $\langle x, c\rangle$ = receive $c$ in
                          let $c$ = send $\langle -x, c\rangle$ in calc_AP $m$ $a$
                 $\mathsf{M}^+$ $c$ → let $\langle x, c\rangle$ = receive $c$ in calc_AP $(m+x)$ $a$
                 MR $c$ → let $c$ = send $\langle m, c\rangle$ in calc_AP $m$ $a$}
```

Unlike $\mathsf{calc}_{\mathbf{rec}}$, this calculator server never stops; rather, it will persist until the access point is no longer accessible by any client code, at which point it may be garbage collected. As $\mathsf{calc}_{\mathbf{rec}}$ never returns, it is polymorphic in its return type. In general, an access point a has type AP S for some session type S. The expression **accept** a returns an end point of type S and **request** a returns an end point of type $\overline{S}$.

We can connect our original user to $\mathsf{calc}_{\mathsf{AP}}$. We use the **new** operator to create a fresh access point and the **spawn** operator to create child threads (without any shared channels).

$$\mathbf{let}\ a = \mathbf{new}\ \mathbf{in}\ \mathbf{spawn}\ (\lambda\langle\rangle.\mathsf{calc}_{\mathsf{AP}}\ 0\ a);\mathsf{user}_1\ (\mathbf{request}\ a)$$

The result of evaluation is again 42. More interestingly, we can connect multiple clients to the same server concurrently.

$$\begin{array}{l}
\mathbf{let}\ a = \mathbf{new}\ \mathbf{in} \\
\mathbf{let}\ \mathsf{mAdd}\ n\ a = \mathbf{send}\ \langle n, \mathbf{select}\ \mathsf{M}^{+}\ (\mathbf{request}\ a)\rangle\ \mathbf{in} \\
\mathbf{let}\ \mathsf{mRecall}\ a = \mathbf{let}\ \langle x, c\rangle = \mathbf{receive}\ (\mathbf{select}\ \mathsf{M}^{+}\ (\mathbf{request}\ a))\ \mathbf{in} \\
\mathbf{spawn}\ (\lambda\langle\rangle.\mathsf{calc}_{\mathsf{AP}}\ 0\ a); \\
\mathbf{spawn}\ (\lambda\langle\rangle.\mathsf{mAdd}\ 19\ (\mathbf{request}\ a)); \\
\mathbf{spawn}\ (\lambda\langle\rangle.\mathsf{mAdd}\ 23\ (\mathbf{request}\ a)); \\
\mathsf{mRecall}\ a
\end{array}$$

The result of evaluating this code is non-deterministic. Depending on the scheduler it may yield 0, 19, 23, or 42.

12.3 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 [13], with the row typing and subkinding of our previous core language for Links [11], and the similar approach to subkinding for linearity of Mazurak et al's lightweight linear types [17].

As our focus is programming with session types rather than their logical connections, we make some simplifications compared to our earlier work [13]. Specifically, we have a single unlimited self-dual type of closed channels, and we omit the operation for linking channels together.

12.3.1 Syntax

To avoid duplication and keep the concurrent semantics of FST 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. The syntax of types and kinds is given in Figure 12.1. The function type $A \to^Y B$ takes an argument of type A and returns a value of type B and has linearity Y. (We write $A \to B$ as an abbreviation for $A \to^\bullet B$.) 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. Intuitively, a row type represents a mapping from labels to ordinary types. In fact, rows also track absent labels, which are, for instance, needed to type polymorphic record extension (a record can only be extended with labels that are not already present). 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

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

Figure 12.1 Syntax of types and kinds.

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 syntactic duality operation on type variables extends to a semantic duality operation on session types and is lifted homomorphically to session row types, and session presence types:

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

Kinds. Types are classified by kinds. Ordinary types have kind Type. Row types R have kind $\mathsf{Row}_{\mathscr{L}}$ where $\mathscr{L}$ 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 [11] and the system of Mazurak et al. on lightweight linear types [17]. 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 $\mathsf{Row}_{\mathscr{L}}(\circ, \pi)$ is inhabited by row types of linear session type and the kind $\mathsf{Presence}(\bullet, \star)$ by presence types of unlimited unconstrained ordinary types.

By convention we use α for ordinary type variables or for type variables of unspecified kind, ρ for type variables of row kind, and θ for type variables of presence kind. We sometimes omit the primary kind, either inferring it from context or assuming a default of Type. For instance, we write $\alpha^{\bullet,\star}$ instead of $\alpha^{\mathsf{Type}(\bullet,\star)}$.

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

$$\frac{}{\vdash \bullet \leq \circ} \qquad \frac{}{\vdash \pi \leq \star} \qquad \frac{\vdash Y \leq Y' \qquad \vdash Z \leq Z'}{\vdash K(Y,Z) \leq K(Y',Z')}$$

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.

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

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

Terms. The syntax of terms and values is given in Figure 12.2. 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 forms **let** $\langle\rangle \leftarrow M$ **in** N and **let** $\langle \ell = x; y \rangle \leftarrow M$ **in** N, the latter of which binds the value labeled by ℓ to x and the remainder of the record to y. Conventional projections $M.\ell$

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

Figure 12.2 Syntax of terms and values.

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 with $\textbf{case}\ L\ \{\ell\ x \to M; y \to N\}$. Hypothetical empty variants are eliminated with $\textbf{case}_{\perp}\ L$.

Concurrency. The concurrency features of FST are provided by special constants. The term $\textbf{send}\ \langle V, W\rangle$ sends V along channel W, returning the updated channel. The term $\textbf{receive}\ W$ receives a value along channel W, and returns a pair of the value and the updated channel. The term $\textbf{fork}\ (\lambda x.M)$ 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:

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

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

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

and

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

where we let χ range over sequences of cases:

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

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

12.3.2 Typing and Kinding Judgments

The kinding rules are given in Figure 12.3. 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

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

$$\textsc{Function}\quad \frac{\Delta \vdash A : \mathsf{Type}(Y,\star) \qquad \Delta \vdash B : \mathsf{Type}(Y',\star)}{\Delta \vdash A \to^{Y''} B : \mathsf{Type}(Y'',\star)} \qquad \textsc{ForAll}\quad \frac{\Delta, \alpha : K(\bullet,Z) \vdash A : \mathsf{Type}(Y,\star)}{\Delta \vdash \forall \alpha^{K(Y',Z)}.A : \mathsf{Type}(Y,\star)}$$

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

$$\textsc{Input}\quad \frac{\begin{array}{c}\Delta \vdash A : \mathsf{Type}(Y,\star)\\ \Delta \vdash S : \mathsf{Type}(Y',\pi)\end{array}}{\Delta \vdash ?A.S : \mathsf{Type}(\circ,\pi)} \qquad \textsc{Output}\quad \frac{\begin{array}{c}\Delta \vdash A : \mathsf{Type}(Y,\star)\\ \Delta \vdash S : \mathsf{Type}(Y',\pi)\end{array}}{\Delta \vdash !A.S : \mathsf{Type}(\circ,\pi)} \qquad \textsc{End}\quad \frac{}{\Delta \vdash \mathsf{End} : \mathsf{Type}(\bullet,\pi)}$$

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

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

$$\textsc{TyVar}\quad \frac{\alpha : K(Y,Z) \in \Delta}{\Delta \vdash \alpha : K(Y,Z)} \qquad \textsc{DualTyVar}\quad \frac{\alpha : K(Y,\pi) \in \Delta}{\Delta \vdash \overline{\alpha} : K(Y,\pi)} \qquad \textsc{Upcast}\quad \frac{\vdash J \leq J' \qquad \Delta \vdash T : J}{\Delta \vdash T : J'}$$

Figure 12.3 Kinding rules.

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 $\mathsf{Row}_{\mathcal{L}}$ 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 $\mathsf{Row}_{\mathcal{L}}(Y,Z)$ for any label set $\mathcal{L}$, linearity Y, and restriction Z. The use of disjoint union in the ExtendRow rule ensures that row types have distinct labels. A row type can only be used to build a record or variant if it has kind $\mathsf{Row}_{\emptyset}$; this constraint ensures that any absent labels in an open row type are mentioned explicitly.

In Figure 12.4 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

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

$$\textsc{L-Empty}\quad \frac{}{\Delta \vdash \cdot : Y} \qquad\qquad \textsc{L-Extend}\quad \frac{\Delta \vdash \Gamma : Y \qquad \Delta \vdash A : K(Y,Z)}{\Delta \vdash (\Gamma, x : A) : Y}$$

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

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

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

Figure 12.4 Linearity of contexts and context splitting.

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 12.5. 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 12.6.

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 [13], inspired by Kobayashi [8], we encode selection and choice using variant types.

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

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

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

$$\textsc{LinLam}\quad \frac{\Delta;\Gamma, x : A \vdash M : B}{\Delta;\Gamma \vdash \lambda^{\circ} x^{A}.M : A \to^{\circ} B} \qquad \textsc{UnlLam}\quad \frac{\begin{array}{c}\Delta \vdash \Gamma : \bullet \\ \Delta;\Gamma, x : A \vdash M : B\end{array}}{\Delta;\Gamma \vdash \lambda^{\bullet} x^{A}.M : A \to^{\bullet} B} \qquad \textsc{App}\quad \frac{\begin{array}{c}\Delta;\Gamma_1 \vdash L : A \to^{Y} B \\ \Delta;\Gamma_2 \vdash M : A\end{array}}{\Delta;\Gamma_1 + \Gamma_2 \vdash L\, M : B}$$

$$\textsc{PolyLam}\quad \frac{\Delta, \alpha :: K(\bullet, Z);\Gamma \vdash V : A \qquad \alpha \notin \mathrm{ftv}(\Gamma)}{\Delta;\Gamma \vdash \Lambda \alpha^{K(Y,Z)}.V : \forall \alpha^{K(Y,Z)}.A} \qquad \textsc{PolyApp}\quad \frac{\begin{array}{c}\Delta;\Gamma \vdash M : \forall \alpha^{K(Y,Z)}.A \\ \Delta \vdash T :: K(Y,Z)\end{array}}{\Delta;\Gamma \vdash M\, T : A[\alpha := T]}$$

$$\textsc{Unit}\quad \frac{\Delta \vdash \Gamma : \bullet}{\Delta;\Gamma \vdash \langle\rangle : \langle\rangle} \qquad \textsc{LetUnit}\quad \frac{\Delta;\Gamma_1 \vdash M : \langle\rangle \qquad \Delta;\Gamma_2 \vdash N : B}{\Delta;\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ \langle\rangle \leftarrow M\ \mathbf{in}\ N : B} \qquad \textsc{CaseZero}\quad \frac{\Delta;\Gamma \vdash L : []}{\Delta;\Gamma \vdash \mathbf{case}_{\bot} L : B}$$

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

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

Figure 12.5 Typing rules.

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

Figure 12.6 Type schemas for constants.

The encoding of **select** uses **fork** in order to generate a fresh channel of the continuation type. In the implementation of 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.

Semantics. In the extended version of this article [15] we give an asynchronous small-step operational semantics for FST. Following Gay and Vasconcelos [7], 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 process and buffers. This semantics differs from our previous work on GV [13] in that is relies on explicit buffers, allowing asynchrony between the sending and receiving of a message, and it uses standard β-reduction instead of weak explicit substitutions [10]. FST, like GV but unlike LAST, is deadlock-free, deterministic, and terminating.

12.4 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 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 on LAST (and previous work on session-typed π-calculi), 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 Links.

12.4.1 Recursion

The grammar of session types we have presented so far is rather limited; for example, it cannot express repeated behavior. As illustrated in Section 12.2, we can use recursive session types to define a calculator that supports multiple calculations. In order to support this kind of example, we can straightforwardly extend FST with equi-recursive types. We add a kinding rule for recursive types and identify each recursive type with its unrolling.

$$\textsc{Rec}\ \frac{\Delta, \alpha : \mathsf{Type}(Y,Z) \vdash A : \mathsf{Type}(Y,Z)}{\Delta \vdash \mathbf{rec}\ \alpha^{Y,Z}.A : \mathsf{Type}(Y,Z)} \qquad \mathbf{rec}\ \alpha^{Y,Z}.A = A[\mathbf{rec}\ \alpha^{Y,Z}.A/\alpha]$$

It is well-known [2, 3] 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 $\mathbf{rec}\ \sigma^{\circ,\pi}.?\sigma.\sigma$. The dual of this type is not $\mathbf{rec}\ \sigma^{\circ,\pi}.!\sigma.\sigma$, as one would obtain by taking the dual of the body of the recursive type directly, but is $\mathbf{rec}\ \sigma^{\circ,\pi}.!\overline{\sigma}.\sigma$ instead.

Bernardi and Hennessy [2] point out that even existing definitions that correctly handle the above instance of recursion variables appearing inside a carried type often fail for other examples. The underlying difficulty arises from attempting to define duality in a setting in which the duality operator may not be applied to atomic type variables. Bernardi and Hennessy show that is is possible to give a correct definition in such a setting, but we prefer the more compositional definition that arises naturally when one admits duals of atomic type variables [16] (something that we want anyway as our calculus is polymorphic).

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

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:

$$\Sigma(\mathbf{fix}) = \forall\alpha^{\bullet,\star}.\forall\beta^{\bullet,\star}.((\alpha \rightarrow^{\bullet} \beta) \rightarrow^{\bullet} (\alpha \rightarrow^{\bullet} \beta)) \rightarrow^{\bullet} (\alpha \rightarrow^{\bullet} \beta)$$

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.

12.4.2 Access Points

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

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

A process M is spawned with **spawn** M, where M is a thunk that returns an arbitrary unlimited value; we can define **spawn** in terms of **fork** and vice versa:

$$\begin{aligned} \textbf{spawn}\, M &\stackrel{\text{def}}{=} (\lambda x^{\mathsf{End}}.\langle\rangle)(\textbf{fork}\ (\lambda x^{\mathsf{End}}.M\ \langle\rangle)) \\ \textbf{fork}\, M &\stackrel{\text{def}}{=} \textbf{let}\, z = \textbf{new}\ \langle\rangle\ \textbf{in spawn}\ (\lambda x.M\ (\textbf{accept}\, z)); \textbf{request}\, z \end{aligned}$$

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.

Adding access points exposes the difference between asynchronous and synchronous semantics. Here is an example of a term that reduces to a value under an asynchronous semantics, but deadlocks under a synchronous semantics.

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

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

Shared State. With access points we can implement shared state cells.

$$\mathsf{State}\, A = \mathsf{AP}\ (!A.\mathsf{End})$$

$$\begin{array}{l} \mathsf{newCell} : \forall \alpha^{\bullet,\star}.\langle\rangle \to \mathsf{State}\ \alpha \\ \mathsf{newCell}\ v = \textbf{let}\, x = \textbf{new}\ \langle\rangle\ \textbf{in spawn}\ (\lambda\langle\rangle.\textbf{send}\ \langle v, \textbf{accept}\, x\rangle); x \end{array}$$

$$\begin{array}{l} \mathsf{put} : \forall \alpha^{\bullet,\star}.\mathsf{State}\ \alpha \to \alpha \to \langle\rangle \\ \mathsf{put}\ x\ v = \textbf{let}\ \langle \_, \_\rangle = \textbf{receive}\ (\textbf{request}\, x)\ \textbf{in spawn}\ (\lambda\langle\rangle.\textbf{send}\ \langle v, \textbf{accept}\, x\rangle); \langle\rangle \end{array}$$

$$\begin{array}{l} \mathsf{get} : \forall \alpha^{\bullet,\star}.\mathsf{State}\ \alpha \to \alpha \\ \mathsf{get}\ x = \textbf{let}\ \langle v, \_\rangle = \textbf{receive}\ (\textbf{request}\, x)\ \textbf{in spawn}\ (\lambda\langle\rangle.\textbf{send}\ \langle v, \textbf{accept}\, x\rangle); v \end{array}$$

Nondeterminism. We can straightforwardly encode 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:

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

One process is left waiting on **accept** z. However, as z cannot escape, this process can be safely garbage collected.

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 (back-patching) [9] to implement recursion. For instance, the following term loops forever:

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

12.5 Links with Session Types

Version 0.6 of the Links web programming language includes an extension based on FST. It is available online from the Links website:

`http://links-lang.org/`

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 similar to that of provided by the impredicative polymorphism extension of GHC [22]). It incorporates a row-type system that is used for records, variants, and effects, and provides equi-recursive 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 [11].

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

12.5.1 Design Choices

Before implementing session types for Links we considered a number of design choices. Linearity is central to our description of 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 [19] or our Haskell encoding of session types [14])
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 [20])
3. bake linearity into the type system of the whole language (as in LAST [7])

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—for example, one that is able to conveniently encode parameterized monads [1], or parameterized higher-order abstract syntax [5]. 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 type system, but opens up interesting possibilities for code reuse, for instance through polymorphism over linearity [24] or through subkinding [17].

Given that we are in the business of developing our own programming language, we decided to pursue the third option. We wanted to include the full expressivity of our language in the linear fragment, so we did not see a significant benefit in stratification, and we wanted to explore possibilities for code-reuse offered by baking linearity into the type system. We were also presented with another choice regarding how to accommodate code reuse. Given that Links already supported subkinding [11] we elected to adopt the linear subkinding approach of Mazurak et al. [17].

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. For example, in FST with recursive types we can define broadcasting a value to a whole list of channels:

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

where $\mathsf{LinList}\,A$ is a linear list data type and linMap is the map operation over linear lists:

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

An attendant drawback to having first-class channels is that one must explicitly rebind channels after each operation. 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 [12, 13, 23]. To ease the job of writing a parser, we explicitly delimit process calculus style syntactic sugar with special brackets $\triangleleft - \triangleright$.

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

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 one-shot calculator server of Section 12.2 as follows:

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

In general, the syntactic sugar allows us to take advantage of a process-calculus style for communication-heavy sequences of code, but switch back to a functional style for compositional programming.

12.5.2 Type Reconstruction

Vanilla Links provides type inference, as in many other typed functional languages. However, as a consequence of the typing of application, the types of higher-order functions in FST are not uniquely determined by their uses.

As an example, consider the application operator in FST, implemented by the following term:

$$\Lambda\alpha_1^{\bullet,\star},\alpha_2^{\bullet,\star}.\lambda^{Y_1} f^{\alpha_1 \to^{Y_2} \alpha_2}.\lambda^{Y_3} x^{\alpha_1}.f\,x$$

This term is well-typed for arbitrary choices of Y_1 and Y_2, and any choice of Y_3 more constraining than Y_2, giving six distinct well-typed instantiations in all.

There are several ways we might hope to restore complete type inference, but they each come with significant additional complexity. We could introduce bounded quantification over linearities, combining the approaches of Tov and Pucella [21] and Walker [24]; in addition to introducing new forms of quantification, the implications of the resulting system for type inference have not been studied. Another approach was recently proposed by Morris [18]. His approach captures all the variations of the term above in a single term, and provides complete type inference. However, it relies on qualified types, an alternative source of complexity. In Links, we prefer unlimited function types $\tau \to^{\bullet} \tau'$ to linear function types $\tau \to^{\circ} \tau'$ when inferring the types of functions. The programmer is always free to override this choice by explicitly providing types. This approach preserves the simplicity of the language and of type reconstruction, but at the cost of some completeness.

12.6 Conclusion and Future Work

We have presented an account of lightweight functional session types, extending a core session-typed linear λ-calculus [13] with: the row typing of the core language for Links [11], the subkinding for linearity of Mazurak et al.'s lightweight linear types [17], and the asynchrony and access points of Gay and Vasconcelos's linear type theory for asynchronous session types [7].

There is a significant 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 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.

References

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

[2] G. Bernardi and M. Hennessy. Using higher-order contracts to model session types. *CoRR*, abs/1310.6176v4, 2015.

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

[4] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*. Springer, 2010.

[5] J. Carette, O. Kiselyov, and C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.

[6] J. Garrigue, G. Keller, and E. Sumii, editors. *ICFP*. ACM, 2016.

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

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

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

[10] J. Lévy and L. Maranget. Explicit substitutions and programming languages. In *FSTTCS*, volume 1738 of *LNCS*, pages 181–200. Springer, 1999.

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

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

[13] S. Lindley and J. G. Morris. A semantics for propositions as sessions. In J. Vitek, editor, *ESOP*, volume 9032 of *Lecture Notes in Computer Science*, pages 560–584. Springer, 2015.

[14] S. Lindley and J. G. Morris. Embedding session types in haskell. In G. Mainland, editor, *Haskell*, pages 133–145. ACM, 2016.

[15] S. Lindley and J. G. Morris. Lightweight functional session types (extended version). `http://homepages.inf.ed.ac.uk/slindley/papers/fst-extended.pdf`, 2016.

[16] S. Lindley and J. G. Morris. Talking bananas: structural recursion for session types. In Garrigue et al. [6], pages 434–447.

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

[18] J. G. Morris. The best of both worlds: linear functional programming without compromise. In Garrigue et al. [6], pages 448–461.

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

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

[21] J. A. Tov and R. Pucella. Practical affine types. In T. Ball and M. Sagiv, editors, *POPL*, pages 447–458. ACM, 2011.

[22] 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.

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

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