LiquidPi: Inferrable Dependent Session Types

Dennis Griffith and Elsa L. Gunter

University of Illinois at Urbana-Champaign

Abstract. The Pi Calculus is a popular formalism for modeling distributed computation. Session Types extend the Pi Calculus with a static, inferable type system. Dependent Types allow for a more precise characterization of the behavior of programs, but in their full generality are not inferable. In this paper, we present LiquidPi an approach that combines the dependent type inferencing of Liquid Types with Honda's Session Types to give a more precise automatically derived description of the behavior of distributed programs. These types can be used to describe/enforce safety properties of distributed systems. We present a type system parametric over an underlying functional language with Pi Calculus connectives and give an inference algorithm for it by means of efficient external solvers and a set of dependent qualifier templates.

1 Introduction

In a world of multiproccessors, embedded systems, and cloud computing, parallel, concurrent, and distributed programs have become ubiquitous. With their growth comes an increased need for tools and theory to design, implement and verify these programs. One of the most successful verification efforts have been type systems [12]. By providing a static characterization of program behavior type systems allow for programmers to prove that certain dangerous behaviors are impossible. Particularly useful have been automatically inferable types since they can allow access to the guarantees of type systems at a low overhead for users. Dependent types [9] focus on increasing the expressivity of type systems by allowing for types to depend on, i.e., be constructed from, the value of terms instead of only on other types. In general, this gives up on inferability, but Rondon *et al.* [13] describe an approach, liquid typing, that can allow for inferencing of certain dependent type systems.

One standard tool for the design of parallel systems is the Pi Calculus [10]. When discussing types for the Pi Calculus the notion of input and output of a process is mostly closely associated with the input and output of its channels. Thus instead of finding the type of variables and expressions like we might in a functional language, we instead look at providing types for channels. As a first pass we might say that each channel has a type like `int` to denote that it can only transmit integers. This notion of channel typing (and similar homogeneous typings [6]) gives almost no ability to characterize the temporal behavior of channels. An improvement on this approach are session types [7] which allow for a rich characterization of the temporal behavior of the channels involved in a

G. Brat, N. Rungta, and A. Venet (Eds.): NFM 2013, LNCS 7871, pp. 185–197, 2013.

system. LiquidPi is an application of the liquid typing approach to session types. The contributions of this paper are the following:

- A dependent session type system for LiquidPi (Section 3)
- An inference algorithm for the LiquidPi type system (Section 4)

2 Basic Syntax and Session Types

The Pi Calculus [10] is a process algebra for modeling distributed computation. It uses synchronous channels to pass data (including channel names) between processes that execute in parallel. The Pi Calculus can be viewed as a wrapper providing these distributed communication constructs around some underlying language of data and computation. For the purposes of this paper we will assume that the underlying language is a simple functional language. We will impose a few other requirements on this underlying language in later sections. The syntax of the Pi Calculus, along with some informal meaning, is presented in the following grammar, where x ranges over a set of data variables, e ranges over expressions in the underlying functional language, k ranges over a distinct set of channel names, τ is a type from the underlying functional language, P_i is a τ-indexed family of processes, and X ranges over a distinct set of definition variables.

$$\begin{aligned} P ::= {} & 0 \mid P \| P \mid \mathsf{accept}\ X(k).P \mid \mathsf{request}\ X(k).P \mid k!(e).P \mid k!(k).P \mid k?(x).P \\ & \mid k?(k).P \mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ P \mid (\nu k)P \mid k \triangleleft e.P_i \mid \mathsf{case}_\tau\ k \Rightarrow P_i \\ & \mid \mathsf{def}\ X(\boldsymbol{x}; \boldsymbol{k}) = P\ \mathsf{in}\ P \mid X(\boldsymbol{e}, \boldsymbol{k}) \end{aligned}$$

Informally, 0 is the terminated process. The process $P_1 \| P_2$ is the processes P_1 and P_2 executing in parallel. The process $\mathsf{accept}\ X(k).P$ initiates the session X along k and proceeds as P. The process $\mathsf{request}\ X(k).P$ is the counterpart to accept that requests the initiation of session X along k and proceeds as P. The process $k!(e).P$ sends the result of e along k and then continues as P. The process $k_1!(k_2).P$ sends the channel k_2 over k_1 and then continues as P. The process $k?(x).P$ binds the next data value sent on k to x and then continues as P. The process $k_1?(k_2).P$ receives the next channel sent on k_1 and then continues as P. The process $\mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$ evaluates e and proceeds as P_1 or P_2 as appropriate. The process $(\nu k)P$ generates a fresh channel and binds it to k. The process $k \triangleleft e.P$ evaluates e and sends it along k then proceeds as P. This will be distinguished from $k!(e).P$ in the type system by allowing the receiving process to offer differently typed behaviors based on the value of e. The process $\mathsf{case}_\tau\ k \Rightarrow P_i$ receives a value of type τ along k then proceeds as the corresponding P_i. The declaration $\mathsf{def}\ X(\boldsymbol{x}; \boldsymbol{k}) = P_1\ \mathsf{in}\ P_2$ defines X as process P_1 that can use the variables in scope along with those supplied by $\boldsymbol{x}$ and $\boldsymbol{k}$, binds the definition to X, and proceeds as P_2. The process $X(\boldsymbol{e}, \boldsymbol{k})$ calls the process defined by X and supplies it as arguments the evaluated $\boldsymbol{e}$ and $\boldsymbol{k}$.

Pi Calculus semantics are traditionally given in terms of a transition semantics that assumes a structural congruence that brings together compatible send/receive instructions so that communication can occur. For more details on semantics see Yoshida's survey [16].

Session Types [7,16] were introduced to provide a static characterization of the temporal behavior of the Pi Calculus. They rule out some dangers present in the Pi Calculus like the nondeterminism possible with channels held by more than two processes and sending and receiving processes disagreeing over the type of data being communicated. The type system disallows these while still allowing for a high degree of expressiveness such as communicating channel names and heterogeneous channel usage. The syntax of session types, S, is given by the following grammar where t ranges over a set of type variable names, τ is a type from the underlying functional language, and S_i is a τ-indexed family of session types.

$$S ::= 0 \mid t \mid \mu t.S \mid !\tau.S \mid ?\tau.S \mid ![S].S \mid ?[S].S \mid \&_\tau S_i \mid \oplus_\tau S_i$$

The informal meaning of these are as follows. 0 is the type of channels that will have no further communication. The types $\mu t.S$ and t allow us to construct (possibly infinite) recursive types. We treat types equirecursively (i.e., we identify a recursive type with its unfolding $\mu t.S = S\{\mu t.S/t\}$). The type $!\tau.S$ is that a channel that sends a data value of type τ and then proceeds as S. The type $?\tau.S$ is that of a channel that receives a data value of type τ and then proceeds as S. The type $![S_1].S_2$ is that of a channel that sends a channel with type S_1 and then proceeds as S_2. The type $?[S_1].S_2$ is that of a channel that receives a channel with type S_1 and then proceeds as S_2. The type $\&_\tau S_i$ is that of a channel that sends a piece of data of type τ and then proceeds as the appropriate τ-index S_i. The type $\oplus_\tau S_i$ is that of a channel that receives a piece of data of type τ and then proceeds as the appropriate τ-index S_i. As with processes, our types have a notion of send/receive pairs. We define the notion of a dual type to encode this correspondence. The dual of a session type S is denoted $\overline{S}$ and defined below.

$$\overline{0} = 0 \qquad \overline{!\tau.S} = ?\tau.\overline{S} \qquad \overline{?\tau.S} = !\tau.\overline{S}$$

$$\overline{![S_1].S_2} = ?[S_1].\overline{S_2} \quad \overline{?[S_1].S_2} = ![S_1].\overline{S_2} \quad \overline{\&_\tau S_i} = \oplus_\tau \overline{S_i} \quad \overline{\oplus_\tau S_i} = \&_\tau \overline{S_i}$$

The session type system will use duality to match up compatible channel users. A channel typing is a mapping from channel variables to session types. The last notation needed is for marking polarity. Polarity markings are superscripts on channel names that will allow us to distinguish the two conceptual "ends" of a channel so that we can rule out send/receive confusion and more than two processes using a channel at once. We use k^+ to denote the positive end of channel k, k^- to denote the negative "end", and $k^{\overline{p}}$ to denote swapping the polarity of k^p.

Using the notions above we can give the rules for session types. We use Θ to denote a mapping from process variables to tuples of their argument types, Γ to denote typings for our functional variables, and Δ to denote channel typings. We use $\Delta_1 \cdot \Delta_2$ to denote the merger of two channel typings that share no

$$\frac{\Theta; \Gamma \vdash_S P : \Delta \cdot (k : S) \qquad \Gamma \vdash e : \tau}{\Theta; \Gamma \vdash_S k!(e).P : \Delta \cdot (k : !\tau.S)}\ \text{T.Send} \qquad \frac{\Theta; \Gamma \cdot x : \tau \vdash_S P : \Delta \cdot (k : S)}{\Theta; \Gamma \vdash_S k?(x).P : \Delta \cdot (k : ?\tau.S)}\ \text{T.Rec}$$

$$\frac{\text{for } (k, S) \in \Delta\text{: } S{=}0}{\Theta; \Gamma \vdash_S 0 : \Delta}\ \text{T.End} \qquad \frac{\Theta; \Gamma \vdash_S P : \Delta \cdot k_1 : S_1}{\Theta; \Gamma \vdash_S k_1!(k_2).P : \Delta \cdot (k_1 : ![S_2].S_1) \cdot (k_2 : S_2)}\ \text{T.Thr}$$

$$\frac{\Theta; \Gamma \vdash_S P : \Delta \cdot (k^p : S) \cdot (k^{\overline{p}} : \overline{S})}{\Theta; \Gamma \vdash_S (\nu\ k)P : \Delta}\ \text{T.Nu} \qquad \frac{\Theta; \Gamma \vdash_S P : \Delta \cdot (k_1 : S_1) \cdot (k_2 : S_2)}{\Theta; \Gamma \vdash_S k_1?(k_2).P : \Delta \cdot (k_1 : ?[S_2].S_1)}\ \text{T.Cat}$$

$$\frac{\Theta; \Gamma \vdash_S P : \Delta \cdot (k^+ : G(X))}{\Theta; \Gamma \vdash_S \mathsf{accept}\ X(k).P : \Delta}\ \text{T.Acc} \qquad \frac{\Theta; \Gamma \vdash_S P : \Delta \cdot (k^- : \overline{G(X)})}{\Theta; \Gamma \vdash_S \mathsf{request}\ X(k).P : \Delta}\ \text{T.Req}$$

$$\frac{\Theta; \Gamma \vdash_S P : \Delta_1 \qquad \Theta; \Gamma \vdash_S P : \Delta_2}{\Theta; \Gamma \vdash_S P \| Q : \Delta_1 \cdot \Delta_2}\ \text{T.Par}$$

$$\frac{\Gamma \vdash e : \mathsf{Bool} \qquad \Theta; \Gamma \vdash_S P : \Delta \qquad \Theta; \Gamma \vdash_S Q : \Delta}{\Theta; \Gamma \vdash_S \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q : \Delta}\ \text{T.If}$$

$$\frac{\Gamma \vdash e : \tau : \textsc{Enum} \qquad \text{for } i \in \tau\text{: } \Theta; \Gamma \vdash_S P_i : \Delta \cdot (k : S_i)}{\Theta; \Gamma \vdash_S k \triangleleft e.P_i : \Delta \cdot k : \&_\tau S_i}\ \text{T.Int}$$

$$\frac{\tau : \textsc{Enum} \qquad \text{for } i \in \tau\text{: } \Theta; \Gamma \vdash_S P_i : \Delta \cdot (k : S_i)}{\Theta; \Gamma \vdash_S \mathsf{case}_\tau\ k \Rightarrow P_i : \Delta \cdot k : \oplus_\tau S_i}\ \text{T.Ext}$$

$$\frac{\text{for } i \in \mathrm{dom}(\Delta)\text{: } \Delta(i) = 0 \qquad \text{for } i\text{: } \Gamma \vdash e_i : \tau_i \qquad \text{for } (k, S) \in \Delta\text{: } S{=}0}{\Theta \cdot X : (\boldsymbol{\tau}, \boldsymbol{S}); \Gamma \vdash_S X(\boldsymbol{e}, \boldsymbol{k}) : \Delta \cdot \boldsymbol{k} : \boldsymbol{S}}\ \text{T.Call}$$

$$\frac{\Theta \cdot X : (\boldsymbol{\tau}, \boldsymbol{S}); \Gamma \cdot \boldsymbol{x} : \boldsymbol{\tau} \vdash_S P : (\boldsymbol{k} : \boldsymbol{S}) \qquad \Theta \cdot X : (\boldsymbol{\tau}, \boldsymbol{S}); \Gamma \vdash_S Q : \Delta}{\Theta; \Gamma \vdash_S \mathsf{def}\ X(\boldsymbol{x}; \boldsymbol{k}) = P\ \mathsf{in}\ Q : \Delta}\ \text{T.Def}$$

Fig. 1. Typing Rules for Simple Session Types

common bindings. We use the hypothesis $\tau : \textsc{Enum}$ to denote that the type τ is a finite enumeration. Depending on the details underlying functional language this may have different interpretations. These enumerations could be smoothly generalized to algebraic datatypes, but we present only the simplified view to avoid unneeded clutter. The judgment $\Theta; \Gamma \vdash_S P : \Delta$ denotes that, assuming the definitions of Θ and the functional types in Γ, the free channel variables of process P have the session types in Δ. We use $\Gamma \vdash e : \tau$ to denote that the type system for the underlying functional language proves that e has type τ from the assumptions in Γ. We assume that sessions have some globally visible type and so assume a mapping, G, from session names to session types. Figure 1 contains a listing of the typing rules for simple session types. To see how the rules eliminate dangerous behavior consider the rule T.Nu. This rule ensures two things:

the fresh channel has two and only two "ends"; the users of each end agree both in the direction of communication at every step and the type of value or channel being communicated.

3 Refinement Type System

Dependent Types [9] are types that allow the meaning of types to depend on data values. As an example, when trying to describe the type of division we might be interested in allowing only non-zero `int`s instead of all `int`s as divisors. What we mostly will be interested in are a restricted class of dependent types called refinement types. A refinement type is a basic type (i.e., a non-compound type–`int` but not `int->float`) with a predicate attached to it; e.g., the positive integers are given by $\{v : \texttt{int} | 0 \leq v\}$. Simple types can naturally be viewed as refinement types by using the trivial always-true predicate. From this follows a natural notion of subtyping (with the normal contravariance for functions). In addition to allowing predicates to incorporate constants, we will want them to allow for dependency on previously bound terms, e.g., $\{v : \texttt{int} | v \leq x\}$ for some previously bound x. For compound functional types we assume that refinements are available on the "leaf" types [13]. Refined session types, Υ, are generated by the following grammar, where ρ denotes a refined simple type and Υ_i denotes a τ-indexed family of refined session types.

$$\Upsilon ::= 0 \mid t \mid \mu t.\Upsilon \mid !\rho.\Upsilon \mid ?x \in \rho.\Upsilon \mid ![\Upsilon].\Upsilon \mid ?[\Upsilon].\Upsilon \mid \&_\tau \Upsilon_i \mid \oplus_\tau \Upsilon_i$$

These types are nearly the same as their simple counterparts but utilizing refined functional types instead of simple functional types. A construct that does change is $?x \in \rho.\Upsilon$. This construct allows refined session types to bind the data value that was sent across the channel and refer to this in later refined types. In particular, this allows for session types like $?\texttt{x} \in \{v : \texttt{int} | \textsc{True}\}.!\{v : \texttt{int} | v \geq \texttt{x}\}.0$, which would be a refined session type for describing a process that receives an integer and then returns the absolute value of that integer. Why not provide more binders? For the sending of data there is no new value introduced, e could always be reconstructed in our refinement as needed, so there is nothing to bind. An additional practical consideration is that it is not obvious what variable to use to bind the result of e. For sending and receiving channels, we assume that the logic that Section 4 uses cannot analyze channels and so have no need to refer to a received channel in our predicates. For the two choice constructs, there is no need to provide an explicit binding for the enumeration value chosen, the τ-indexed family of types can already implicitly use this knowledge.

We will need a few more definitions before introducing the typing rules for refined session types. First, $\rho\lfloor$ is a refined type with all the refinement information striped out (e.g., $\{v : \texttt{int} | 0 \leq v\}\lfloor = \texttt{int}$). This has a natural generalization to environments and typings. The notion of the dual of a session type is essentially unchanged except for the need to handle bindings during the reception of data, so we say that for any x, $\overline{!\rho.\Upsilon} = ?x \in \rho.\overline{\Upsilon}$ and $\overline{?x \in \rho.\Upsilon} = !\rho.\overline{\Upsilon}$. Additionally, refinements introduce a notion of subtyping. We use $\Gamma \vdash \rho_1 \sqsubseteq \rho_2$ to denote that

$$\frac{\Theta; \Gamma \vdash_{SL} P : \Delta \cdot k : \Upsilon \qquad \Gamma \vdash_L e : \rho \qquad \Gamma \vdash \rho \sqsubseteq \rho'}{\Theta; \Gamma \vdash_{SL} k!(e).P : \Delta \cdot (k : !\rho'.\Upsilon)} \text{ R.Send}$$

$$\frac{\Theta; \Gamma \cdot x : \rho \vdash_{SL} P : \Delta \cdot (k : \Upsilon) \qquad \Gamma \vdash \rho' \sqsubseteq \rho}{\Theta; \Gamma \vdash_{SL} k?(x).P : \Delta \cdot (k : ?x \in \rho'.\Upsilon)} \text{ R.Rec}$$

$$\frac{\Theta; \Gamma \vdash_{SL} P : \Delta \cdot k_1 : \Upsilon_1}{\Theta; \Gamma \vdash_{SL} k_1!(k_2).P : \Delta \cdot (k_1 : ![\Upsilon_2].\Upsilon_1) \cdot (k_2 : \Upsilon_2)} \text{ R.Thr}$$

$$\frac{\Theta; \Gamma \vdash_{SL} P : \Delta \cdot (k_1 : \Upsilon_1) \cdot (k_2 : \Upsilon_2)}{\Theta; \Gamma \vdash_{SL} k_1?(k_2).P : \Delta \cdot (k_1 : ?[\Upsilon_2].\Upsilon_1)} \text{ R.Cat} \qquad \frac{\Theta; \Gamma \vdash_{SL} P : \Delta \cdot (k^p : \Upsilon) \cdot (k^{\overline{p}} : \overline{\Upsilon})}{\Theta; \Gamma \vdash_{SL} (\nu\ k)P : \Delta} \text{ R.Nu}$$

$$\frac{\Theta; \Gamma \vdash_{SL} P : \Delta_1 \qquad \Theta; \Gamma \vdash_{SL} P : \Delta_2}{\Theta; \Gamma \vdash_{SL} P \| Q : \Delta_1 \cdot \Delta_2} \text{ R.Par} \qquad \frac{\text{for } (k, \Upsilon) \in \Delta\text{: } \Upsilon{=}0}{\Theta; \Gamma \vdash_{SL} 0 : \Delta} \text{ R.End}$$

$$\frac{\Theta; \Gamma \vdash_{SL} P : \Delta \cdot (k^+ : G_L(X))}{\Theta; \Gamma \vdash_{SL} \mathsf{accept}\ X(k).P :} \text{ R.Acc} \qquad \frac{\Theta; \Gamma \vdash_{SL} P : \Delta \cdot (k^- : \overline{G_L(X)})}{\Theta; \Gamma \vdash_{SL} \mathsf{request}\ X(k).P :} \text{ R.Req}$$

$$\frac{\Theta; \Gamma \cdot e \vdash_{SL} P : \Delta_1 \qquad \Theta; \Gamma \cdot \neg e \vdash_{SL} Q : \Delta_2 \qquad \begin{array}{c}\Gamma \vdash_L e : \rho \qquad \rho\downarrow = \mathsf{Bool} \\ \Gamma \cdot e \vdash \Delta_1 \sqsubseteq \Delta \qquad \Gamma \cdot \neg e \vdash \Delta_2 \sqsubseteq \Delta\end{array}}{\Theta; \Gamma \vdash_{SL} \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q : \Delta} \text{ R.If}$$

$$\frac{\Gamma \vdash_L e : \rho \qquad \rho\downarrow\text{: Enum} \qquad \text{for } i \in \rho\downarrow\text{: } \Theta; \Gamma \vdash_{SL} P_i : \Delta \cdot (k : S_i)}{\Theta; \Gamma \vdash_{SL} k \triangleleft e.P_i : \Delta \cdot k : \&_\tau S_i} \text{ R.Int}$$

$$\frac{\tau : \text{Enum} \qquad \text{for } i \in \tau\text{: } \Theta; \Gamma \vdash_{SL} P_i : \Delta \cdot (k : S_i)}{\Theta; \Gamma \vdash_{SL} \mathsf{case}_\tau\ k \Rightarrow P_i : \Delta \cdot k : \oplus_\tau S_i} \text{ R.Ext}$$

$$\frac{\text{for i: } \Gamma \vdash_L e_i : \rho_i' \qquad \text{for i: } \Gamma \vdash \rho_i' \sqsubseteq \rho_i \qquad \text{for } (k, \Upsilon) \in \Delta\text{: } \Upsilon{=}0}{\Theta \cdot X : (\boldsymbol{\rho}, \boldsymbol{\Upsilon}); \Gamma \vdash_{SL} X(\boldsymbol{e}, \boldsymbol{k}) : \Delta \cdot \boldsymbol{k} : \boldsymbol{\Upsilon}} \text{ R.Call}$$

$$\frac{\Theta \cdot X : (\boldsymbol{\rho}, \boldsymbol{\Upsilon}); \Gamma \cdot \boldsymbol{x} : \boldsymbol{\rho} \vdash_{SL} P : (\boldsymbol{k} : \boldsymbol{\Upsilon}) \qquad \Theta \cdot X : (\boldsymbol{\rho}, \boldsymbol{\Upsilon}); \Gamma \vdash_{SL} Q : \Delta}{\Theta; \Gamma \vdash_{SL} \mathsf{def}\ X(\boldsymbol{x}; \boldsymbol{k}) = P\ \mathsf{in}\ Q : \Delta} \text{ R.Def}$$

Fig. 2. Type Rules for Refined Session Types

ρ_1 is a subtype of ρ_2 under the assumptions in Γ (defined by Rondon [13]) and $\Gamma \vdash \Upsilon_1 \sqsubseteq \Upsilon_2$ for subtyping of refined session types, defined below.

$$\frac{}{\Gamma \vdash 0 \sqsubseteq 0} \qquad \frac{\Gamma \vdash \rho_1 \sqsubseteq \rho_2 \qquad \Gamma \vdash \Upsilon_1 \sqsubseteq \Upsilon_2}{\Gamma \vdash !\rho_1.\Upsilon_1 \sqsubseteq !\rho_2.\Upsilon_2}$$

$$\frac{\Gamma \vdash \rho_1 \sqsubseteq \rho_2 \qquad \Gamma \vdash \Upsilon_1 \sqsubseteq \Upsilon_2}{\Gamma \vdash ?x \in \rho_1.\Upsilon_1 \sqsubseteq ?x \in \rho_2.\Upsilon_2} \qquad \frac{\Gamma \vdash \Upsilon_1 \sqsubseteq \Upsilon_2}{\Gamma \vdash ![\Upsilon].\Upsilon_1 \sqsubseteq ![\Upsilon].\Upsilon_2}$$

$$\frac{\Gamma \vdash \Upsilon_1 \sqsubseteq \Upsilon_2}{\Gamma \vdash ?[\Upsilon].\Upsilon_1 \sqsubseteq ?[\Upsilon].\Upsilon_2} \qquad \frac{\Gamma \vdash \Upsilon_1 \sqsubseteq \Upsilon_2}{\Gamma \vdash ?[\Upsilon].\Upsilon_1 \sqsubseteq ?[\Upsilon].\Upsilon_2}$$

$$\frac{\Gamma \vdash \text{for } i\text{: } \Upsilon_i \sqsubseteq \Upsilon_i'}{\Gamma \vdash \&_\tau \Upsilon_i \sqsubseteq \&_\tau \Upsilon_i'} \qquad \frac{\text{for all } i\text{: } \Gamma \vdash \Upsilon_i \sqsubseteq \Upsilon_i'}{\Gamma \vdash \oplus_\tau \Upsilon_i \sqsubseteq \oplus_\tau \Upsilon_i'}$$

Figure 2 introduces the typing rules for Refined Session Types. $\Theta; \Gamma \vdash_{SL} P : \Delta$ denotes that, using the definitions of Θ and assumptions of Γ (many of which are just functional typing assignments), the free process channels of process P have the refined session types in Δ. $\Gamma \vdash_L e : \rho$ denotes that, under the assumptions of Γ, e has refined type ρ, the details of which depend on the underlying functional language. The rules are similar to the rules presented for unrefined session types

but with the addition of subtyping information where appropriate. R.Send uses the idea that a process may transmit a subtype of its declared type and still maintain correct behavior. Conversely, R.Rec encodes that a process may use a looser approximation of its received data than required while still maintaining correctness. R.Nu remains "unchanged" for two reasons. First, the notion of duality has changed a bit, so an implicit change to handle refinements occurs. Second, while this would be a reasonable place to include subtyping information but the rules R.Send and R.Rec already account for this. Similarly R.Call and not R.Def encapsulates the idea that definitions usage can accepted more tightly constrained types for a particular instance than they accept in general. Perhaps the most interesting rule is R.If. This rule makes refined session types path sensitive [1] by allowing for both branches to have different types and slightly different assumptions (e vs. $\neg e$) and then combining to have one unified typing for the whole process.

The type system for refined session types has a close connection with the simple session types as exhibited by the following lemma.

Lemma 1 (Judgement Correspondence). *For refined definition environment Θ, refined functional assumptions Γ, process P and refined channel environment Δ, $\Theta; \Gamma \vdash_{SL} P : \Delta$ implies $\Theta\lfloor; \Gamma\lfloor \vdash_S P : \Delta\lfloor$. For simple definition environment Θ_1, simple functional environment Γ_1, and simple channel typing Δ_1, $\Theta_1; \Gamma_1 \vdash_S P : \Delta_1$ implies there exists Θ_2, Γ_2, and Δ_2 s.t. $\Theta_2; \Gamma_2 \vdash_{SL} P : \Delta_2$ and $\Theta_2\lfloor = \Theta_1$, $\Gamma_2\lfloor = \Gamma$, and $\Delta_2\lfloor = \Delta_1$.*

Proof (Sketch). Both proofs proceed by induction on the size of proof trees. For the first result, notice that by dropping all the refinement information (and subtyping) each of the refined session type rules becomes a simple session typing rule. For the second result, use the trivial always-true predicate to (not) constrain the types.

4 Inferencing

Inferring arbitrary refinement predicates is undecidable in general (consider trying to infer the type of a function that generates random primes) so we will restrict our attention significantly. In particular, we will fix some set of basic predicates and then infer predicates that are finite conjunctions drawn from this set. For example, if wishing to infer simple interval properties we might have a set of predicates like $\{v \leq 5, v \leq \mathsf{x}, \mathsf{y} \leq v, \dots\}$. Following [13], we will assume that this set is generated by a finite set of templates instantiated by program variables. We then look for conjunctions of ground substitutions for these templates that are suitable solutions to our constraints.

Inferring refined session types proceeds in three major steps:

1. Infer simple types and record some information from doing so
2. Add predicate variables to types and gather constraints on them
3. Solve these constraints

4.1 Simple Types

Inferring simple types is done by utilizing prior work [7,1,3]. In particular, we assume that for our functional language we can infer simple types. During this inferencing we will need to record a bit of extra information. Specifically, we will assume that the simple session type inferencing algorithm annotates channel generation with the channel's session type. Because of polarity considerations there is not a single type for a channel but two dual types, one for each end. For presentational compactness, we will assume that $(\nu k)P$ is annotated to become $(\nu\ k : S)P$ were S was the type of k^+ found during inferencing. Additionally, we will assume that parallel compositions are annotated with how to split the combined channel typing environment for the process into one typing for each of the two subprocesses. We will denote this split by converting $P_1 \| P_2$ into $P_{1\,K_1} \|_{K_2} P_2$ with the names of K_i being those for P_i. Last, we assume that definitions are annotated with their argument types. That is $\mathsf{def}\ X(\boldsymbol{x}; \boldsymbol{k}) = P_1\ \mathsf{in}\ P_2$ becomes $\mathsf{def}\ X(\boldsymbol{x}; \boldsymbol{k}) : (\boldsymbol{\tau}; \boldsymbol{S}) = P_1\ \mathsf{in}\ P_2$. With these annotations we will be able to calculate at any point the simple channel typing of a subprocess of the process that we are trying to infer types. A more complicated implementation might be able to cache information closer to its use location, but we think these annotations provide a good trade-off between clarity and completeness.

4.2 Constraints

We utilize constraints of the following forms during constraint generation. $\Gamma \vdash_{\text{wf}} \Upsilon$ indicates that Υ is well-formed w.r.t. Γ, i.e., that the free variables in Υ are bound in Γ, ($\text{FN}(\Upsilon) \subseteq \text{dom}(\Gamma)$). Additionally, we use subtyping requirements of the form $\Gamma \vdash \Upsilon_1 \sqsubseteq \Upsilon_2$ and $\Gamma \vdash \rho_1 \sqsubseteq \rho_2$. The constraint $\Upsilon_1 = \overline{\Upsilon_2}$ is used to enforce duality. We also lift our constraints to work on (equal length) vectors of types pointwise (e.g., $\Gamma \vdash_{\text{wf}} \boldsymbol{\rho}$ is equivalent to $\bigcup\{\Gamma \vdash_{\text{wf}} \rho_i\}$).

We assume that we have some constraint generation algorithm that will produce correct constraints for our underlying functional language [13]. Armed with this we can read our typing rules as generating constraints by inserting subtyping constraints as appropriate (and in the case of T.Nu a duality constraint). Throughout the process of constraint gathering we will occasionally need to generate new refined session types with predicate variables, we denote this by $\tau\uparrow$ for basic types and $S\uparrow$ for session types. Whenever we perform this generation we will provide some well-formedness constraint in addition to any subtyping constraints generated by the typing rules.

As an example consider the rule R.Send. Suppose that we know $\Theta\downarrow; \Gamma\downarrow \vdash_{SL} k!(e).P : \Delta \cdot (k : !\tau.S)$ from our simple inference step. When we generate constraints for this we will make one call to our functional constraint generation algorithm ($\Gamma \vdash_L e : \rho$), one recursive call to our session type constraint generation algorithm ($\Theta; \Gamma \vdash_{SL} P : \Delta \cdot k : \Upsilon$), generate one refined type $\tau\uparrow$, and add the constraints $\Gamma \vdash \rho \sqsubseteq \tau\uparrow$ and $\Gamma \vdash_{\text{wf}} \tau\uparrow$, which corresponds to the constraints imposed by the typing rule.

Figure 3 provides a listing of the constraint generation algorithm for refined session types. $\text{Constr}_{\text{SL}}(\Theta, \Gamma, P, \Delta_S)$ returns (Δ, C) a pair of refined channel

$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma, 0, \Delta_S) = (\Delta_S, \emptyset)$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma, \mathsf{accept}\ X(k).P, \Delta_S) =$
  $(\Delta \cdot (k^+ : \Upsilon), C) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma, P, \Delta \cdot (k^+ : G(X)))$
  $\mathsf{Return}\ (\Delta, C \cup \{\Gamma \vdash \Upsilon \sqsubseteq G_L(X)\})$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma, \mathsf{request}\ X(k).P, \Delta_S) =$
  $(\Delta \cdot (k^- : \Upsilon), C) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma, P, \Delta \cdot (k^- : \overline{G(X)}))$
  $\mathsf{Return}\ (\Delta, C \cup \{\Gamma \vdash \Upsilon \sqsubseteq \overline{G_L(X)}\})$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma, k!(e).P, \Delta_S \cdot (k :!\tau.S)) =$
  $(\Delta \cdot k : \Upsilon, C) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma, P, \Delta_S \cdot (k : S))$
  $(\rho, C') \leftarrow \textsc{Constr}_{\mathrm{L}}(\Gamma, e)$
  $\rho' \leftarrow \tau\!\uparrow$
  $\mathsf{Return}\ (\Delta \cdot k :!\rho'.\Upsilon, C \cup C' \cup \{\Gamma \vdash_{\mathrm{wf}} \rho'; \Gamma \vdash \rho \sqsubseteq \rho'\})$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma, k?(x).P, \Delta_S \cdot (k :?\tau.S)) =$
  $\rho \leftarrow \tau\!\uparrow$
  $\rho' \leftarrow \tau\!\uparrow$
  $(\Delta \cdot k : \Upsilon, C) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma \cdot x : \rho, P, \Delta_S \cdot (k : S))$
  $\mathsf{Return}\ (\Delta \cdot k :?x \in \rho'.\Upsilon, C \cup \{\Gamma \vdash_{\mathrm{wf}} \rho; \Gamma \vdash_{\mathrm{wf}} \rho'; \Gamma \vdash \rho \sqsubseteq \rho'\})$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma, k_1!(k_2).P, \Delta_S \cdot (k_1 :![S_2].S_1) \cdot (k_2 : S_2)) =$
  $(\Delta \cdot k_1 : \Upsilon_1, C) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma, P, \Delta_S \cdot (k_1 : S_1))$
  $\Upsilon_2 \leftarrow S_2\!\uparrow$
  $\mathsf{Return}\ (\Delta \cdot k_1 :![\Upsilon_2].\Upsilon_1, C \cup \{\Gamma \vdash_{\mathrm{wf}} \Upsilon_2\})$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma, k_1?(k_2).P, \Delta_S \cdot (k_1 :?[S_2].S_1)) =$
  $(\Delta \cdot (k_1 : \Upsilon_1) \cdot (k_2 : \Upsilon_2), C) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma, P, \Delta_S \cdot (k_1 : S_1) \cdot (k_2 : S_2))$
  $\mathsf{Return}\ (\Delta \cdot (k_1 :?[\Upsilon_2].\Upsilon_1), C)$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma, P_1 {}_{K_1}\|_{K_2} P_2, \Delta_S) =$
  $(\Delta_1, C_1) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma, P_1, \Delta_S \upharpoonright_{K_1})$
  $(\Delta_2, C_2) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma, P_2, \Delta_S \upharpoonright_{K_2})$
  $\mathsf{Return}\ (\Delta_1 \cdot \Delta_2, C_1 \cup C_2)$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma, k \triangleleft e.P_i, \Delta_S \cdot (k : \&_\tau S_i)) =$
  for i: $(\Delta \cdot (k : \Upsilon_i), C_i) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma \cdot (e = i), P_i, \Delta_S \cdot (k : S_i))$
  $\mathsf{Return}\ (\Delta \cdot (k : \&_\tau \Upsilon_i), \bigcup C_i)$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma, \mathsf{case}_k\ e \Rightarrow P_i, \Delta_S \cdot (k : \oplus_\tau S_i)) =$
  for i: $(\Delta \cdot (k : \Upsilon_i), C_i) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma \cdot (e = i), P_i, \Delta_S \cdot (k : S_i))$
  $\mathsf{Return}\ (\Delta \cdot (k : \&_\tau \Upsilon_i), \bigcup C_i)$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma, \mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2, \Delta_S) =$
  $(\Delta_1, C_1) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma \cdot e, P_1, \Delta_S)$
  $(\Delta_2, C_2) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma \cdot (\neg e), P_2, \Delta_S)$
  for $k \in \mathrm{dom}(\Delta_S)$: $\Upsilon_k \leftarrow \Delta_S(k)\!\uparrow$
  $\mathsf{Return}\ \left(\boldsymbol{k} : \boldsymbol{\Upsilon}, C_1 \cup C_2 \cup \bigcup_{k \in \mathrm{dom}(\Delta)} \left\{ \begin{array}{l} \Gamma \vdash_{\mathrm{wf}} \Upsilon_k; \\ \Gamma \cdot e \vdash \Delta_1(k) \sqsubseteq \Upsilon_k; \\ \Gamma \cdot (\neg e) \vdash \Delta_2(k) \sqsubseteq \Upsilon_k \end{array} \right\}\right)$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma, (\nu\ k : S)P, \Delta_S) =$
  $(\Delta \cdot (k^+ : \Upsilon_1) \cdot (k^- : \Upsilon_2), C) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma, P, \Delta \cdot (k^+ : S) \cdot (k^- : \overline{S}))$
  $\mathsf{Return}\ (\Delta, C \cup \{\Upsilon_1 = \overline{\Upsilon_2}\})$
$\textsc{Constr}_{\mathrm{SL}}(\Theta \cdot (X : (\boldsymbol{\rho}, \boldsymbol{\Upsilon})), \Gamma, X(\boldsymbol{e}, \boldsymbol{k}), \Delta_S \cdot (\boldsymbol{k} : \boldsymbol{S})) =$
  for i: $(\rho'_i, C_i) \leftarrow \textsc{Constr}_{\mathrm{L}}(\Gamma, e_i)$
  $\mathsf{Return}\ (\Delta_S \cdot (\boldsymbol{k} : \boldsymbol{\Upsilon}), \bigcup C_i \cup \{\Gamma \vdash \boldsymbol{\rho'} \sqsubseteq \boldsymbol{\rho}\})$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Gamma, \mathsf{def}\ X(\boldsymbol{x}; \boldsymbol{k}) : (\boldsymbol{\tau}; \boldsymbol{S}) = P_1\ \mathsf{in}\ P_2, \Delta_S \cdot (\boldsymbol{k} : \boldsymbol{S})) =$
  $(\boldsymbol{\rho}; \boldsymbol{\Upsilon}) \leftarrow (\boldsymbol{\tau}\!\uparrow; \boldsymbol{S}\!\uparrow)$
  $(\Delta_1 \cdot (\boldsymbol{k} : \boldsymbol{\Upsilon'}), C_1) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta \cdot (X : (\boldsymbol{\rho}; \boldsymbol{\Upsilon})), \Gamma \cdot (\boldsymbol{x} : \boldsymbol{\rho}), P_1, (\boldsymbol{k} : \boldsymbol{S}))$
  $(\Delta_2, C_2) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta \cdot (X : (\boldsymbol{\rho}; \boldsymbol{\Upsilon'})), \Gamma, P_2, \Delta_S)$
  $\mathsf{Return}\ (\Delta_2, C_1 \cup C_2 \cup \{\Gamma \vdash_{\mathrm{wf}} \boldsymbol{\rho}; \Gamma \vdash_{\mathrm{wf}} \boldsymbol{\Upsilon}; \Gamma \vdash \boldsymbol{\Upsilon'} \sqsubseteq \boldsymbol{\Upsilon}\})$

Fig. 3. Constraint Generation Algorithm

typing (with predicate variables) and a set of constraints. We use $\textsc{Constr}_{\mathrm{L}}(\Gamma, e)$ to denote the assumed constraint gatherer of our underlying functional language. The algorithm assumes that, for functional assumptions Γ, $\textsc{Constr}_{\mathrm{L}}(\Gamma, e)$ returns (ρ, C) a pair of a refined functional type and a set of constraints (both well-formedness and subtyping). A small abuse of notation occurs in the case for terminated processes and in process variable definition. Specifically, we use Δ_S as both a simple session typing and as a refined one. From our typing rules we know in both cases it must be entirely composed of mappings of the form $(k : 0)$

and so can be reasonably used in both contexts. While most of the cases used to define $\textsc{Constr}_{\textsc{SL}}$ are relatively straightforward we highlight a few rules here.

Consider the case for conditional branching, perhaps the most complicated case. First we make recursive calls with the altered assumptions, allowing for sensitivity to the value of e. From R.If we know that both of the typings returned by these must be subtypes of our overall typing. Since we do not have a preexisting typing use for this subtyping we have to generate one $(\Delta_S(k)\restriction)$. We then return this typing along with our recursively generated constraints and three new constraints for each channel in our typing. The first new constraint ensures that our freshly generated types are well-formed. The other encodes the subtyping present in the rule. One might worry that if both k^p and $k^{\overline{p}}$ appear in our typing that this might cause them to become delinked. Since we will only use our constraint generation on closed processes after simple session type inferencing we know that these paired channel ends will eventually be generated by some $((\nu\ k)P)$ and thus duality will be ensured there.

The following lemma gives us the correctness of our constraint generation algorithm.

Lemma 2 (Constraint Correctness). *For a closed annotated process P, empty definition and functional environments and simple typing,* $\textsc{Constr}_{SL}(\emptyset, \emptyset, P, \emptyset)$ *returns (Δ, C), s.t. $\emptyset; \emptyset \vdash_{SL} P : \Delta$ if and only if C has a solution.*

Proof (Sketch). Induction on the proof trees of $\Theta; \Gamma \vdash_{SL} P : \Delta$ for a generalization of the lemma to non-empty environments.

4.3 Solving

Once all constraints have been generated, we will have many predicate variables left. A solution to a system of constraints is a ground substitution for predicate variables such that all constraints are satisfied. Assuming that our constraints allow all legal solutions (Lemma 2), we know that there is at least one possible solution, the trivial always-true solution. The important question is then that of finding a maximally specific solution. We search for a maximal solution using the normal implication ordering lifted to maps (i.e., $\sigma_1 \geq \sigma_2 \iff \forall x.\sigma_1(x) \implies \sigma_2(x)$).

A first pass removes all duality constraints by performing the substitutions implied by the equations. Since all Γ in our constraints are finite and every predicate variable has at least one well-formedness constraint, we know that for any given predicate variable, there can be at most a finite number of ground substitutions admissible by its well-formedness constraints. This together with the observation that only the predicate variables mentioned in our constraints matter for a substitution's admissibility, we have only a finite number of "interesting" substitutions that might be solutions. Assuming that we can decide admissibility and solution ordering (e.g., via an SMT solver) then we can just try all solutions and select a maximal one. This requirement for being able to decide ordering is perhaps the biggest constraint on what we can choose as our

templates, since we need to stay away from choosing those that are incompatible with our choice of SMT solver.

This proposed solution process is unsatifyingly slow, so we instead suggest using Iterative Weakening [13]. Iterative Weakening is a technique that starts from the strongest admissible ground substitution (for each predicate variable a conjunction of all predicates admissible by its well-formedness constraints) and iteratively removes an offending conjunct. Since we deal with conjunctions of instantiated templates we know that removing a conjunct can at most preserve a substitution's strength and the always-true substitution is a solution, we know that iterative weakening will find a maximally specific solution. From the above arguments we have the following lemma.

Lemma 3 (Solver Correctness). *For a given set of constraints, s.t. every predicate variable has at least one (finite) well-formedness constraint, iterative weakening produces a maximally specific solution.*

Proof (Sketch). Outlined above, this is proven by a generalization of Rondon [13].

5 Related Work

The most direct related work are the series of papers by Rondon *et al.* [13,8,14] applying their liquid typing approach to infer refinement types for various languages. Our work can be seen as a continuance of this line of work by applying it to the Pi Calculus.

Gay and Hole [5] present a Pi Calculus with session types and subtyping on the choice operators, allowing flexibility on the number of branches for the type of an internal choice and its corresponding external choice. Additionally, read-write permission using Pi Calculus type systems tend to have subtyping for their permissions [6]. Here subtyping is utilized to track permissions with having only read or write permissions being viewed as a subtype of having both. Both of these notions of subtyping are orthogonal to the subtyping used in this paper.

Perhaps closest to our dependent session types are those found in Caires *et al.* [2]. They allow for full dependent types and envision writing proof carrying code, at the functional level, by transmitting proofs across channels. They do not (as expected) address inferring types for their programs. Additionally they use linear logic as a basis for their typing system which gives a fairly different feeling, since we do not need to worry about differentiating between linear and replicateable resources.

While we use some of the simplest session types as a basis for LiquidPi a number of extensions to them have been made [15,4,11]. In particular, these recent works have studied global types, providing a holistic description of system communication, instead of per channel types using asynchronous communication. These too also involve a notion of subtyping, but it is used for dealing with asynchrony and not at the functional level. We expect that the liquid typing approach would likely apply in these cases, though with extra complexity arising from their more complicated type systems.

6 Conclusion and Future Work

We have presented LiquidPi an approach that combines the dependent type inferencing of Liquid Types with Hondas Session Types to give a more precise automatically derived description of the behavior of distributed programs. These types can be used to describe/enforce safety properties of distributed systems. We presented a type system parametric over an underlying functional language with Pi Calculus connectives and give an inference algorithm for it by means of efficient external solvers and a set of dependent qualifier templates. By doing this we demonstrate that inferring dependent types for communication is achievable, gaining a fair amount of expressivity compared to previous techniques.

As described in Section 5, there are many variations of type systems for distributed systems that have been presented, it would be interesting to integrate inferable dependent types into them as well to yield greater expressivity. Another natural thing to do with this work is to create an efficient implementation. With the ease of use of modern SMT solvers a simple prototype shouldn't be infeasible, but heuristics for the weakening step of iterative weakening might need more investigation.

Acknowledgments. This material is based upon work supported by the Army Research Office under Award No. W911NF-09-1-0273 and by NASA Contract No. NNA10DE79C . Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the Army Research Office or NASA.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley (1988)
2. Caires, L., Pfenning, F., Toninho, B.: Towards concurrent type theory. In: Pierce, B.C. (ed.) TLDI, pp. 1–12. ACM (2012)
3. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: DeMillo, R.A. (ed.) POPL, pp. 207–212. ACM Press (1982)
4. Demangeon, R., Honda, K.: Nested protocols in session types. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 272–286. Springer, Heidelberg (2012)
5. Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. Acta Inf. 42(2-3), 191–225 (2005)
6. Hennessy, M.: A Distributed Pi-Calculus. Cambridge University Press (2007)
7. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
8. Kawaguchi, M., Rondon, P.M., Jhala, R.: Type-based data structure verification. In: Hind, M., Diwan, A. (eds.) PLDI, pp. 304–315. ACM (2009)
9. Martin-Löf, P.: Intuitionistic type theory (1984)
10. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, i. Inf. Comput. 100(1), 1–40 (1992)

11. Mostrous, D., Yoshida, N., Honda, K.: Global principal typing in partially commutative asynchronous sessions. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 316–332. Springer, Heidelberg (2009)
12. Pierce, B.C.: Types and programming languages. MIT Press, Cambridge (2002)
13. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: Gupta, R., Amarasinghe, S.P. (eds.) PLDI, pp. 159–169. ACM (2008)
14. Rondon, P.M., Kawaguchi, M., Jhala, R.: Low-level liquid types. In: Hermenegildo, M.V., Palsberg, J. (eds.) POPL, pp. 131–144. ACM (2010)
15. Toninho, B., Caires, L., Pfenning, F.: Functions as session-typed processes. In: Birkedal, L. (ed.) FOSSACS 2012. LNCS, vol. 7213, pp. 346–360. Springer, Heidelberg (2012)
16. Yoshida, N., Vasconcelos, V.T.: Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. Electr. Notes Theor. Comput. Sci. 171(4), 73–93 (2007)