Correspondence Assertions for Process Synchronization in Concurrent Communications

Eduardo Bonelli [1,3]

Stevens Institute of Technology and LIFIA [6]

Adriana Compagnoni [1,4]

Stevens Institute of Technology

Elsa Gunter [1,2,5]

New Jersey Institute of Technology

Abstract

High-level specification of patterns of communications such as protocols can be modeled elegantly by means of session types [14]. However, a number of examples suggest that session types fall short when finer precision on protocol specification is required. In order to increase the expressiveness of session types we appeal to the theory of correspondence assertions [5,10]. The resulting type discipline augments the types of long term channels with effects and thus yields types which may depend on messages read or written earlier within the same session. We prove that evaluation preserves typability and that well-typed processes are safe. Also, we illustrate how the resulting theory allows us to address the shortcomings present in the pure theory of session types.

[1] This work was supported in part by the NSF Grant No. CCR-0220286 ITR:Secure Electronic Transactions.

[2] It was also supported in part by the ARO under Award No. DAAD-19-01-1-0473

[3] Email:ebonelli@cs.stevens-tech.edu

[4] Email:abc@cs.stevens-tech.edu

[5] Email:elsa@cis.njit.edu

[6] Faculty of Informatics, University of La Plata, La Plata, Argentina

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

Distributed and concurrent programming paradigms are increasingly popular, specially since the Internet entered the public domain. This has brought along new challenges including the specification and implementation of these programs together with techniques for the formal verification of their properties. One such specification method is that of protocol specification. This consists of identifying the sequence of message interchanges that take place between a number of parties in order to carry out some specific task. Recently, the use of type systems to formalize protocols has interested many researchers, in particular *session types* [13,14] has emerged as a promising approach. Interaction between a number of parties is achieved by specifying sequences of reciprocal interchanges of messages through private channels. Such sequences are modeled as types, the two parties at each end of the channel having *dual* such types. These pair of dual types constitute a *session type*. Session types are assigned to long term channels and are shared among processes. A long term channel is a *port* whose communication protocol is pre-specified. An example of a session type is:

$$(\downarrow \mathbf{Int}; \downarrow \mathbf{Int}; \uparrow \mathbf{Int} \ , \ \uparrow \mathbf{Int}; \uparrow \mathbf{Int}; \downarrow \mathbf{Int})$$

The first component, namely $\downarrow \mathbf{Int}; \downarrow \mathbf{Int}; \uparrow \mathbf{Int}$, indicates the expected behavior at one session point: the process must read an integer from the channel, then another one, and then write an integer to the channel (think of an "adding" server that reads in two numbers and writes out their sum). In order for the other party to interact correctly, it is assigned a *dual* type expression (the second component of the pair).

Quite some effort is being invested in the study of session types, motivated by the benefits that such a system provides for the analysis of protocols. Starting from the work of Honda et al [13], a suitable notion of subtypes for session types has been explored in [6], the benefits of session types in component based software development was presented in [21], bounded polymorphism in the presence of session types has been studied in [12], session types formulated in a λ-calculus with input/output operations is considered in [7].

This paper addresses a strengthening of session types by incorporating a theory of *correspondence assertions* (cf. Section 1.2). We shall address a number of examples in which the shortcomings of session types are illustrated and shall exhibit how correspondence assertions successfully overcome these difficulties. The resulting type discipline is strictly richer than the pure theory of session types. More precisely, a number of "unsafe" programs which are well-typed in the theory of pure session types shall be rejected by our typing rules. To the best of our knowledge, this is the first study of a theory of correspondence assertions for long term channels.

$\mathsf{Cl}(id, amt, a) = \texttt{request}\ a(k)\ \texttt{in}\ k![id];\ k \triangleleft \texttt{deposit};\ k![amt];\ k?(bal)\ \texttt{in}\ \texttt{stop}$

$\mathsf{ATM}(a, b) = \texttt{accept}\ a(k)\ \texttt{in}\ k?(idA)\ \texttt{in}$
$k \triangleright \{\ \texttt{deposit:}\ \texttt{request}\ b(h)\ \texttt{in}\ k?(amtA)\ \texttt{in}$
$h \triangleleft \texttt{deposit};\ h![idA];\ h![amtA];\ h?[balA]\ \texttt{in}\ k![balA];\ \mathsf{ATM}[a, b]$
$\Box\ \texttt{withdraw:}\ \texttt{request}\ b(h)\ \texttt{in}\ k?(amtA)\ \texttt{in}$
$h \triangleleft \texttt{withdraw};\ h![idA];\ h![amtA];\ h?(OKedAmtA)\ \texttt{in}$
$k![OKedAmtA];\ \mathsf{ATM}[a, b]\ \}$

$\mathsf{Bank}(b) = \texttt{accept}\ b(h)\ \texttt{in}$
$h \triangleright \{\ \texttt{deposit:}\ h?(idB)\ \texttt{in}\ h?(amtB)\ \texttt{in}\ updateData;\ h![balB];\ \mathsf{Bank}[b]$
$\Box\ \texttt{withdraw:}\ h?(idB)\ \texttt{in}\ h?(amtB)\ \texttt{in}$
$getOK\_AmtForIdB;\ h![OKedAmtB];\ \mathsf{Bank}[b]\ \}$

Fig. 1. The ATM example

1.1 Motivation

Consider the following example consisting of three parties: a Client, an ATM, and a Bank [14], as illustrated in Figure 1, which we briefly describe below:

The Client. A session is requested (through the shared name a), and then the Client sends its id number, selects a `deposit` operation, tells the amount of the deposit, and then waits for the new account balance.

The ATM. First it listens on name a for a client to request a session, then it reads in the client's id number (idA) and waits for the client's selection of one of two available operations: `deposit` or `withdraw`. In the case of a deposit operation, the ATM requests a session with the bank (on name b), reads in the amount the client wishes to deposit (from a) and then selects the `deposit` operation of the Bank. It then sends the Bank the client's id and the deposit amount, gets the new balance, reports it back to the client, and returns to the starting point. The ATM's `withdraw` operation is similar.

The Bank. It listens on name b (shared with the ATM) for requests for a session, and then waits for the ATM to indicate the operation it wishes to perform (either `deposit` or `withdraw`). If it is a `deposit` operation, it reads in the id and the amount, updates its data, sends back the new balance, and then returns to its starting point. In the case of `withdraw` it proceeds accordingly.

Let the expression ATM| Client| Bank denote the concurrent execution of the indicated parties. The type system presented in [14] asserts that this expression is well-typed. Indeed, assigning the following session types to a and b (where $\sigma(\alpha)$ is an abbreviation for the pair consisting of α and its dual)

we may type ATM| Client| Bank.

$$a : \sigma(\downarrow \mathbf{Int}; \&\{\texttt{deposit} :\downarrow \mathbf{Int}; \uparrow \mathbf{Int}; \mathbf{1}, \\ \Box\ \texttt{withdraw} :\downarrow \mathbf{Int}; \uparrow \mathbf{Int}; \mathbf{1}\})$$

$$b : \sigma(\&\{\texttt{deposit} :\downarrow \mathbf{Int}; \downarrow \mathbf{Int}; \uparrow \mathbf{Int}; \mathbf{1}, \\ \Box\ \texttt{withdraw} :\downarrow \mathbf{Int}; \downarrow \mathbf{Int}; \uparrow \mathbf{Int}; \mathbf{1}\})$$

The first type says that all communication sessions established on a must abide by the communication pattern described by the argument of σ on one endpoint and its dual on the other. The inner argument type may be read as follows: after an integer is input, wait for one of two operations from the opposite endpoint `deposit` or `withdraw`; if `deposit` is selected then input an integer, output an integer and disallow further communication, and likewise if the operation selected is `withdraw`.

Note that these types express how the long-term channels a and b behave *independently* of each other, even though they both belong to a common specification, namely that of the protocol specifying how Client, ATM, and Bank should interact in order to carry out a specific operation (a deposit or withdrawal). This may be witnessed as follows. Consider the ATM ATM′ resulting from ATM by replacing `deposit` with the following variant:

Example 1.1 [Deposit I]

`deposit`:

`request` $b(h)$`in` $k?(amtA)$ `in` $h \triangleleft$ `deposit`; $h![idA]$; $h![amtA - 1.5]$; **(1)**

$h?(balA)$ `in` $k![balA]$;

`request` $b(h')$ `in` $h' \triangleleft$ `deposit`; $h'![diffId]$; $h'![1.5]$; $h'?(balA')$ **(2)**

`in` ATM$[a, b]$

This version of the `deposit` operation deposits into the client's account 1.5 units fewer than the amount told by the Client **(1)**, and deposits the remaining 1.5 units in some account different from the client's, by means of a new deposit request **(2)** to the Bank, which was not present in the original ATM.

Unfortunately, this modified ATM is typable under the *same* type assumptions as the previous one. Likewise, if the `deposit` operation of the good ATM were replaced by the same one, except that the bank was not notified, then the resulting ATM also types under the same type assumptions as the good one.

Example 1.2 [Deposit II] The following variant of `deposit` allows the ATM to keep the deposit of the Client without depositing it in the account. If we call the resulting system ATM″, then ATM″| Client| Bank is well-typed under

exactly the same type assumptions as ATM| Client| Bank.

`deposit`: $k?(amtA)$ `in` $k![1000]$; ATM$[a, b]$

These examples suggest that although session types elegantly encode communication patterns of message interchange, they lack expressiveness in order to restrict interaction *between* sessions and also to enforce consistency of forwarded values (those received and then sent again). This paper introduces a type system based on *correspondence assertions* [22,10] in which ATM may be distinguished from the variants depicted above.

1.2 Correspondence Assertions

Correspondence assertions originated in the context of model-checking [22]. In [9] a type system for correspondence assertions is presented for the spi-calculus; a lucid account in the setting of an asynchronous π-calculus is presented by the same authors in [10]. Intuitively, correspondence assertions are used to formalize the idea that some point of execution in some process P must have been preceded by some other point of execution in some other process Q, in all possible executions of $P|\,Q$. Assertions are used to mark execution points in processes. As in [10], the assertions in this paper may have one of two forms: `begin` L or `end` L where L is an *assertion label*. A process is said to be *safe* if for every `end` L assertion reached in any execution, there is a corresponding `begin` L assertion which was reached sometime before, possibly in some other process.

By inserting appropriate correspondence assertions in untrusted code (including code communicating with the suspect code) and asking if the resulting code is safe, we may test for unexpected or malicious behavior in the communicating parties. Safety may be determined by a type system, hence allowing us to perform such checks statically.

Example 1.3 [Deposit I (continued)] Correspondence assertions allow us to show that the variant of ATM in Example 1.1 is unsafe if we assert that the amount to be deposited in the bank is the same as the amount given by the Client and appropriately augment the types of the sessions a and b. To show this, first we replace the code of Client by code including a begin assertion to obtain Client':

`request` $a(k)$ `in begin` $\langle id, amt\rangle$; $k![id]$; $k\triangleleft$`deposit`; $k![amt]$; $k?(bal)$ `in` `stop`

Note that the label of the `begin` assertion contains an occurrence of the expressions id and amt. These are values generated by the Client and passed to the ATM. Next we add an `end` assertion to the `deposit` operation of Bank (2) in Figure 1 obtaining Bank':

`deposit`: $h?(idB)$ `in` $h?(amtB)$ `in end` $\langle idB, amtB\rangle$;
$updateData$; $h![balB]$; Bank$[b]$

Finally, the session types of a and b are augmented with appropriate effects (see Figure 5 in Section 3) such that if the ATM requests a deposit operation to the bank and sends off some values for idB and $amtB$, then the incurred credit shall have to be paid off by a corresponding communication with the client: the client must have supplied these values.

The system ATM| Client'| Bank' shall be safe if every time the Bank's `deposit` operation is executed for an id number idB and amount $amtB$, the client requested the same operation on ATM, and $idB = id$, the id entered by the Client, and $amtA = amtB$, the amount entered by the Client.

We may address Example 1.2 similarly by forcing the ATM to engage in communication with the bank and, moreover, requiring that the `deposit` operation be selected. This is achieved by forcing the balance information sent by the ATM to the client to be retrieved from the bank. In this case, the `begin` assertion is inserted in the bank and the `end` assertion in the client. See [2] for full details.

The type rules we present in Section 2 show that the system of Example 1.3 is unsafe for the given correspondence assertions. The question of how the type system forces the `end` assertion in Bank' to be executed only after the corresponding `begin` assertion in Client' has been executed is answered by means of *latent effects* on channels. In order to "reach" the `end` assertion, the Bank' must have previously executed the read operations of `deposit` (i.e. $h?(idB)$ `in` $h?(amtB)$). Now, h is a channel which is shared between Bank' and ATM'| Client' (via ATM'). Via the placement of latent effects on the channel h, Bank' may pass back to whomever tries to send values on that channel the obligation of matching the `end` assertion. Similarly, ATM' can use latent effects on the channel it shares with the Client to further pass along the obligation. In fact, since the ATM' code has no assertions of its own, that is all it can do with the obligation. As the obligation is passed back through latent effects, it must be translated with respect to the substitution taking place as a result of the message passing on the channel. As the obligation is passed back from the Bank' to the ATM', it becomes $\langle idA, amtA - 1.5\rangle$, since these are the amounts sent for idB and $amtB$. As we pass the obligation back to Client, it is further transformed to $\langle id, amt - 1.5\rangle$, which does not match with the assertion `begin` $\langle id, amt\rangle$. We may conclude, therefore, that the program is not safe. It is worth noting that if we changed the `begin` assertion to `begin` $\langle id, amt - 1.5\rangle$, then the program would type check and be declared safe. We would, in effect, be acknowledging that ATM' had a right to charge a 1.5 unit fee for a deposit transaction.

Contribution. In this paper we introduce a type-based theory of correspondence assertions for session types.

- In contrast to previous type systems for such assertions, session types allow the effects of an input/output type to depend on messages which were interchanged prior in the same session. We also include the branching/selection

and delegation constructs from [14] in our analysis. The resulting type system shall allow us to distinguish the three above-mentioned variants of the ATM. This is achieved by introducing appropriate type directives (i.e. assertions) in the code and assigning appropriate types to names and channels, and then type checking using the type discipline presented in this paper.

- We define a new type system of *dependent session types* combining session types and correspondence assertions. This combination introduces a number of technical difficulties. For example, the usual representation of environments as sequences of assumptions [1,10] fails to yield a calculus satisfying some standard basic properties (cf. Remark 2.6).
- We show that evaluation preserves typability and that processes typable under empty effects are safe.

Related work. This work may be included among others in which type systems for the π-calculus are studied [18,16,17,20]. Subtyping is introduced in the setting of session types in [6]; however, the concept of synchronization between sessions is not explored. The works [23] and [19] do not explore session types either: the first studies a typing scheme for processes based on graph types and the second a type system for restricting communication in concurrent objects; their relation to session types is discussed in [14]. While [10] shares a fair amount in common with this work, there is a major difference. In [10] dependencies in types are "horizontal" in the sense that in a type expression such as $\downarrow [x : T_1, y : T_2]$ the type of y may depend on the value of x, this being fixed for *all* communications over a channel of this type. However, since our setting is that of session types we allow "vertical" dependencies of the form $\downarrow [x : T_1];\downarrow [y : T_2];\downarrow [z : T_3]$. In this case, the type of the value read for z may depend on either or both of the values read for x and y. These latter values are read in the *same* channel, but prior in time to z. Thus, in the present work, dependency spans whole sessions. Recently, type systems where CCS-like processes are used for typing process expressions have appeared. The generic type system of [15] is an example, although it does not incorporate correspondence assertions (however see Section 4). Another approach is [4] in which models (types as CCS-processes) of π-calculus expressions are obtained and the validity of temporal formulas are analyzed through model-checking techniques in order to deduce properties of the process expressions. They propose a type-and-effect system which incorporates correspondence assertions, however no long term channel types are available.

Structure of the paper. Section 2 defines π_s, a system combining session types [14] and correspondence assertions [10]. Section 2.2.1 presents a type system with effects for π_s. The proof of safety is given in Section 3 by introducing an appropriate labeled transition semantics. Finally we conclude and suggest further research directions.

2 The π_s-Calculus

2.1 Syntax

This section describes the syntax of π_s. We begin with a set of *names* $x, y, z, \ldots$. We distinguish two distinct kinds of names: *expression names*, for which we will use $a, b, c, \ldots$ (and which range over sessions and integers); and *channel names*, for which we will use $k, h, k', \ldots$. We also have *integer constants* $\ldots, -1, 0, 1, \ldots$, (branching) *labels* $l, l', \ldots$ and *process variables* written $X, Y, \ldots$. A *value* is an expression name or an integer constant and is denoted with letters $v, v', \ldots$. Assertion labels, written $L, L', \ldots$, are tuples of values and are written $\langle v_1, \ldots, v_n \rangle$. Process expressions, denoted with $P, Q, \ldots$, are defined as follows:

$$\begin{aligned} P ::= {} & \texttt{request}\ a(k)\ \texttt{in}\ P \mid \texttt{accept}\ a(k)\ \texttt{in}\ P \mid k?(x)\,\texttt{in}\ P \mid k![v]; P \mid \\ & \texttt{throw}\ k[k']; P \mid \texttt{catch}\ k(k')\ \texttt{in}\ P \mid (\nu a : T)P \mid (\nu k : \bot_e)P \mid \\ & k \lhd l; P \mid k \rhd \{l_1 : P_1 \Box \ldots \Box\ l_n : P_n\} \mid \texttt{stop} \mid P \,|\, Q \mid \\ & \texttt{def}\ D\ \texttt{in}\ P \mid X[\vec{v}] \mid \texttt{begin}\ L; P \mid \texttt{end}\ L; P \end{aligned}$$

Process definitions D take the form $X_1[\vec{a_1}] = P_1$ `and` $\ldots$ `and` $X_n[\vec{a_n}] = P_n$.

Remark 2.1 Parentheses are binding constructs. The notation $\vec{v}$ stands for $v_1, \ldots, v_n$, and likewise for $\vec{a_i}$ with $i \in 1..n$. Any two process expressions which differ only in the names of their bound names (called α-equivalent) shall be considered equal. We use the notation $P\{a \leftarrow v\}$ for the result of substituting all free occurrences of a in P by v, and similarly for $P\{k \leftarrow k'\}$. Note that for the benefit of a clear presentation we have chosen to present a monadic calculus; an extension to the polyadic case should be straightforward.

The `request` primitive requests a session on name a. When this session is established the fresh private channel k shall be used for message interchange. The `accept` receives a request on the same name a and generates a new private channel for message interchange to be used once the session is established. The `request` and `accept` constructs each bind all free occurrences of the immediately following channel variable, k, in the subsequent process, P. The synchronous sending and receiving of messages is achieved with $k![v]; Q$ and $k?(x)$ `in` P respectively, although, as in [14], a translation to an asynchronous calculus with branching is possible. Controlled side-stepping of linearity constraints on channel usage is achieved by means of the channel delegation constructs `throw` $k[k']; P$ and `catch` $k(k')$ `in` Q. Mechanisms for selection of a label and branching are available as $k \lhd l; P$ and $k \rhd \{l_1 : P_1 \Box \ldots \Box\ l_n : P_n\}$. The notation $P \,|\, Q$ has already been explained; we also use `stop` for inaction. We write $(\nu a : T)P$ or $(\nu k : \bot_e)P$ for the usual constructs for name hiding, where the former is for expression names and the latter for channel names. T denotes a type expression (Definition 2.2) and $\bot_e$ is the "complete" channel type with effect e. Definitions of processes are also allowed through

the `def` D `in` P construct, possibly introducing recursion. The `begin` and `end` assertions shall be used as type directives in the type system for π_s (Section 2.2.1): `begin` $L; P$ simply asserts `begin` L and then behaves as P; likewise `end` $L; P$ asserts `end` L and then behaves as P.

2.2 The Type Discipline

The present section enriches the type system of [14] with correspondence assertions in order to address the shortcomings mentioned in the introduction.

2.2.1 Session types and effects

The type system shall assign an effect to a process under a given set of type assumptions. The effect of a process reflects its pending obligations. An assertion of the form `begin` L shall reduce these obligations by withdrawing the assertion label L from the current effect; likewise `end` L shall augment the current effect with L. Thus effects determine lower-bounds of the number of `begin` assertions that must be present. If the process has an empty effect, then all `end` assertions correspond to a matching `begin` assertion.

As explained above, effects also have to be attached to channel types in order for two or more processes to share information on their pending or latent effects. Effects added to channels are thus called *latent effects.*

Definition 2.2 [Types with Effects] Assertion labels, effects and types are given by the following grammar:

$$
\begin{array}{lll}
\textit{Plain Type} & T & ::= \mathbf{Int} \mid \sigma(\alpha) \\
\textit{Channel Type} & \alpha, \beta & ::= \downarrow[a:T]e;\alpha \mid \uparrow[a:T]e;\alpha \mid \downarrow[\alpha]e;\beta \\
 & & \quad \mid \uparrow[\alpha]e;\beta \mid \&\{l_1:\alpha_1,\ldots,l_n:\alpha_n\}e \\
 & & \quad \mid \oplus\{l_1:\alpha_1,\ldots,l_n:\alpha_n\}e \mid \mathbf{1} \mid \bot_e \\
\textit{Effect} & e, e' & ::= (\!| L_1,\ldots,L_n |\!) \\
\textit{Assertion Label} & L, L_i & ::= \langle v_1,\ldots,v_n\rangle
\end{array}
$$

A *type* is either a plain type or a channel type; we use U, U_i to range over types. The set of free names of a type U, written $\mathtt{fn}(U)$, is defined as usual (see [2]). The base type **Int** is the type of integer constants. Session types are represented as $\sigma(\alpha)$ and may informally be seen to denote a pair consisting of a channel type α and its dual $\overline{\alpha}$:

$$
\begin{array}{llll}
\overline{\downarrow[a:T]e;\alpha} \stackrel{def}{=} \uparrow[a:T]e;\overline{\alpha} & \overline{\uparrow[a:T]e;\alpha} \stackrel{def}{=} \downarrow[a:T]e;\overline{\alpha} & \overline{\mathbf{1}} \stackrel{def}{=} \mathbf{1} \\
\overline{\downarrow[\alpha]e;\beta} \stackrel{def}{=} \uparrow[\alpha]e;\overline{\beta} & \overline{\uparrow[\alpha]e;\beta} \stackrel{def}{=} \downarrow[\alpha]e;\overline{\beta} & \\
\overline{\&\{l_i:\alpha_i\}e} \stackrel{def}{=} \oplus\{l_i:\overline{\alpha_i}\}e & \overline{\oplus\{l_i:\alpha_i\}e} \stackrel{def}{=} \&\{l_i:\overline{\alpha_i}\}e &
\end{array}
$$

The types α and $\overline{\alpha}$ shall be assigned to the two endpoints of a communication

session. Note that $\overline{\perp_e}$ is not defined. A channel type consists of a sequence of input/output types of values or channels, or branch/selection types; the sequence is assumed to terminate with the channel type terminator **1**. Each of these is accompanied by a latent *effect*. An effect is a multi-set of assertion labels; we use $(\!| \ldots |\!)$ for the multi-set constructor. Multiset subtraction is defined as $e \setminus e'$, the smallest multiset e'' such that $e \leq e' + e''$, where "+" is multiset union. The special channel type $\perp_e$ models a "complete" or "closed" channel which is already being used by two existing endpoints and thus through which no further communication is possible (cf. Definition 2.5).

2.2.2 Typing Rules:
An *environment* Γ is a set of type assumptions $x_1 : U_1 \cdot \ldots \cdot x_n : U_n$ where $x_1, \ldots, x_n$ are distinct names. We use letters $\Gamma, \Delta, \ldots$ for environments. The domain of Γ, written $\mathtt{dom}(\Gamma)$, is the set $\{x_1, \ldots, x_n\}$. Also, we write $\mathtt{domCh}(\Gamma)$ for the subset of names to which Γ assigns channel types and $\mathtt{domPl}(\Gamma)$ for the subset of names to which Γ assigns plain types. In an assumption $x : U$, x is called the subject; if the type assigned to the subject is a plain type then the assumption is said to be a *plain assumption*, otherwise it is a *channel assumption*. We write $\Gamma \cdot x : U$ for the environment resulting from extending Γ with the type assumption $x : U$ for $x \notin \mathtt{dom}(\Gamma)$. The notation $\Gamma \setminus x : U$ stands for the environment resulting from dropping the assumption $x : U$ from Γ (assuming it exists).

Definition 2.3 [Depends on] $x_i : U_i$ *depends directly on* $x_j : U_j$ *in* Γ (written $(x_j : U_j) \hookrightarrow_d (x_i : U_i)$), if $x_j \in \mathtt{fn}(U_i)$. We say $x_i : U_i$ *depends on* $x_j : U_j$ *in* Γ if $x_i : U_i \hookrightarrow x_j : U_j$, where $\hookrightarrow$ denotes the transitive closure of $\hookrightarrow_d$.

We say that an environment is be *well-formed* if it satisfies the following two conditions:

C1. For each $i \in 1..n$, $\mathtt{fn}(U_i) \subseteq \mathtt{dom}(\Gamma) \setminus \{x_i\}$.

C2. $\hookrightarrow$ is irreflexive [7].

Condition **C1** requires that all free names in types assigned by Γ must be declared within Γ. Note that since channel names may not appear in assertion labels (hence not in $\mathtt{fn}(U_i)$), types may only depend on names which are assigned plain types. Since interaction through channel names is restricted by linearity conditions in the sense of linear logic [8] (see explanation of Type Par rule below), this restriction states that we do not allow types depending on linear assumptions (we do however allow types depending on plain or "intuitionistic" assumptions). The intended application of our type discipline is not disturbed by such a restriction, and it is not clear whether the technical complications of the meta-theory resulting from lifting it outweigh its benefits. In fact this restriction already appears in other settings in which linear and

[7] $R \subseteq A \times A$ is *irreflexive* iff for every $x \in A$ it is not the case that xRx.

intuitionistic assumptions coexist, such as the linear logical framework of [3]. The second condition, **C2**, requires that Γ have no cyclic dependencies. This is usually guaranteed by the representation of environments as sequences of type assumptions, in which an assumption $x : U$ depends only on those appearing to its left. Such a representation seems unfit in a setting where channel types are present since basic results on admissibility of structural rules fail (Remark 2.6).

The π_s type system consists of the following four *judgements*:

$\Gamma \vdash_\Theta \diamond$	well-formed environment Γ and process protocol Θ
$\Gamma \vdash_\Theta v : T$	well-typed value v of type T
$\Gamma \vdash_\Theta (\vec{v}) : (\vec{a} : \vec{T})$	well-typed process parameters $\vec{v}$ of type $(\vec{a} : \vec{T})$
$\Gamma \vdash_\Theta P : e$	well-typed process P with effect e

The letter Θ stands for a *process protocol*: a set of expressions of the form $X_j : (\vec{a}_j : \vec{T}_j)$, for $j \in 1..n$, where each $\vec{a}_j : \vec{T}_j$ is an environment indicating the types of process parameters to X_j. The judgement $\Gamma \vdash_\Theta \diamond$ holds if Γ is a well-formed environment, and also each environment $\vec{a}_j : \vec{T}_j$ in the process protocol Θ is well-formed.

The type rules of π_s are presented in Figure 2. The rules Type Acpt and Type Rcv introduce a new channel name in the environment, thus guaranteeing that a private channel is being used for the session. Note that dual channel types are used for the requesting and accepting parties. Type Bgn and Type End affect process effects by eliminating or adding a new assertion label. The rules Type Snd and Type Rcv allow the typing of the communication primitives for sending and receiving data. Note that data is sent and received over channels only. Also, note that the type of k in the upper right-hand judgement of Type Snd is $\alpha\{a \leftarrow v\}$, reflecting the fact that the "rest" of the channel type, namely α, may depend on the output value v. The same comment applies to the Type Rcv rule. Type Brnch and Type Sel type the branching and selection primitives, respectively; if pending effects are seen as credits, then it is clear that the effects of each branch in Type Brnch must be joined. Channel delegation is achieved by means of the throw and catch primitives, which are typed by means of Type Thr and Type Cat. The rule Type Thr is subject to the restriction that $\beta \neq \mathbf{1}$; this restricts delegation of channels to those through which communication is possible, i.e. no "dead" channels[8]. Channel and name restriction (for non-channel names) are typed as expected. Type Stop types the inaction `stop`; it requires all communication through channel names to have been completed. The Type Subsum rule allows increasing the required assertion obligations of a process term. The Type Par rule types the parallel execution of two processes. A channel may be used

[8] Technically, this allows us to correct a problem present in [14], namely the failure of Subject Congruence.

by one of the two processes P or Q. The only exception to this rule is when both P and Q use a channel k of dual types. Since channel usage must be restricted in order to guarantee such linear usage, the environments Γ and Γ' are required to be *compatible*.

Definition 2.4 [Compatibility $\asymp$] The relation $\asymp$ is defined as follows: $\emptyset \asymp \emptyset$, and $\Gamma \asymp \Gamma'$ implies

(i) $\Gamma \cdot a : T \asymp \Gamma' \cdot a : T$

(ii) $\Gamma \cdot k : \alpha \asymp \Gamma' \cdot k : \overline{\alpha}$

(iii) $\Gamma \cdot k : \alpha \asymp \Gamma'$, if $k \notin \mathtt{dom}(\Gamma')$

(iv) $\Gamma \asymp \Gamma' \cdot k : \alpha$, if $k \notin \mathtt{dom}(\Gamma)$

Note that the notion of compatibility makes sense for two sets of assumptions which do not necessarily constitute well-formed environments. Once this notion of compatibility is in place we may define how two environments are combined through environment *composition*.

Definition 2.5 [Composition $\circ$] Let Γ, Γ' be two environments such that $\Gamma \asymp \Gamma'$. We define $\Gamma \circ \Gamma'$ as follows: $\emptyset \circ \emptyset = \emptyset$ and

(i) $(\Gamma \cdot a : T) \circ (\Gamma' \cdot a : T) = (\Gamma \circ \Gamma') \cdot a : T$

(ii) $(\Gamma \cdot k : \alpha) \circ (\Gamma' \cdot k : \overline{\alpha}) = (\Gamma \circ \Gamma') \cdot k : \bot_{\mathtt{fnMult}(\alpha)}$

(iii) $(\Gamma \cdot k : \alpha) \circ (\Gamma') = (\Gamma \circ \Gamma') \cdot k : \alpha$, if $k \notin \mathtt{dom}(\Gamma')$

(iv) $\Gamma \circ (\Gamma' \cdot k : \alpha) = (\Gamma \circ \Gamma') \cdot k : \alpha$, if $k \notin \mathtt{dom}(\Gamma)$

The effect $\mathtt{fnMult}(\alpha)$ is the multiset which includes a label for each occurrence of a free name in α. Other variants for the second clause of Definition 2.5 are possible as long as the effect subscript of $\bot$ faithfully records the name dependencies of the dual channel types from which it arises (i.e. no dependency information is lost).

For the sake of readability, in Figure 2, we have omitted the hypotheses that the environment of the conclusion of the rule be well-formed, for all those rules where the environment of the conclusion is different from the environment of all hypothesis. Note that in some of the latter rules the condition is superfluous, namely Type CRes, Type Par, Type Subsum, Type PVar and Type Def.

Remark 2.6 A representation of environments based on sequences of hypothesis, as usually adopted in the literature on dependent type systems [1], is not applicable to our system. The reason is that basic results on the admissibility of structural rules fail. In particular, the Exchange Lemma, which states that the order of independent hypothesis is irrelevant for the sake of derivability, fails. Indeed, consider the following possible type rule Type Snd formulated in

$$\frac{\Gamma \cdot a : \sigma(\alpha) \cdot k : \alpha \vdash_\Theta P : e}{\Gamma \cdot a : \sigma(\alpha) \vdash_\Theta \texttt{accept}\ a(k)\ \texttt{in}\ P : e}\ \text{Type Acpt}$$

$$\frac{\Gamma \cdot a : \sigma(\alpha) \cdot k : \overline{\alpha} \vdash_\Theta P : e}{\Gamma \cdot a : \sigma(\alpha) \vdash_\Theta \texttt{request}\ a(k)\ \texttt{in}\ P : e}\ \text{Type Requ}$$

$$\frac{\Gamma \vdash_\Theta P : e \quad \texttt{fn}(L) \subseteq \texttt{dom}(\Gamma)}{\Gamma \vdash_\Theta \texttt{begin}\ L; P : e \setminus (\!| L |\!)}\ \text{Type Bgn} \qquad \frac{\Gamma \vdash_\Theta P : e \quad \texttt{fn}(L) \subseteq \texttt{dom}(\Gamma)}{\Gamma \vdash_\Theta \texttt{end}\ L; P : e + (\!| L |\!)}\ \text{Type End}$$

$$\frac{\Gamma \vdash_\Theta v : T \quad \Gamma \cdot k : \alpha\{a \leftarrow v\} \vdash_\Theta P : e}{\Gamma \cdot k :\uparrow [a : T]e'; \alpha \vdash_\Theta k![v]; P : e + e'\{a \leftarrow v\}}\ \text{Type Snd}$$

$$\frac{\Gamma \cdot a : T \cdot k : \alpha \vdash_\Theta P : e \quad \texttt{fn}(e \setminus e') \subseteq \texttt{dom}(\Gamma)}{\Gamma \cdot k :\downarrow [a : T]e'; \alpha \vdash_\Theta k?(y)\ \texttt{in}\ P : e \setminus e'}\ \text{Type Rcv}$$

$$\frac{\Gamma \cdot k : \alpha_1 \vdash_\Theta P_1 : e_1 \ldots\ \Gamma \cdot k : \alpha_n \vdash_\Theta P_n : e_n}{\Gamma \cdot k : \&\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\}e' \vdash_\Theta k \rhd \{l_1 : \alpha_1, \ldots, l_n : \alpha_n\} : (\bigvee e_i) \setminus e'}\ \text{Type Brnch}$$

$$\frac{\Gamma \cdot k : \alpha_j \vdash_\Theta P : e \quad (1 \leq j \leq n)}{\Gamma \cdot k : \oplus\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\}e' \vdash_\Theta k \lhd l_j; P : e + e'}\ \text{Type Sel}$$

$$\frac{\Gamma \cdot k : \alpha \vdash_\Theta P : e}{\Gamma \cdot k' : \beta \cdot k :\uparrow [\beta]e'; \alpha \vdash_\Theta \texttt{throw}\ k[k']; P : e + e'}\ \text{Type Thr}$$

$$\frac{\Gamma \cdot k' : \beta \cdot k : \alpha \vdash_\Theta P : e}{\Gamma \cdot k :\downarrow [\beta]e'; \alpha \vdash_\Theta \texttt{catch}\ k(k')\ \texttt{in}\ P : e \setminus e'}\ \text{Type Cat}$$

$$\frac{}{\texttt{ranCh}(\Gamma) \subseteq \{\mathbf{1}, \bot_e\}}\ \Gamma \vdash_\Theta \texttt{stop} : (\!|\,|\!)\ \text{Type Stop} \qquad \frac{\Gamma \cdot a : T \vdash_\Theta P : e}{\Gamma \vdash_\Theta (\nu a : T)P : e}\ \text{Type NRes}$$

$$\frac{\Gamma \cdot k : \bot_{e'} \vdash_\Theta P : e}{\texttt{fn}(e') \subseteq \texttt{dom}(\Gamma)}\ \Gamma \vdash_\Theta (\nu k : \bot_{e'})P : e\ \text{Type CRes} \qquad \frac{\Gamma \vdash_\Theta P : e \quad \Gamma' \vdash_\Theta Q : e' \quad \Gamma \asymp \Gamma'}{\Gamma \circ \Gamma' \vdash_\Theta P | Q : e + e'}\ \text{Type Par}$$

$$\frac{\Gamma \vdash_\Theta P : e \quad e \leq e' \quad \texttt{fn}(e') \subseteq \texttt{dom}(\Gamma)}{\Gamma \vdash_\Theta P : e'}\ \text{Type Subsum}$$

$$\frac{\Gamma \vdash_\Theta (\vec{v}) : (\vec{a} : \vec{T}) \quad X : (\vec{a} : \vec{T}) \in \Theta \quad \texttt{ranCh}(\Gamma) \subseteq \{\mathbf{1}, \bot_e\}}{\Gamma \vdash_\Theta X[\vec{v}] : (\!|\,|\!)}\ \text{Type PVar}$$

$$\frac{\Gamma \setminus \texttt{chan}(\Gamma) \cdot \vec{a_i} : \vec{T_i} \vdash_\Theta P_i : (\!|\,|\!) \quad \Theta(X_i) = (\vec{a_i} : \vec{T_i}) \quad \Gamma \vdash_\Theta Q : e}{\Gamma \vdash_{\Theta \setminus \vec{X}} \texttt{def}\ X_1(\vec{a_1}) = P_1 \ldots \texttt{and} \ldots X_n(\vec{a_n}) = P_n\ \texttt{in}\ Q : e}\ \text{Type Def}$$

Fig. 2. Well-formed process expressions

$$\frac{\Gamma \cdot a : T \vdash_\Theta \diamond}{\Gamma \cdot a : T \vdash_\Theta a : T}\ \text{Wf Val Name} \qquad \frac{\Gamma \vdash_\Theta \diamond \quad n \in \mathbf{Z}}{\Gamma \vdash_\Theta n : \mathbf{Int}}\ \text{Wf Val Int}$$

$$\frac{\Gamma \vdash_\Theta \diamond}{\Gamma \vdash_\Theta () : ()}\ \text{Wf PP Nil} \qquad \frac{\Gamma \vdash_\Theta (\vec{v}) : (\vec{a} : \vec{T}) \quad \Gamma \vdash_\Theta v : T\{\vec{a} \leftarrow \vec{v}\} \quad b \notin \{\vec{a}\} \cup \mathtt{dom}(\Gamma)}{\Gamma \vdash_\Theta (\vec{v}, v) : (\vec{a} : \vec{T}, b : T)}\ \text{Wf PP Cons}$$

Fig. 3. Well-formed values and process parameters

a setting where environments are sequences:

$$\frac{\Gamma_1 \cdot \Gamma_2 \vdash_\Theta v : T \quad \Gamma_1 \cdot k : \alpha\{a \leftarrow v\} \cdot \Gamma_2 \vdash_\Theta P : e \quad \Gamma_1 \cdot k :\uparrow [a : T]e'; \alpha \cdot \Gamma_2 \vdash_\Theta \diamond}{\Gamma_1 \cdot k :\uparrow [a : T]e'; \alpha \cdot \Gamma_2 \vdash_\Theta k![v]; P : e + e'\{a \leftarrow v\}}$$

Assume that $\Gamma_1 = \Gamma_1' \cdot v : T$. Then note that $v : T$ and $k :\uparrow [a : T]e'; \alpha$ satisfy the condition of the Exchange Lemma, since neither one depends on the other. However, when we attempt to exchange $v : T$ and $k : \alpha\{a \leftarrow v\}$ in the upper middle judgement we fail, since $\alpha\{a \leftarrow v\}$ may have free occurrences of v. Note that these issues do not appear in previous type-theoretic formulations of correspondence assertions for concurrent/distributed calculi since long-term session types are not considered.

3 Safety Proof for π_s

In order to trace the execution of certain actions such as `begin` and `end` assertions, we shall introduce a labeled transition semantics [10] (LTS) for π_s. The LTS is defined modulo structural congruence $\equiv$ and shall be used for formalizing the notion of *safe process* and showing that all typable processes with null effects are safe. The *actions* of the transition system, denoted with letters $\psi, \phi, \ldots$, are:

- $P \xrightarrow{\mathtt{begin}\ L} P'$ P reaches a `begin` L assertion.
- $P \xrightarrow{\mathtt{end}\ L} P'$ P reaches a `end` L assertion.
- $P \xrightarrow{\mathtt{res}(a : T)} P'$ P generates a new *session name* a.
- $P \xrightarrow{\mathtt{res}(k : \bot_e)} P'$ P generates a new *channel name* k.
- $P \xrightarrow{\tau} P'$ P performs an internal action.

Thus the set of actions is `begin` L, `end` L, $\mathtt{res}(a : T), \mathtt{res}(k : \bot_e), \tau$. The labeled transition system for π_s is given in Figure 4; we write $P \xrightarrow{\psi} P'$ when P reduces to P' through action ψ. The same figure defines the *free* and *generated names* of an action.

A sequence of transitions may be tracked with traces. A *trace* s is a sequence $\psi_1 \ldots \psi_n$ of actions. We use ϵ for the empty trace. The free names

$$\begin{array}{lll|lll}
\mathtt{fn}(\tau) & \stackrel{def}{=} & \emptyset & \mathtt{gn}(\tau) & \stackrel{def}{=} & \emptyset \\
\mathtt{fn}(\mathtt{begin}\ L) & \stackrel{def}{=} & \mathtt{fn}(L) & \mathtt{gn}(\mathtt{begin}\ L) & \stackrel{def}{=} & \emptyset \\
\mathtt{fn}(\mathtt{end}\ L) & \stackrel{def}{=} & \mathtt{fn}(L) & \mathtt{gn}(\mathtt{end}\ L) & \stackrel{def}{=} & \emptyset \\
\mathtt{fn}(\mathtt{res}(a:T)) & \stackrel{def}{=} & \{a\} \cup \mathtt{fn}(T) & \mathtt{gn}(\mathtt{res}(a:T)) & \stackrel{def}{=} & \{a\} \\
\mathtt{fn}(\mathtt{res}(k:\bot_e)) & \stackrel{def}{=} & \{k\} \cup \mathtt{fn}(e) & \mathtt{gn}(\mathtt{res}(k:\bot_e)) & \stackrel{def}{=} & \{k\}
\end{array}$$

$$(\mathtt{accept}\ a(k)\ \mathtt{in}\ P_1) \mid (\mathtt{request}\ a(k)\ \mathtt{in}\ P_2) \xrightarrow{\tau} (\nu k:\bot_e)(P_1 \mid P_2) \quad \text{Trans Link}$$

$$(k![v]; P_1) \mid (k?(a)\,\mathtt{in}\ P_2) \xrightarrow{\tau} P_1 \mid P_2\{a \leftarrow v\} \quad \text{Trans Comm}$$

$$(k \lhd l_i; P) \mid (k \rhd \{l_1 : P_1 \Box \ldots \Box\, l_n : P_n\}) \xrightarrow{\tau} P \mid P_i, \text{ if } i \in 1..n \quad \text{Trans Brnch}$$

$$(\mathtt{throw}\ k[k']; P_1) \mid (\mathtt{catch}\ k(k'')\ \mathtt{in}\ P_2) \xrightarrow{\tau} P_1 \mid P_2\{k'' \leftarrow k'\} \quad \text{Trans Catch}$$

$$\mathtt{def}\ D\ \mathtt{in}\ (X[\vec{v}] \mid Q) \xrightarrow{\tau} \mathtt{def}\ D\ \mathtt{in}\ (P\{\vec{a} \leftarrow \vec{v}\} \mid Q), \quad \text{Trans Def1}$$

$$\text{if } X(\vec{a}) = P \in D$$

$$\mathtt{begin}\ L; P \xrightarrow{\mathtt{begin}\ L} P \quad \text{Trans Begin}$$

$$\mathtt{end}\ L; P \xrightarrow{\mathtt{end}\ L} P \quad \text{Trans End}$$

$$(\nu a:T)P \xrightarrow{\mathtt{res}(a:T)} P \quad \text{Trans ResN}$$

$$(\nu k:\bot_e)P \xrightarrow{\mathtt{res}(k:\bot_e)} P \quad \text{Trans ResCh}$$

$$\frac{P \xrightarrow{\psi} P'}{\mathtt{def}\ D\ \mathtt{in}\ P \xrightarrow{\psi} \mathtt{def}\ D\ \mathtt{in}\ P'}\ \text{Trans Def2}$$

$$\frac{P \xrightarrow{\psi} P'}{P \mid Q \xrightarrow{\psi} P' \mid Q}\ \text{Trans Par, if } \mathtt{gn}(\psi) \cap \mathtt{fn}(Q) = \emptyset$$

$$\frac{P \equiv P' \quad P' \xrightarrow{\psi} Q' \quad Q' \equiv Q}{P \xrightarrow{\psi} Q}\ \text{Trans} \equiv$$

Fig. 4. LTS for π_s

(resp. generated names) of a trace $\psi_1 \ldots \psi_n$ are defined as $\mathtt{fn}(\psi_1) \cup \ldots \cup \mathtt{fn}(\psi_1)$ (resp. $\mathtt{gn}(\psi_1) \cup \ldots \cup \mathtt{gn}(\psi_1)$). A traced transition is a sequence of actions:

Definition 3.1 [Traced Transitions] P reduces to P' with trace s if $P \xrightarrow{s} P'$, where $\xrightarrow{s}$ is defined as:

$$P \equiv P' \Rightarrow P \xrightarrow{\epsilon} P' \qquad \text{Trace} \equiv$$

$$P \xrightarrow{\psi} Q, Q \xrightarrow{s} P' \Rightarrow P \xrightarrow{\psi s} P' \ \text{Trace Action}, \quad \text{where } \mathtt{fn}(\psi) \cap \mathtt{gn}(s) = \emptyset$$

In order to define when a process is *safe* we shall need to count the number of `begin`'s and `end`'s in traces. The former is defined as $\mathtt{begins}(\psi_1 \ldots \psi_n) \stackrel{def}{=}$

$\mathtt{begins}(\psi_1) \cup \ldots \cup \mathtt{begins}(\psi_n)$ and the latter $\mathtt{ends}(\psi_1 \ldots \psi_n) \stackrel{def}{=} \mathtt{ends}(\psi_1) \cup \ldots \cup \mathtt{ends}(\psi_n)$, where $\cup$ stands for multi-set union and

$$\begin{array}{llll} \mathtt{begins}(\mathtt{begin}\ L) & \stackrel{def}{=} (\!|\, L \,|\!) & \mathtt{ends}(\mathtt{begin}\ L) & \stackrel{def}{=} (\!|\,|\!) \\ \mathtt{begins}(\mathtt{end}\ L) & \stackrel{def}{=} (\!|\,|\!) & \mathtt{ends}(\mathtt{end}\ L) & \stackrel{def}{=} (\!|\, L \,|\!) \\ \mathtt{begins}(\mathtt{res}(u)) & \stackrel{def}{=} (\!|\,|\!) & \mathtt{ends}(\mathtt{res}(u)) & \stackrel{def}{=} (\!|\,|\!) \\ \mathtt{begins}(\tau) & \stackrel{def}{=} (\!|\,|\!) & \mathtt{ends}(\tau) & \stackrel{def}{=} (\!|\,|\!) \end{array}$$

Definition 3.2 [Safe Process] A process P is *safe* if and only if for all traces s and processes P', if $P \stackrel{s}{\longrightarrow} P'$ then $\mathtt{ends}(s) \leq \mathtt{begins}(s)$.

Thus a process is safe if every $\mathtt{end}\ L$ is accounted for by a corresponding $\mathtt{begin}\ L$. For example, $\mathtt{begin}\ L; \mathtt{stop}$ is safe, however $\mathtt{begin}\ L; \mathtt{end}\ L; \mathtt{end}\ L; \mathtt{stop}$ is not. We now address the proof of safety, namely that a process typable with null effect is safe. This requires first showing that process reduction preserves typings and effects.

Theorem 3.3 *Assume* $\Gamma \vdash_\Theta P : e$.

(i) *(Subject Congruence) If* $P \equiv Q$, *then* $\Gamma \vdash_\Theta Q : e$.

(ii) *(Subject Reduction)*

(a) *If* $P \stackrel{\tau}{\longrightarrow} P'$, *then* $\Gamma' \vdash_\Theta P' : e$, *where* Γ' *and* Γ *differ only in the effects assigned to the channel type* $\bot$ *(if any).*

(b) *If* $P \stackrel{\mathtt{begin}\ L}{\longrightarrow} P'$, *then* $\Gamma \vdash_\Theta P' : e + (\!|\, L \,|\!)$.

(c) *If* $P \stackrel{\mathtt{end}\ L}{\longrightarrow} P'$, *then* $\Gamma \vdash_\Theta P' : e \setminus (\!|\, L \,|\!)$ *and* $L \in e$.

(d) *If* $P \stackrel{\mathtt{res}(a:T)}{\longrightarrow} P'$ *and* $a \notin \mathtt{dom}(\Gamma)$, *then* $\Gamma \cdot a : T \vdash_\Theta P' : e$.

(e) *If* $P \stackrel{\mathtt{res}(k:\bot_f)}{\longrightarrow} P'$ *and* $k \notin \mathtt{dom}(\Gamma)$, *then* $\Gamma \cdot k : \bot_f \vdash_\Theta P' : e$.

Subject Congruence is proved by induction on the derivation of $P \equiv Q$; the fact that effects are not lost when environments are composed (Def. 2.5) is crucial to its proof. Subject reduction is proved by cases according to the action which takes place (see [2]). Finally, we may put the results together and obtain the main result. Its proof is based upon observing that the following invariant holds: If $\Gamma \vdash_\Theta P : e$ and $P \stackrel{s}{\longrightarrow} P'$ and $\mathtt{gn}(s) \cap \mathtt{dom}(\Gamma) = \emptyset$, then $\mathtt{ends}(s) \leq \mathtt{begins}(s) + e$ ([2]).

Theorem 3.4 (Safety) *If* $\Gamma \vdash_\Theta P : (\!|\,|\!)$, *then* P *is a safe process.*

Let us return to the example of the ATM. By assigning the types indicated in Figure 5 to the session names a and b, the good ATM Example from Figure 1 may be considered safe. However, as one might expect, with this type assignment Example 1.3 is not safe according to our type system. Note that the necessary assertion labels are inserted as already explained in that example. See [2] for further details.

$a : \sigma(\downarrow [idA : \mathbf{Int}](\!|\,|\!); \&\{\texttt{deposit} :\downarrow [amtA : \mathbf{Int}](\!|\,\langle idA, amtA\rangle\,|\!); \uparrow [balA : \mathbf{Int}](\!|\,|\!); \mathbf{1},$

$\quad \square\ \texttt{withdraw} :\downarrow [amtA : \mathbf{Int}](\!|\,|\!); \uparrow [balA : \mathbf{Int}](\!|\,|\!); \mathbf{1}\}(\!|\,|\!))$

$b : \sigma(\&\{\texttt{deposit} :\downarrow [idB : \mathbf{Int}](\!|\,|\!); \downarrow [amtB : \mathbf{Int}](\!|\,\langle idB, amtB\rangle\,|\!); \uparrow [balB : \mathbf{Int}](\!|\,|\!); \mathbf{1},$

$\quad \square\ \texttt{withdraw} :\downarrow [idB : \mathbf{Int}](\!|\,|\!); \downarrow [amtB : \mathbf{Int}](\!|\,|\!); \uparrow [balB : \mathbf{Int}](\!|\,|\!); \mathbf{1}\}(\!|\,|\!))$

Fig. 5. Types with effects for the ATM example

4 Conclusions

This paper combines correspondence assertions and session types. The latter are a versatile mechanism for restricting process behavior in multi-party interactions. A session describes the message exchange pattern between two parties. However, these types provide no means of synchronization between sessions in a multi-session system. Indeed, we have shown an example illustrating how, when processing a client's request for a withdrawal operation, an ATM may either decide not to interact with the Bank at all, or to deposit an amount less than the client requested and at the same time deposit the rest in some other account (creating an unintended message exchange with the Bank). Session types are not expressive enough to distinguish these variants: In both these cases, the same type can be assigned as in the case of the "correct" ATM. By introducing correspondence assertions into the type system, we are able to draw a fine line between them and identify the "correct" ATM from the faulty or malicious ones.

However, there are a number of situations that our system does not capture. For example, consider $P \,|\, Q$ where

$P = \texttt{begin}\ \langle 3\rangle; k![3]; \texttt{stop}$
$Q = k?(x); \texttt{end}\ \langle x\rangle\ \texttt{in stop}$

and assume that the type of k in P is $k :\uparrow [x : \mathbf{Int}]\langle x\rangle; \mathbf{1}$ and the type of k in Q is its dual, namely $k :\downarrow [x : \mathbf{Int}]\langle x\rangle; \mathbf{1}$. Then the fact that $P \,|\, Q$ is safe allows us to infer that if a value x was received in Q then it must be the case that P sent it. However, only under the additional assumption that the communication channel is not tampered with may we assume that the value received for x is in fact the value 3 sent by P. In many situations this is somewhat unrealistic. One possible approach to address this drawback is to incorporate encryption primitives as in [9].

Another situation not captured by our system is the following. Consider a process *Forwd* that receives a channel k from P and passes it on to some other process Q.

$$\begin{aligned} P(l) &= \texttt{request}\ PF(h)\ \texttt{in throw}\ h(l); \texttt{stop} \\ \mathit{Forwrd} &= \texttt{accept}\ PF(h_1)\ \texttt{in request}\ FQ(h_2) \\ &\qquad \texttt{in catch}\ h_1(k)\ \texttt{in throw}\ h_2(k); \texttt{stop} \\ Q &= \texttt{accept}\ FQ(h)\ \texttt{in catch}\ h(k'); \texttt{stop} \end{aligned}$$

The process Q may be interested in verifying that if it received some channel k', then this channel was exactly the one sent by P. If α is the type of l and "$\bullet$" is some dummy value, then one could attempt to insert effects in the type associated to h_1 and h_2

$$PF : \sigma(\downarrow [\alpha](\!| \langle \bullet \rangle |\!); \mathbf{1})$$

$$FQ : \sigma(\uparrow [\alpha](\!| \langle \bullet \rangle |\!); \mathbf{1}),$$

a corresponding `begin` $\langle \bullet \rangle$ assertion just before the `throw` operation in P, and a corresponding `end` $\langle \bullet \rangle$ assertion just after the catch operation in Q. The program $P|\, \mathit{Forwrd} \,|\, Q$ is indeed safe; however, the type assignment does not reflect our intentions. Indeed, if *Forwrd* did a `throw` with any channel of type α, in particular a channel different from k, then the resulting code would also be safe. What we really want is a type assignment of the form:

$$PF : \sigma(\downarrow [k : \alpha](\!| \langle k \rangle |\!); \mathbf{1})$$

$$FQ : \sigma(\uparrow [k : \alpha](\!| \langle k \rangle |\!); \mathbf{1})$$

However, such a type assignment is not allowed in our system since effects may not contain occurrences of channel names, namely k.

In addition to studying extensions of the calculus that remedy these situations, other issues require further attention:

- One issue that has not been addressed is some process for automated insertion of correspondence assertions and effects in types. Such a process would require as input a precise description of the property to be verified.
- When a deposit operation is requested by the client, correspondence assertions allow us to check that the account number which the ATM communicates to the Bank is exactly the same as the one punched in by the client as received by the ATM. It would be interesting to consider a language for describing constraints on multisets such as these. Then, instead of requiring safe programs to type with an empty effect, the satisfiability of an appropriate set of constraints would determine when this program is safe.
- Session types look much like processes. In [15] a generic type system for the π-calculus is studied in which types are CCS-like processes. They suggest that it is possible to integrate a theory of correspondence assertions into their framework. We are currently looking into this issue.
- Additional future work includes developing the formal theory of this calculus in HOL [11] and using the development to encode and reason about security and networking protocols.

Acknowledgments: We are grateful to the Laboratory for Secure Systems group at Stevens for interesting discussions, and in particular to Tom Chothia for suggesting session types as a relevant concept. We also thank Healfdene Goguen for comments and suggestions on previous drafts. This

work was partially supported by The Stevens Technogenesis Fund, the NSF Grant No. CCR-0220286 ITR:Secure Electronic Transactions, and the ARO Award No. DAAD-19-01-1-0473.

References

[1] Barendregt, H. P., *Lambda calculi with types*, in: S. Abramsky, D. M. Gabbay and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science: Background - Computational Structures (Volume 2)*, Clarendon Press, Oxford, 1992 pp. 117–309.

[2] Bonelli, E., A. Compagnoni and E. Gunter, *Correspondence assertions for process synchronization in concurrent communications*, Technical Report 2003-7, Department of Computer Science, Stevens Institute of Technology (2003).

[3] Cervesato, I. and F. Pfenning, *A linear logical framework*, Information and Computation **179** (2002), pp. 19–75.

[4] Chaki, S., S. Rajamani and J. Rehof, *Types as models: Model checking message-passing programs*, in: *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2002), pp. 45–57.

[5] Clarke, E. and W. Marrero, *Using formal methods for analyzing security*, Information Survivability Workshop (1998).

[6] Gay, S. and M. Hole, *Types and subtypes for client-server interactions*, in: *Proceedings of the European Symposium on Programming Languages and Systems*, number 1576 in LNCS (1999), pp. 74–90.

[7] Gay, S., V. Vasconcelos and A. Ravara, *Session types for inter-process communication*, Technical Report TR-2003-133, Department of Computing Science, University of Glasgow (2003).

[8] Girard, J.-Y., *Linear logic*, Theoretical Computer Science (1987), pp. 1–102.

[9] Gordon, A. and A. Jeffrey, *Authenticity by typing for security protocols*, in: *14th IEEE Computer Security Foundations Workshop* (2001), pp. 145–159.

[10] Gordon, A. and A. Jeffrey, *Typing correspondence assertions for communication protocols*, in: *Seventeenth Conference on the Mathematical Foundations of Programming Semantics (MFPS 2001)*, number 45 in ENTCS (2001).

[11] Gordon, M. and T. Melham, "Introduction to HOL: A theorem proving environment for higher-order logic," CUP, Cambridge, 1993.

[12] Hole, M. and S. Gay, *Bounded polymorphism in session types*, Technical Report TR-2003-132, Department of Computing Science, University of Glasgow (2003).

[13] Honda, K., M. Kubo and K. Takeuchi, *An interaction-based language and its typing system*, in: *Proc. of PARLE'94*, number 817 in LNCS (1994), pp. 398–413.

[14] Honda, K., V. Vasconcelos and M. Kubo, *Language primitives and type discipline for structured communication-based programming*, in: *Proc. of ESOP'98*, LNCS (1998), pp. 122–138.

[15] Igarashi, A. and N. Kobayashi, *A generic type system for the pi-calculus*, in: *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2001), pp. pp.128–141.

[16] Kobayashi, N., *A partially deadlock-free type process calculus*, in: *Proceedings of the Twelth Annual IEEE Symposium on Logic in Computer Science* (1997), pp. 128–139.

[17] Kobayashi, N., B. Pierce and D. Turner, *Linearity in the pi-calculus*, in: *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, 1996, pp. 358–371.

[18] Pierce, B. and D. Sangiorgi, *Typing and subtyping for mobile processes*, in: *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science* (1993), pp. 376–385.

[19] Puntigam, F., *Synchronization expressed in the types of communication channels*, in: *Proceedings of the EURO-PAR'96*, number 1123 in LNCS (1996), pp. 762–769.

[20] Turner, D., "The Polymorphic Pi-Calculus: Theory and Implementation," Ph.D. thesis, University of Edinburgh (1995).

[21] Vallecillo, A., V. Vasconcelos and A. Ravara, *Typing the behavior of objects and component using session types*, Electronic Notes in Theoretical Computer Science **68** (2003).

[22] Woo, T. and S. Lam, *A semantic model for authentication protocols*, in: *Proceedings of the IEEE Symposium on Security and Privacy*, 1993, pp. 178–194.

[23] Yoshida, N., *Graph types for monadic mobile processes*, in: *FST/TCS'16*, number 1180 in LNCS (1996), pp. 371–386.