JFP **15** (2): 219–247, 2005. © 2005 Cambridge University Press
DOI: 10.1017/S095679680400543X Printed in the United Kingdom

Correspondence assertions for process synchronization in concurrent communications

EDUARDO BONELLI

Stevens Institute of Technology, Castle Point on Hudson, Hoboken, NJ 07030, USA and LIFIA, Faculty of Informatics, University of La Plata, La Plata (CP 1900), Argentina
http://guinness.cs.stevens.edu/~ebonelli
(*e-mail:* ebonelli@cs.stevens.edu)

ADRIANA COMPAGNONI

Stevens Institute of Technology, Castle Point on Hudson, Hoboken, NJ 07030, USA
http://www.cs.stevens.edu/~abc
(*e-mail:* abc@cs.stevens.edu)

ELSA GUNTER

Department of Computer Science, University of Illinois at Urbana - Champaign, Thomas M. Siebel Center for Computer Science, 201 N. Goodwin, Urbana, IL 61801-2302
http://www.cs.uiuc.edu/~egunter
(*e-mail:* egunter@cs.uiuc.edu)

Abstract

High-level specification of patterns of communications such as protocols can be modeled elegantly by means of session types (Honda *et al.*, 1998). 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 (Clarke & Marrero, 1998; Gordon & Jeffrey, 2003b). 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. This new type system can be used to check:

- source of information,
- whether data is propagated as specified across multiple parties,
- if there are unspecified communications between parties, and
- if the data being exchanged has been modified by the code in an unspecified way.

We prove that evaluation preserves typability and that well-typed processes are safe. Also, we illustrate how the resulting theory allows us to address shortcomings present in the pure theory of session types.

Capsule Review

This paper unifies prior work on session types with that on correspondence assertions. Session types were introduced by Honda *et al.* to provide a static approximation of the possible interactions of a concurrent system. Correspondence assertions were introduced by Woo and Lam as a model of authenticity in a cryptographically secured communication protocol. In this paper, these two systems are unified to provide a model of authenticated interacting processes.

1 Introduction

Distributed and concurrent programming paradigms are increasingly popular, especially 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, the use of *session types* Honda *et al*., 1994, 1998) 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. This pair of dual types constitutes a *session type*. Session types are assigned to long-term channels and are shared among processes, where a long-term channel is a *port* whose communication protocol is prespecified. An example of a session type is:

$$(\downarrow \mathbf{Int}; \downarrow \mathbf{Int}; \uparrow \mathbf{Int}; \mathbf{1},\ \uparrow \mathbf{Int}; \uparrow \mathbf{Int}; \downarrow \mathbf{Int}; \mathbf{1})$$

The first component, namely $\downarrow \mathbf{Int}; \downarrow \mathbf{Int}; \uparrow \mathbf{Int}; \mathbf{1}$, 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 (one may think of an "adding" server that reads in two numbers and writes out their sum), and finally close. In order for the other party to interact correctly, it is assigned a *dual* type expression (the second component of the pair).

Note that session types specify the pattern of interaction between two parties. However, in a multi-party network consisting of three or more parties, no information is provided by these types as to how the interactions between different sessions are related. As a consequence, a process which implements a protocol specified by a system of session types may behave in a manner that cannot be tamed by session types alone. A detailed example, involving processes Client, ATM and Bank, is developed in section 1.1. It illustrates examples of situations which cannot be captured by session types such as:

- When Client requests a deposit operation from ATM, ATM may redirect some of the funds to a different account without violating the session-type based protocol description.
- ATM may forward an amount which does not coincide with the one it read in from Client.
- ATM may receive a deposit from Client and never contact Bank.

This paper addresses a strengthening of session types by incorporating a theory of *correspondence assertions* (cf. section 1.2). We shall discuss 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

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

$\mathsf{Client}(idC, amtC, a) = \texttt{request}\ a(k)\ \texttt{in}\ k![idC]; k \lhd \texttt{deposit}; k![amtC]; k?(balC)\ \texttt{in}\ \texttt{stop}$

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

$\mathsf{Bank}(b) = \texttt{accept}\ b(h)\ \texttt{in}$
$h \rhd \{\ \texttt{deposit}: h?(idB)\ \texttt{in}\ h?(amtB)\ \texttt{in}\ h![getNewBal(idB, amtB)];$
$\mathsf{Bank}[b] \Box \texttt{withdraw}: h?(idB)\ \texttt{in}\ h?(amtB)\ \texttt{in}$
$h![getOKAmt(idB)]; \mathsf{Bank}[b]\ \}$

Fig. 1. The ATM example.

number of "unsafe" programs which are well-typed in the theory of pure session types shall be rejected by our typing rules.

1.1 Motivation

Consider the following example, illustrated in Figure 1, consisting of three parties: Client, ATM, and Bank (Honda *et al.*, 1998) which we briefly describe below:

Client. On receiving a session request (through the shared name a), Client sends its id number (idC), selects a `deposit` operation, tells the amount of the deposit, and then waits for the new account balance.

ATM. The ATM first 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, 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 Bank. It then sends 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.

Bank. The bank listens on name b (shared with ATM) for requests for a session, and then waits for 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 $\mathsf{ATM}[a, b] | \mathsf{Client}[idC, amtC, a] | \mathsf{Bank}[b]$ denote the code for the concurrent execution of the indicated parties. The type system presented in (Honda *et al.*, 1998) 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 $\mathsf{ATM}[a, b] | \mathsf{Client}[idC, amtC, a] | \mathsf{Bank}[b]$.

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

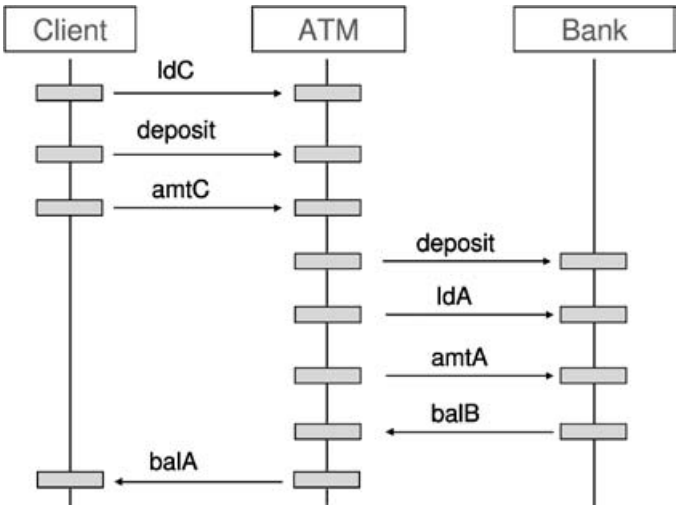

Fig. 2. Well-behaved execution sequence.

$$
\begin{aligned}
a: \quad & \sigma(\downarrow \mathbf{Int}; \&\{\texttt{deposit} :\downarrow \mathbf{Int}; \uparrow \mathbf{Int}; \mathbf{1}, \\
& \qquad \texttt{withdraw} :\downarrow \mathbf{Int}; \uparrow \mathbf{Int}; \mathbf{1}\}) \\
b: \quad & \sigma(\&\{\texttt{deposit} :\downarrow \mathbf{Int}; \downarrow \mathbf{Int}; \uparrow \mathbf{Int}; \mathbf{1}, \\
& \qquad \texttt{withdraw} :\downarrow \mathbf{Int}; \downarrow \mathbf{Int}; \uparrow \mathbf{Int}; \mathbf{1}\})
\end{aligned}
$$

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 to be selected at 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`. Figure 2 exhibits a sample execution sequence, written in message sequence chart-style notation (Z.120, 1996), complying with these types. 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 fact may be witnessed as follows. Consider 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◁deposit; h![idA]; h![amtA − 10];  (1)
h?(balA) in k![balA];
request b(h′) in  h′◁deposit; h′![diffId]; h′![10]; h′?(balA′)   (2)
in ATM[a, b]
```

This version of the `deposit` operation deposits into the client's account 10 units less than the amount told by Client **(1)**, and deposits the remaining 10 units in some account different from the client's by means of a new deposit request **(2)** to Bank which was not present in the original ATM.

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

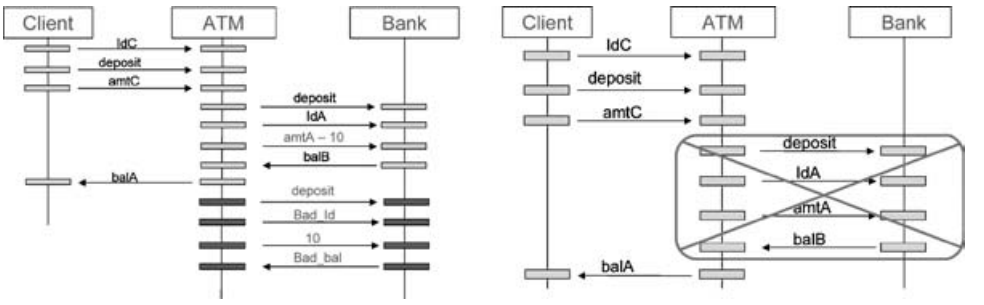

Fig. 3. Unwanted execution sequences.

One of the possible execution sequences of the resulting system is depicted to the left, in Figure 3. Unfortunately, this modified ATM is typable under the *same* type assumptions as the previous one.

Likewise, if the `deposit` operation of ATM were replaced by the same one except that the bank was not notified, then the resulting ATM would also type under the same type assumptions as the good one. In terms of execution sequences, the one shown to the right in Figure 3 is permitted under the same type assumptions. Here is the code that generated such a sequence.

Example 1.2 (Deposit II) The following variant of `deposit` allows ATM to keep the deposit of Client without depositing it in the account. If we call the resulting system ATM", then $\mathsf{ATM}"[a,b] \mid \mathsf{Client}[idC, amtC, a] \mid \mathsf{Bank}[b]$ is well-typed under exactly the same type assumptions as $\mathsf{ATM}[a,b] \mid \mathsf{Client}[idC, amtC, a] \mid \mathsf{Bank}[b]$.

$\texttt{deposit}\colon k?(amtA)\ \texttt{in}\ k![1000];\ \mathsf{ATM}[a,b]$

These examples suggest that, although session types elegantly encode communication patterns of message interchange, they cannot restrict interaction *between* sessions or enforce consistency of forwarded values (those received and then sent again). This paper introduces a type system based on *correspondence assertions* (Woo & Lam, 1993; Gordon & Jeffrey, 2003b) in which ATM may be distinguished from the variants depicted above.

1.2 Correspondence assertions

Correspondence assertions were introduced in (Woo & Lam, 1993) for reasoning about authentication protocols. In Gordon & Jeffrey (2001a), 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 Gordon & Jeffrey (2001b). 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 $P \mid Q$. Assertions are used to mark execution points in processes. As in Gordon & Jeffrey (2001b), 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

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

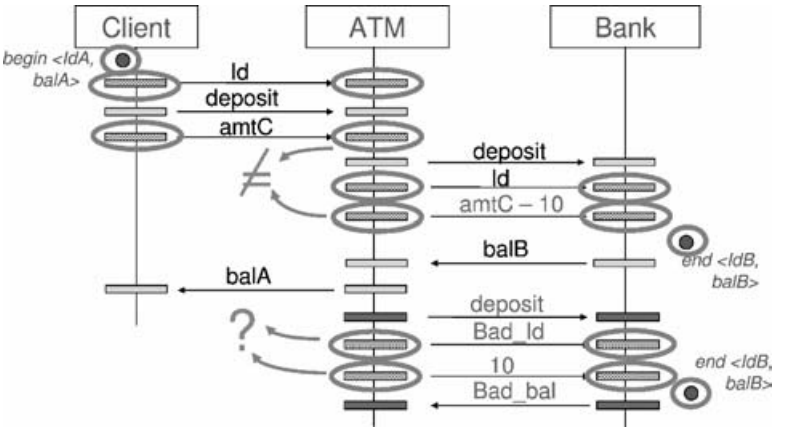

Fig. 4. Effects of Example 1.3.

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 Client and we 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':

$$\texttt{request}\ a(k)\ \texttt{in}\ \texttt{begin}\ \langle idC, amtC\rangle; k![idC]; k \triangleleft \texttt{deposit}; k![amtC]; k?(balC)\ \texttt{in}\ \texttt{stop}$$

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

$$\texttt{deposit}: h?(idB)\ \texttt{in}\ h?(amtB)\ \texttt{in}\ \texttt{end}\ \langle idB, amtB\rangle; h![getNewBal(idB, amtB)]; \mathsf{Bank}[b]$$

Finally, the session types of a and b are augmented with appropriate effects (see Figure 9 in section 3) such that, if ATM requests a deposit operation of the bank and sends some values for idB and $amtB$, then the incurred debt shall have to be relieved by a corresponding communication with the client; the client will have had to supply these values. This is depicted in Figure 4.

Now, the system $\mathsf{ATM}[a, b] \mid \mathsf{Client}'[idC, amtC, a] \mid \mathsf{Bank}'[b]$ shall be safe if every time the bank's `deposit` operation is executed for an id number idB and amount

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

$amtB$, the client requested the same operation on ATM, and $idB = idC$, the id entered by the client, and $amtB = amtC$, the amount entered by the client.

We may address Example 1.2 similarly by forcing ATM to engage in communication with the bank and, moreover, requiring that the `deposit` operation be selected. This is achieved by forcing interaction with the corresponding operation from the bank. In this case, the `begin` assertion is inserted in the Bank and the `end` assertion in the Client. Example 3.6 provides further details.

The type rules we present in section 2 allow us to 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. To "reach" the `end` assertion, Bank' must have previously executed the read operations of `deposit`, namely $h?(idB)$ and $h?(amtB)$. Now, h is a channel that is shared between Bank' and ATM'| Client' (via ATM'). As a consequence of 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 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. Indeed, as the obligation is passed back from Bank' to ATM', it becomes $\langle idA, amtA - 10 \rangle$, since these are the amounts sent for idB and $amtB$. As we pass the obligation back to Client, it is further transformed to $\langle idC, amtC - 10 \rangle$, which does not match with the assertion `begin` $\langle idC, amtC \rangle$. We may conclude, therefore, that the program is not safe. It is worth noting that if we changed the `begin` assertion to `begin` $\langle idC, amtC - 10 \rangle$, then the program would type-check and be declared safe. We would, in effect, be acknowledging that ATM' had a right to charge a 10 unit fee for a deposit transaction.

1.3 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 exchanged prior in the same session. We also include the branching/selection and delegation constructs from (Honda *et al.*, 1998) in our analysis. The resulting type system shall allow us to distinguish the three above-mentioned variants of 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. Our type system can be used:

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

— to check the source of information. In our example, we can check that the balance that Client receives always comes from Bank, and that the amount to be deposited received by Bank always comes from Client.
— to verify whether data is propagated as specified across multiple parties. In our example, ATM should behave as a forwarder that does not alter the data received from Bank or Client.
— to check if there are unspecified communications between parties. In our example, we can detect that ATM' is attempting a deposit not instructed by Client.
— and to check if the data being exchanged has been modified by the code in an unspecified way. In our example, ATM' tries to deposit a smaller amount than the one specified by Client.

- The combination of session types and correspondence assertions yields a dependently typed system that introduces a number of technical difficulties. For example, the usual representation of environments as sequences of assumptions (Barendregt, 1992; Gordon & Jeffrey, 2003b) fails to yield a calculus with basic properties such as admissibility of the Exchange structural rule (cf. Remark 2.6), a basic ingredient required in proofs of the so-called Subject Reduction property. Also, recording of effects in closed channels is crucial in order to benefit from properties such as Subject Congruence (cf. section 2.2.1 and 3).
- We show that evaluation preserves typability and that processes typable under empty effects are safe.

1.4 Related work

This work may be included among others in which type systems for the π-calculus are studied (Pierce & Sangiorgi, 1996; Kobayashi, 1998; Kobayashi *et al.*, 1999; Turner, 1995). Work on session types includes the study of: subtypes (Gay & Hole, 1999), bounded polymorphism (Hole & Gay, 2003), component-based software development (Vallecillo *et al.*, 2003) and formulations in a λ-calculus with input/output operations (Gay *et al.*, 2003). Although Yoshida (1996) and Puntigam (1996) do not explore session types they too aim at restricting process communication: the first studies a typing scheme for processes based on graph types and the second a type system for restricting communication in concurrent objects. The relation of these to session types is discussed in Honda *et al.* (1998).

While Gordon & Jeffrey (2001b) shares a fair amount in common with this work, there are a number of differences. Our language is targeted as a high-level specification language for protocols based on the notion of a session as a fundamental abstraction for structuring interactions. Such abstractions are not available in Gordon & Jeffrey (2001a, 2001b), where the authentication of low-level protocols such as those for key-exchange is dealt with. Indeed, there is no notion of session types or channel types. Correspondence assertions alone cannot capture deadlock due to inconsistency in the communication protocol: in Gordon & Jeffrey (2001b) channels are assigned a type independent of their use to send or receive

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

data, and so two processes trying to receive data on the same channel at the same time will deadlock. On the other hand, adding session types we may prevent such a situation since two communicating processes will be synchronized by a session type that specifies that while one process sends data the other process receives data.

Introducing linearity or dependent type systems does not suffice for encoding session types, since the key notion of dual types (cf. section 2.2.1) and compatibility (cf. Definition 2.3) of such types is not subsumed by these features. However, we could attempt to encode just channels into the type system of (Gordon & Jeffrey, 2003b; Gordon & Jeffrey, 2003a) in the following style

$$\begin{aligned}\mathscr{T}(k![v];P) &= (\nu k' : Ty)(k![v,k'];\mathscr{T}(P)\{k \leftarrow k'\})\\ \mathscr{T}(k?(a)\ \texttt{in}\ P) &= (\nu k' : Ty)(k?(a,k')\ \texttt{in}\ \mathscr{T}(P)\{k \leftarrow k'\})\end{aligned}$$

where the type expression Ty may be obtained from the channel type of k. However, it is not clear how to extend this translation to branching.

Recently, type systems where CCS-like processes are used for typing process expressions have appeared. The generic type system of Igarashi & Kobayashi (2001) is an example, although it does not incorporate correspondence assertions. Another approach is that of Chaki *et al.* (2002), 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.

1.5 Structure of the paper

Section 2 defines Iris, a system combining session types (Honda *et al.*, 1998) and correspondence assertions (Gordon & Jeffrey, 2003b). Section 2.2.1 presents a type system with effects for Iris. 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 Iris-calculus

2.1 Syntax

This section describes the syntax of Iris. 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* $\ldots, -1, 0, 1, \ldots$ ranged over by n, *branching labels* $l, l', \ldots$ and *process variables* written $X, Y, \ldots$ and also ATM, Bank, $\ldots$ 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:

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

P	::=	$\texttt{request}\ a(k)\ \texttt{in}\ P$	session request
		$\texttt{accept}\ a(k)\ \texttt{in}\ P$	session accept
		$k?(a)\ \texttt{in}\ P$	receive value
		$k![v];P$	send value
		$\texttt{catch}\ k(k')\ \texttt{in}\ P$	receive channel
		$\texttt{throw}\ k[k'];P$	send channel
		$(va : T)P$	expression name restriction
		$(vk : \bot_e)P$	channel restriction
		$k \lhd l;P$	label selection
		$k \rhd \{l_1 : P_1 \Box \ldots \Box\ l_n : P_n\}$	label branching
		$\texttt{stop}$	null process
		$P \mid Q$	parallel composition
		$\texttt{def}\ D\ \texttt{in}\ P$	process definition
		$X[\vec{v}]$	process variables
		$\texttt{begin}\ L;P$	begin assertion
		$\texttt{end}\ L;P$	end assertion
D	::=	$X_1(\vec{a}_1 : \vec{T}_1) = P_1\ \texttt{and} \ldots \texttt{and}\ X_n(\vec{a}_n : \vec{T}_n) = P_n$	process declarations

Remark 2.1 The notation $\vec{v}$ stands for $v_1, \ldots, v_n$, and likewise for $\vec{a}_i$ and $\vec{T}_i$ with $i \in 1..n$. Parentheses are binding constructs. Any two process expressions that 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 $k?(a)\ \texttt{in}\ P$ respectively, although, as in Honda *et al.* (1998), a translation to an asynchronous calculus with branching is possible. Controlled side-stepping of linearity constraints on channel usage is achieved by means of channel delegation $\texttt{throw}\ k[k'];P$ and $\texttt{catch}\ k(k')\ \texttt{in}\ Q$. We write $(va : T)P$ or $(vk : \bot_e)P$ for the usual constructs for name hiding; the former is for expression names and the latter for channel names. T denotes a type expression (Definition 2.1) and $\bot_e$ is the "closed" channel type with effect e (Definition 2.1). A mechanism for selection of a label and branching is available as $k \lhd l;P$ and $k \rhd \{l_1 : P_1 \Box \ldots \Box\ l_n : P_n\}$. We use `stop` for inaction, and the notation $P \mid Q$ has already been explained. Definitions of processes are also allowed through the $\texttt{def}\ D\ \texttt{in}\ P$ construct, possibly introducing recursion. They are used by the

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

application of a previously defined process variable to appropriate values, $X[\vec{v}]$. The `begin` and `end` assertions shall be used as type directives in the type system for Iris (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.

The set of free names of process expressions and assertion labels is defined in the usual manner. Recall that parentheses denote variable binding. A few cases of the definition are:

$$\begin{array}{lcl}
\mathtt{fn}(\mathtt{accept}\ a(k)\ \mathtt{in}\ P) = \mathtt{fn}(\mathtt{request}\ a(k)\ \mathtt{in}\ P) & \stackrel{def}{=} & \{a\} \cup (\mathtt{fn}(P) \setminus \{k\}) \\
\mathtt{fn}(k \rhd \{l_1 : P_1 \Box \ldots \Box l_n : P_n\}) & \stackrel{def}{=} & \{k\} \cup \bigcup_{i=1..n} \mathtt{fn}(P_i) \\
\mathtt{fn}((\nu k : \bot_e)P) & \stackrel{def}{=} & \mathtt{fn}(\bot_e) \cup (\mathtt{fn}(P) \setminus \{k\}) \\
\mathtt{fn}(X[\vec{v}]) & \stackrel{def}{=} & \bigcup_{i=1..n} \mathtt{fn}(v_i) \\
\mathtt{fn}(\mathtt{begin}\ L;P) = \mathtt{fn}(\mathtt{end}\ L;P) & \stackrel{def}{=} & \mathtt{fn}(L) \cup \mathtt{fn}(P) \\
\mathtt{fn}(X_1(\vec{a_1} : \vec{T_1}) = P_1\ \mathtt{and} \ldots \mathtt{and}\ X_n(\vec{a_n} : \vec{T_n}) = P_n) & \stackrel{def}{=} & \\
\multicolumn{3}{r}{\bigcup_{i=1..n} \mathtt{fn}(P_i) \setminus \{\vec{a_i}\} \cup \bigcup_{i=1..n} \mathtt{fn}((\vec{a_i} : \vec{T_i}))}
\end{array}$$

The set of free process variables of a process P and a process declaration D, written $\mathtt{fpv}(P)$ and $\mathtt{fpv}(D)$, respectively, may be defined in a similar manner.

2.2 *The type discipline*

The present section enriches the type system of Honda *et al.* (1998) 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 the pending obligations it has. 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 each `end` assertion corresponds to a matching `begin` assertion.

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

Definition 2.1 (Types with Effects) Assertion labels, effects and types are given by the following grammar:

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

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

A *type* is either a plain type or a channel type; we use U, U_i to range over types. 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 $\overline{\alpha}$:

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

The types α and $\overline{\alpha}$ shall be assigned to the two endpoints of a communication session. Note that $\overline{\bot_e}$ 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**. The type $\downarrow [a:T]e;\alpha$ is that of a channel that reads in a value v of type T and then behaves according to the type $\alpha\{a \leftarrow v\}$. The e is called a latent *effect*; an effect is a multi-set of assertion labels. The result of executing the input operation is that of removing $e\{a \leftarrow v\}$ from the current effects (cf. the typing rule Type Rcv). The type $\uparrow [a:T]e;\alpha$ is similar except that latent effects are added. The type $\downarrow [\alpha]e;\beta$ is that of a channel that reads in a channel of type α and then behaves according to the type β. The result of executing the input operation is removing e from the current multi-set of effects (cf. the typing rule Type Cat). Likewise for $\uparrow [\alpha]e;\beta$, except that the latent effect is added. The type $\&\{l_1:\alpha_1,\ldots,l_n:\alpha_n\}e$ is that of a process that expects to receive a selection of one of the operations labeled l_1 to l_n. Once l_i is selected, the resulting process behaves as described by the type expression α_i. Also, the latent effect e is removed from the current effects (cf. typing rule Type Brnch). The type $\oplus\{l_1:\alpha_1,\ldots,l_n:\alpha_n\}e$ is that of a process that makes a selection of one of the operations l_i. Once l_i is selected, the resulting process behaves as described by the type expression α_i. Also, the latent effect e is added to the current effects (cf. typing rule Type Sel). The type expression **1** simply terminates the sequence of types that conforms to a channel type, as already described above. The special type $\bot_e$ is that of a channel that is used at two dual channel types (if at all) as described by the Type Par typing rule and the notion of composition (Definition 2.4).

As already mentioned, we use $(\!| \ldots |\!)$ for the multi-set constructor. Multi-set subtraction, $e \setminus e'$, is the smallest multi-set e'' such that $e \leqslant e' + e''$, where "+" is multi-set union. The set of free names of a type U is defined as expected; the only interesting cases are:

$$\begin{array}{lll}
\mathtt{fn}(\updownarrow [a:T]e;\alpha) & \stackrel{def}{=} & \mathtt{fn}(T) \cup ((\mathtt{fn}(e) \cup \mathtt{fn}(\alpha)) \setminus \{a\}) \\
\mathtt{fn}(\updownarrow [\beta]e;\alpha) & \stackrel{def}{=} & \mathtt{fn}(\beta) \cup \mathtt{fn}(e) \cup \mathtt{fn}(\alpha) \\
\mathtt{fn}((\epsilon)) & \stackrel{def}{=} & \{\} \\
\mathtt{fn}((\vec{a}:\vec{T}, a':T')) & \stackrel{def}{=} & \mathtt{fn}((\vec{a}:\vec{T})) \cup (\mathtt{fn}(T') \setminus \{\vec{a}\}
\end{array}$$

where $\updownarrow [a:T]e$ denotes both $\downarrow [a:T]e$ and $\uparrow [a:T]e$, and likewise for $\updownarrow [\beta]e$. Regarding the definition of substitution, some representative cases are:

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

$$\begin{array}{lcl}\sigma(\alpha)\{a \leftarrow v\} & \stackrel{def}{=} & \sigma(\alpha\{a \leftarrow v\}) \\ (\updownarrow [b : T]e; \alpha)\{a \leftarrow v\} & \stackrel{def}{=} & \updownarrow [b : T\{a \leftarrow v\}]e\{a \leftarrow v\}; \alpha\{a \leftarrow v\} \\ \bot_e\{a \leftarrow v\} & \stackrel{def}{=} & \bot_{e\{a \leftarrow v\}}\end{array}$$

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 $\{x_1, \ldots, x_n\}$, and the range of Γ, written $\mathtt{ran}(\Gamma)$, is the set $\{U_1, \ldots, U_n\}$. 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. The free names of Γ, written $\mathtt{fn}(\Gamma)$, is the set of names occurring either in the domain of Γ, or free in a type in the range of Γ, *i.e.* $\mathtt{fn}(\Gamma) = \mathtt{dom}(\Gamma) \cup \bigcup_{U \in \mathtt{ran}(\Gamma)} \mathtt{fn}(U)$. 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 2.2 (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 *well-formed* if it satisfies the following three conditions:

C1. For each $x \in \mathtt{domPl}(\Gamma)$, x is an expression name, and for each $y \in \mathtt{domCh}(\Gamma)$, y is a channel name.
C2. For each $i \in 1..n$, $\mathtt{fn}(U_i) \subseteq \mathtt{dom}(\Gamma) \setminus \{x_i\}$.
C3. The relation $\hookrightarrow$ is irreflexive, that is, $x_i : U_i \not\hookrightarrow x_i : U_i$ for all $x_i : U_i \in \Gamma$.

The first condition, **C1** requires that only channel types be assigned to channel names, and only plain types be assigned to expression names. Condition **C2** requires that all free names in types assigned by Γ must be declared within Γ. Also, it states that in an assumption $x : U$, x may not occur free in U. The second condition, **C3**, 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 some structural rules fail (Remark 2.6).

Remark 2.2 Note that since channel names may not appear in assertion labels, types may only depend on names which are assigned plain types. For example, environments such as $k :\downarrow [a : \mathbf{Int}](\!|\,|\!); \mathbf{1} \cdot k' :\downarrow [b : T](\!|\,\langle k \rangle\,|\!); \mathbf{1}$ are not permitted since channel names (in this case k) may not be present in effects. Since interaction

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

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

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

Fig. 5. Well-formed values and process parameters.

through channel names is restricted by linearity conditions in the sense of linear logic (Girard, 1987) (see explanation of Type Par rule below), this restriction states that we do not allow types depending on linear assumptions (in contrast to shared 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 outweighs its benefits. In fact this restriction already appears in other settings in which linear and intuitionistic (or shared) assumptions live together such as the linear logical framework of Cervesato & Pfenning (2002).

Iris's type system defines the following four *judgments*:

$\Gamma \vdash_\Theta \diamond$	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

We shall often use $\mathscr{J}$ for the fragments of judgments $\diamond$, $v : T$, $(\vec{v}) : (\vec{a} : \vec{T})$, or $P : e$. 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 judgment $\Gamma \vdash_\Theta \diamond$ 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 rules for well-formed environments and process protocols, well-typed values, and well-typed process parameters are found in Figure 5.

The type rules of Iris are presented in Figure 6. 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 judgment 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. In the Type Snd rule, the latent effect associated to the ouput type of k becomes a credit. In other words, it becomes a "payment" obligation that must be met by some prior begin assertion or some prior receive operation. Similar comments apply to Type Rcv. Note, however, that this time the latent effect of the type of the parameter of the input (i.e. "b")

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

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

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

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

$$\frac{\Gamma \vdash_\Theta v : T \quad \Gamma \cdot k : \alpha\{a \leftarrow v\} \vdash_\Theta P : e \quad \texttt{fn}(e') \subseteq \texttt{dom}(\Gamma) \cup \{a\}}{\Gamma \cdot k : \uparrow [a : T]e'; \alpha \vdash_\Theta k![v]; P : e + e'\{a \leftarrow v\}}\ \textsf{Type Snd}$$

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

$$\frac{\Gamma \cdot k : \alpha_1 \vdash_\Theta P_1 : e_1 \quad \ldots \quad \Gamma \cdot k : \alpha_n \vdash_\Theta P_n : e_n \quad \texttt{fn}(e) \subseteq \texttt{dom}(\Gamma)}{\Gamma \cdot k : \&\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\}e \vdash_\Theta k \rhd \{l_1 : P_1, \ldots, l_n : P_n\} : (\bigvee e_i) \setminus e}\ \textsf{Type Brnch}$$

$$\frac{\Gamma \cdot k : \alpha_j \vdash_\Theta P : e \quad l_j \in \{l_1, \ldots, l_n\} \quad \texttt{fn}(\oplus\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\}e') \subseteq \texttt{dom}(\Gamma)}{\Gamma \cdot k : \oplus\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\}e' \vdash_\Theta k \lhd l_j; P : e + e'}\ \textsf{Type Sel}$$

$$\frac{\Gamma \cdot k : \alpha \vdash_\Theta P : e \quad \beta \neq \mathbf{1} \quad \texttt{fn}(\beta, e') \subseteq \texttt{dom}(\Gamma) \quad k' \notin \texttt{dom}(\Gamma) \cup \{k\}}{\Gamma \cdot k' : \beta \cdot k : \uparrow [\beta]e'; \alpha \vdash_\Theta \texttt{throw}\ k[k']; P : e + e'}\ \textsf{Type Thr}$$

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

$$\frac{\Gamma \vdash_\Theta \diamond \quad \texttt{ranCh}(\Gamma) \subseteq \{\mathbf{1}, \bot_e\}}{\Gamma \vdash_\Theta \texttt{stop} : (\!|\,|\!)}\ \textsf{Type Stop} \qquad \frac{\Gamma \cdot a : T \vdash_\Theta P : e \quad a \notin \texttt{fn}(\Gamma, e)}{\Gamma \vdash_\Theta (va : T)P : e}\ \textsf{Type NRes}$$

$$\frac{\Gamma \cdot k : \bot_{e'} \vdash_\Theta P : e}{\Gamma \vdash_\Theta (vk : \bot_{e'})P : e}\ \textsf{Type CRes} \qquad \frac{\Gamma \vdash_\Theta P : e \quad \Gamma' \vdash_\Theta Q : e' \quad \Gamma \asymp \Gamma'}{\Gamma \circ \Gamma' \vdash_\Theta P \,|\, Q : e + e'}\ \textsf{Type Par}$$

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

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

$$\frac{\Gamma \setminus \texttt{chan}(\Gamma) \cdot \vec{a}_i : \vec{T}_i \vdash_\Theta P_i : (\!|\,|\!) \quad \Theta(X_i) = (\vec{a}_i : \vec{T}_i) \quad \Gamma \vdash_\Theta Q : e}{\Gamma \vdash_{\Theta \setminus \vec{X}} \texttt{def}\ X_1(\vec{a}_1 : \vec{T}_1) = P_1 \ldots \texttt{and} \ldots X_n(\vec{a}_n : \vec{T}_n) = P_n\ \texttt{in}\ Q : e}\ \textsf{Type Def}$$

Fig. 6. Well-formed process expressions.

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

becomes a debit or payment. 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 in the sense of taking the least upper bound. 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.[1] 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 rules Type NRes and Type CRes introduce a new private expression name and a new private channel name, respectively.

The Type Par rule types the parallel execution of two processes. A channel may be used 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 to guarantee such linear usage the environments Γ and Γ' are required to be *compatible*.

Definition 2.3 (Compatibility $\asymp$) The relation $\asymp$ is defined as follows: $\emptyset \asymp \emptyset$, and $\Gamma \asymp \Gamma'$ implies

1. $\Gamma \cdot a : T \asymp \Gamma' \cdot a : T$
2. $\Gamma \cdot k : \alpha \asymp \Gamma' \cdot k : \overline{\alpha}$
3. $\Gamma \cdot k : \alpha \asymp \Gamma'$, if $k \notin \mathtt{dom}(\Gamma')$
4. $\Gamma \asymp \Gamma' \cdot k : \alpha$, if $k \notin \mathtt{dom}(\Gamma)$

Note that the notion of compatibility makes sense for two sets of assumptions which 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 2.4 (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

1. $(\Gamma \cdot a : T) \circ (\Gamma' \cdot a : T) = (\Gamma \circ \Gamma') \cdot a : T$
2. $(\Gamma \cdot k : \alpha) \circ (\Gamma' \cdot k : \overline{\alpha}) = (\Gamma \circ \Gamma') \cdot k : \bot_{\mathtt{fnEff}(\alpha)}$
3. $(\Gamma \cdot k : \alpha) \circ (\Gamma') = (\Gamma \circ \Gamma') \cdot k : \alpha$, if $k \notin \mathtt{dom}(\Gamma')$
4. $\Gamma \circ (\Gamma' \cdot k : \alpha) = (\Gamma \circ \Gamma') \cdot k : \alpha$, if $k \notin \mathtt{dom}(\Gamma)$

The effect $\mathtt{fnEff}(\alpha)$ is the multi-set which includes an assertion label $\langle a \rangle$ for each occurrence of a free expression name x in α. Other variants for the second clause of Definition 2.4 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). Some basic properties of compatibility and composition are:

[1] Technically, this allows us to correct a problem present in Honda *et al.* (1998), namely the failure of Subject Congruence.

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

Lemma 2.3 (Basic Properties of $\asymp$ and $\circ$)

1. If $\Gamma_1 \vdash_\Theta \diamond$ and $\Gamma_2 \vdash_\Theta \diamond$, then $\Gamma_1 \circ \Gamma_2 \vdash_\Theta \diamond$.
2. ($\circ$ is partially commutative) If $\Gamma_1 \circ \Gamma_2$ is defined, then so is $\Gamma_2 \circ \Gamma_1$ and moreover $\Gamma_1 \circ \Gamma_2 = \Gamma_2 \circ \Gamma_1$.
3. ($\circ$ is partially associative) If $\Gamma_1 \asymp \Gamma_2$ and $\Gamma_2 \asymp \Gamma_3$ and $\Gamma_1 \asymp \Gamma_2 \circ \Gamma_3$, then $(\Gamma_1 \circ \Gamma_2) \circ \Gamma_3 \asymp \Gamma_1 \circ (\Gamma_2 \circ \Gamma_3)$.
4. $\Gamma_1 \circ \Gamma_2 \asymp \Gamma_3$ and $\Gamma_1 \asymp \Gamma_2$ implies $\Gamma_2 \asymp \Gamma_3$ and $\Gamma_1 \asymp \Gamma_2 \circ \Gamma_3$.

Proof

1. By close inspection of the corresponding definitions.
2. If $\Gamma \asymp \Gamma'$, then $\Gamma' \asymp \Gamma$ and a close inspection of Definition 2.4 yields the desired result.
3. As for associativity, we proceed by induction on the number of type assumptions in Γ_1.
4. By induction on the number of type assumptions in Γ_1.

□

The Type Subsum rule allows increasing the required assertion obligations of a process term. Although such a rule is natural and informative when deriving judgments it does not allow more terms to be typed:

Lemma 2.4 (Subsumption Elimination) If $\Gamma \vdash_\Theta P : e$, then for some $e' \leqslant e$, $\Gamma \vdash_\Theta P : e'$ is derivable without using the rule Type Subsum.

Proof

By induction on the derivation of $\Gamma \vdash_\Theta P : e$ using the properties of $\leqslant$ on multi-sets. Let us consider as an example the case where the derivation of $\Gamma \vdash_\Theta P : e$ ends in an application of Type Bgn:

$$\frac{\Gamma \vdash_\Theta P' : e_1 \quad \mathtt{fn}(L) \subseteq \mathtt{dom}(\Gamma)}{\Gamma \vdash_\Theta \mathtt{begin}\ L; P' : e_1 \setminus (\!| L |\!)}\ \textsf{Type Bgn}$$

and $e = e_1 \setminus (\!| L |\!)$. From the induction hypothesis we know that $\Gamma \vdash_\Theta P' : e_1'$ is derivable without the use of Type Subsum and with $e_1' \leqslant e_1$. We may then apply Type Bgn and obtain:

$$\frac{\Gamma \vdash_\Theta P' : e_1' \quad \mathtt{fn}(L) \subseteq \mathtt{dom}(\Gamma)}{\Gamma \vdash_\Theta \mathtt{begin}\ L; P' : e_1' \setminus (\!| L |\!)}\ \textsf{Type Bgn}$$

Let e' be $e_1' \setminus (\!| L |\!)$. Then $\Gamma \vdash_\Theta P : e'$ is derivable without using Type Subsum and $e' \leqslant e$.

for some effect e'', then $e'' \setminus (\!| L |\!) = e''$. □

The remaining rules, Type PVar and Type Def, type process variables and process definitions, respectively. In the former case, note that all channel types must have been consumed before calling the process X. In the latter case, each P_i with $i \in 1..n$ must be typed without making use of the channels in the environment Γ. This is a means for preserving the linearity constraints on channel names.

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

As one might expect, the following result (derivability implies wf of environments) may be proved by induction on the size of the derivation of $\Gamma \vdash_\Theta \mathscr{J}$.

Lemma 2.5 If $\Gamma \vdash_\Theta \mathscr{J}$, then $\Gamma \vdash_\Theta \diamond$.

Remark 2.6 A representation of environments based on sequences of hypothesis, as usually adopted in the literature on dependent type systems (Barendregt, 1992), 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 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 $k :\uparrow [a : T]e'; \alpha$ are in condition of being exchanged 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 judgment 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.

Let $\Gamma \vdash_\Theta \diamond$ and $\Delta \vdash_\Theta \diamond$. A *renaming* for Γ in Δ is a parallel substitution γ from names to names that respects sorts (expression names are mapped to expression names and channel names are mapped to channel names) such that

1. for every $x : U$ in Γ, $\gamma(x) : U$ is in Δ, and
2. for every $x : U \in \Delta$ which is not of the form $\gamma(y) : U$ for some $y : U$ in Γ, if U is a channel type then $U = \bot_e$ or $U = \mathbf{1}$.

Proposition 2.7 (Renaming) If $\Gamma \vdash_\Theta \mathscr{J}$ and γ is a renaming for Γ in Δ, then $\Delta \vdash_\Theta \gamma(\mathscr{J})$.

The proof is by induction on the derivation of $\Gamma \vdash_\Theta \mathscr{J}$.

As a consequence of the Renaming Proposition, hypothesis in environments may be exchanged without affecting derivability.

Lemma 2.8 (Exchange) If $\Gamma \cdot x : U \cdot x' : U' \cdot \Gamma' \vdash_\Theta \mathscr{J}$, then $\Gamma \cdot x' : U' \cdot x : U \cdot \Gamma' \vdash_\Theta \mathscr{J}$.

Furthermore, the following Weakening Lemma also follows from the Renaming Proposition. It holds without restrictions for plain types. However, in the case of channel types we must require the new type introduced into the environment to be $\mathbf{1}$ or $\bot_e$. This is necessary due to the Type Stop and Type PVar typing rules.

Lemma 2.9 (Weakening) If $\Gamma \vdash_\Theta \mathscr{J}$ and $x \notin \mathtt{dom}(\Gamma)$ and $\Gamma \cdot x : U \vdash_\Theta \diamond$, then

1. $\Gamma \cdot x : U \vdash_\Theta \mathscr{J}$, if U is a plain type.

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

2. $\Gamma \cdot x : U \vdash_\Theta \mathscr{J}$, if U is a channel type and $U = \mathbf{1}$ or $U = \bot_e$.

Lemma 2.10 (Strengthening) If $\Gamma \cdot x : U \vdash_\Theta \mathscr{J}$ and $x \notin \mathtt{fn}(\Gamma, \Theta, \mathscr{J})$, then $\Gamma \vdash_\Theta \mathscr{J}$.

Recall from Section 2.2.1 that the relation $x_i : U_i$ *depends directly on* $x_j : U_j$ *in* Γ (written $(x_j : U_j) \hookrightarrow_d (x_i : U_i)$) holds 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 $\hookrightarrow_d$; likewise we say some subset of hypothesis $\Gamma' \subset \Gamma$ depends on $x : U \in \Gamma$ in Γ if for each $x' : U' \in \Gamma'$, $x' : U'$ depends on $x : U$.

Lemma 2.11 (Substitution Lemma) Suppose $\Gamma \cdot \Delta \cdot a : T \vdash_\Theta \mathscr{J}$, where Γ does not depend on a and Δ does. Suppose, furthermore, that $\Sigma \vdash_\Theta v : T$ with $\Gamma \asymp \Sigma$. Then $\Gamma \cdot \Delta\{a \leftarrow v\} \vdash_\Theta \mathscr{J}\{a \leftarrow v\}$.

3 Safety proof for Iris

To trace the execution of certain actions such as `begin` and `end` assertions we shall introduce a labeled transition semantics (Gordon & Jeffrey, 2003a) (LTS) for Iris. 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 standard definition of structural congruence applies, as depicted in Figure 7.

The *actions*, denoted with letters $\psi, \phi, \ldots$, of the transition system are explained informally below:

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

Thus the set of actions is $\mathtt{begin}\ L, \mathtt{end}\ L, \mathtt{res}(a : T), \mathtt{res}(k : \bot_e), \tau$. The labeled transition system for Iris is given in Figure 8; we write $P \xrightarrow{\psi} P'$ when P reduces to P' through action ψ. The set of *free* and *generated names* of an action are given by:

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

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 (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:

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

$P \equiv P$ SC Refl
$P \equiv Q \Rightarrow Q \equiv P$ SC Symm
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$ SC Trans

$P \,|\, \texttt{stop} \equiv P$ SC Stop
$P \,|\, Q \equiv Q \,|\, P$ SC Par Comm
$(P \,|\, Q) \,|\, R \equiv P \,|\, (Q \,|\, R)$ SC Par Asoc

$P \equiv P' \Rightarrow (\nu x : U)P \equiv (\nu x : U)P'$ SC New Name/Chan
$P \equiv P' \Rightarrow P \,|\, Q \equiv P' \,|\, Q$ SC Par
$P \equiv P' \Rightarrow k?(a)\ \texttt{in}\ P \equiv k?(a)\ \texttt{in}\ P'$ SC Rcv
$P \equiv P' \Rightarrow k![v]; P \equiv k![v]; P'$ SC Send
$P \equiv P' \Rightarrow \texttt{accept}\ a(k)\ \texttt{in}\ P \equiv \texttt{accept}\ a(k)\ \texttt{in}\ P'$ SC Acpt
$P \equiv P' \Rightarrow \texttt{request}\ a(k)\ \texttt{in}\ P \equiv \texttt{request}\ a(k)\ \texttt{in}\ P'$ SC Requ
$P \equiv P' \Rightarrow k \lhd l; P \equiv k \lhd l; P'$ SC Sel
$P \equiv P' \Rightarrow k \rhd \{\ldots \Box l_i : P \Box \ldots\} \equiv k \rhd \{\ldots \Box l_i : P' \Box \ldots\}$ SC Brnch
$P \equiv P' \Rightarrow \texttt{throw}\ k[k']; P \equiv \texttt{throw}\ k[k']; P'$ SC Thr
$P \equiv P' \Rightarrow \texttt{catch}\ k(k')\ \texttt{in}\ P \equiv \texttt{catch}\ k(k')\ \texttt{in}\ P'$ SC Cat
$P \equiv P' \Rightarrow \texttt{def}\ D\ \texttt{in}\ P \equiv \texttt{def}\ D\ \texttt{in}\ P'$ SC Def
$P \equiv P' \Rightarrow \texttt{begin}\ L; P \equiv \texttt{begin}\ L; P'$ SC Begin
$P \equiv P' \Rightarrow \texttt{end}\ L; P \equiv \texttt{end}\ L; P'$ SC End

$(\nu x_1 : U_1)(\nu x_2 : U_2)P \equiv (\nu x_2 : U_2)(\nu x_1 : U_1)P$, if $x_1 \neq x_2, x_1 \notin \texttt{fn}(U_2), x_2 \notin \texttt{fn}(U_1)$ SC Res Res
$(\nu x : U)(P \,|\, Q) \equiv (\nu x : U)P \,|\, Q$, if $x \notin \texttt{fn}(Q)$ SC Res Par
$(\nu x : U)\texttt{def}\ D\ \texttt{in}\ P \equiv \texttt{def}\ D\ \texttt{in}\ (\nu x : U)P$, if $x \notin \texttt{fn}(D)$ SC Res Def

$(\texttt{def}\ X_1(\vec{a_1} : \vec{T_1}) = P_1 \ldots \texttt{and} \ldots X_n(\vec{a_n} : \vec{T_n}) = P_n\ \texttt{in}\ P) \,|\, Q$
$\equiv \texttt{def}\ X_1(\vec{a_1} : \vec{T_1}) = P_1 \ldots \texttt{and} \ldots X_n(\vec{a_n} : \vec{T_n}) = P_n\ \texttt{in}\ (P \,|\, Q)$, if $\{X_1, \ldots, X_n\} \cap \texttt{fpv}(Q) = \emptyset$ SC Def Par
$\texttt{def}\ X_1(\vec{a_1} : \vec{T_1}) = P_1 \ldots \texttt{and} \ldots X_n(\vec{a_n} : \vec{T_n}) = P_n$
$\texttt{in def}\ Y_1(\vec{b_1} : \vec{S_1}) = Q_1 \ldots \texttt{and} \ldots Y_m(\vec{b_m} : \vec{S_m}) = Q_m\ \texttt{in}\ P$
$\equiv \texttt{def}\ X_1(\vec{a_1} : \vec{T_1}) = P_1 \ldots \texttt{and} \ldots X_n(\vec{a_n} : \vec{T_n}) = P_n$
$\texttt{and}\ Y_1(\vec{b_1} : \vec{S_1}) = Q_1 \ldots \texttt{and} \ldots Y_m(\vec{b_m} : \vec{S_m}) = Q_m\ \texttt{in}\ P$, if $(\{X_1, \ldots, X_n\} \cup \bigcup_{i=1}^{n} \texttt{fpv}(P_i)) \cap \{Y_1, \ldots, Y_m\} = \emptyset$ SC Def And

Fig. 7. Structural congruence.

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

$P \equiv P' \Rightarrow P \xrightarrow{\epsilon} P'$ Trace $\equiv$
$P \xrightarrow{\psi} Q, Q \xrightarrow{s} P' \Rightarrow P \xrightarrow{\psi s} P'$ Trace Action (where $\texttt{fn}(\psi) \cap \texttt{gn}(s) = \emptyset$)

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 $\texttt{begins}(\psi_1 \ldots \psi_n) \stackrel{def}{=} \texttt{begins}(\psi_1) \cup \ldots \cup \texttt{begins}(\psi_n)$ and the latter $\texttt{ends}(\psi_1 \ldots \psi_n) \stackrel{def}{=} \texttt{ends}(\psi_1) \cup \ldots \cup \texttt{ends}(\psi_n)$, where

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 8. LTS for Iris.

$\cup$ stands for multi-set union and

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

Definition 3.2 (Safe Process) A process P is *safe* if and only if for all traces s and processes P', if $P \xrightarrow{s} P'$ then $\texttt{ends}(s) \leqslant \texttt{begins}(s)$.

Thus a process is safe if every `end` L is accounted for by a corresponding `begin` L. For example, `begin` L;`stop` is safe however, `begin` L;`end` L;`end` L;`stop` is not. We now address the proof of safety, namely that a process typable with null effect is safe. This requires showing that process reduction preserves typings and effects. Since reduction is defined in terms of the structural congruence relation, we must first verify that typing is invariant with respect to this relation. More precisely,

Lemma 3.1 (Subject Congruence) Assume $\Gamma \vdash_\Theta P : e$. If $P \equiv Q$, then $\Gamma \vdash_\Theta Q : e$.

Subject Congruence is proved by induction on the derivation of $P \equiv Q$; the fact that effects are not lost when environments are composed (clause 2 in Definition 2.4) is crucial to its proof.

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

Remark 3.2 As a consequence of the fact that $\overline{\mathbf{1}}=\mathbf{1}$ the following substitutability result is required for Subject Congruence:

Lemma 3.3 If $\Gamma \cdot k : \mathbf{1} \vdash_\Theta \mathscr{J}$, then $\Gamma \cdot k : \bot_{e'} \vdash_\Theta \mathscr{J}$ for any effect e' such that $\mathtt{fn}(e') \subseteq \mathtt{dom}(\Gamma)$.

Proof
By induction on the derivation of $\Gamma \cdot k : \mathbf{1} \vdash_\Theta \mathscr{J}$. Some cases which are worth commenting on are:

Type Thr. The condition $\beta \neq \mathbf{1}$ in the formulation of Type Thr is required for this case to go through.

Type Par. If $k : \mathbf{1} \in \Gamma_1 \circ \Gamma_2$, then either $k : \mathbf{1} \in \Gamma_1 \setminus \Gamma_2$ or $k : \mathbf{1} \in \Gamma_2 \setminus \Gamma_1$. We thus apply the induction hypothesis to the appropriate upper judgment. Note that the resulting judgment shall still be compatible with the other upper judgment of Type Par. □

Note that the converse of Lemma 3.3 does not hold. Also, it fails in Honda ém et al. (1998), and seems to be the culprit for failure of Subject Congruence of the calculus proposed in Honda ém et al. (1998). a dual terminator, say $\underline{\mathbf{1}}$ and then define $\overline{\mathbf{1}}=\underline{\mathbf{1}}$ and also $\underline{\mathbf{1}}=\mathbf{1}$ (adapting the definition of compatiblity and composition, and the type rules accordingly). However, Lemma 3.3 still fails to the non-linearity in the type rule Type Thr. This suggests that a detailed logical analysis of Iris could be of interest as a further research topic.

Theorem 3.4 (Subject Reduction) Assume $\Gamma \vdash_\Theta P : e$.

1. If $P \xrightarrow{\tau} P'$, then there exists Γ' such that $\Gamma' \vdash_\Theta P' : e$ and Γ' and Γ differ only in the effects assigned to the channel type $\bot$ (if any).
2. If $P \xrightarrow{\mathtt{begin}\, L} P'$, then $\Gamma \vdash_\Theta P' : e + (\!| L |\!)$.
3. If $P \xrightarrow{\mathtt{end}\, L} P'$, then $\Gamma \vdash_\Theta P' : e \setminus (\!| L |\!)$ and $L \in e$.
4. If $P \xrightarrow{\mathtt{res}(a : T)} P'$ and $a \notin \mathtt{dom}(\Gamma)$, then $\Gamma \cdot a : T \vdash_\Theta P' : e$.
5. If $P \xrightarrow{\mathtt{res}(k : \bot_f)} P'$ and $k \notin \mathtt{dom}(\Gamma)$, then $\Gamma \cdot k : \bot_f \vdash_\Theta P' : e$.

then $\vec{X} \notin \mathtt{dom}(\Theta)$ and $\Gamma \vdash_{\Theta \cdot \vec{X} : (\vec{x}_i : \vec{T}_i)} P' : e$.

Proof
Subject reduction is proved by cases in a standard manner according to the action which takes place. A representative case is provided below. It relies on the following $\equiv$-Elimination observation which may be verified by induction on the derivation of $P \xrightarrow{\upsilon} P'$: If $P \xrightarrow{\upsilon} P'$, then for some $Q \equiv P$ and $Q' \equiv P'$, $Q \xrightarrow{\upsilon} Q'$ is derivable without using the rule Trans $\equiv$.

Suppose $P \xrightarrow{\tau} P'$ derives from the Trans Comm transition, then by $\equiv$-Elimination:

$$\begin{array}{rcl} P & \equiv & \mathtt{def}\ D\ \mathtt{in}\ (k![v]; P_1) \,|\, (k?(b)\ \mathtt{in}\ P_2) \,|\, R \\ P' & \equiv & \mathtt{def}\ D\ \mathtt{in}\ P_1 \,|\, P_2\{b \leftarrow v\} \,|\, R \end{array}$$

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

where $D = X_1(\vec{a}_1 : \vec{T}_1) = O_1 \ldots \texttt{and} \ldots X_n(\vec{a}_n : \vec{T}_n) = O_n$. By Subsumption Elimination and Subject Congruence the derivation of $\Gamma \vdash_\Theta P : e$ is

$$\frac{\Gamma \setminus \texttt{chan}(\Gamma) \cdot \vec{a}_i : \vec{T}_i \vdash_\Theta O_i : (\!|) \quad \Theta(X_i) = (\vec{a}_i : \vec{T}_i) \quad \Gamma \vdash_\Theta Q : e}{\Gamma \vdash_{\Theta \setminus \vec{X}} \texttt{def } X_1(\vec{a}_1 : \vec{T}_1) = O_1 \ldots \texttt{and} \ldots X_n(\vec{a}_n : \vec{T}_n) = O_n \texttt{ in } Q : e} \text{ Type Def}$$

The derivation of $\Gamma \vdash_\Theta Q : e$ takes the following form.

- On the one hand we have:
$$\frac{\Gamma \vdash_\Theta v : T \quad \Gamma \cdot k : \alpha\{a \leftarrow v\} \vdash_\Theta P_1 : e_{P_1} \quad \texttt{fn}(e') \subseteq \texttt{dom}(\Gamma) \cup \{a\}}{\Gamma \cdot k :\uparrow [a : T]e'; \alpha \vdash_\Theta k![v]; P_1 : e_{P_1} + e'\{a \leftarrow v\}} \text{ Type Snd}$$
- On the other we have:
$$\frac{\Delta \cdot b : T \cdot k : \overline{\alpha}\{a \leftarrow b\} \vdash_\Theta P_2 : e_{P_2} \quad b \notin \texttt{fn}(e_{P_2} \setminus e'\{a \leftarrow b\}) \cup \texttt{fn}(\alpha, \Delta) \quad \texttt{fn}(e') \subseteq \texttt{dom}(\Delta) \cup \{a\}}{\Delta \cdot k :\downarrow [a : T]e'; \overline{\alpha} \vdash_\Theta k?(b) \texttt{ in } P_2 : e_{P_2} \setminus e'\{a \leftarrow b\}} \text{ Type Rcv}$$
- Finally, the derivation ends in an application of Type Par to the above two derivations yielding the top left-hand judgment in:
$$\frac{(\Gamma \circ \Delta) \cdot k : \bot_f \vdash_\Theta (k![v]; P_1) | (k?(b) \texttt{ in } P_2) : e'' \quad \Sigma \vdash_\Theta R : e_R}{((\Gamma \circ \Delta) \cdot k : \bot_f) \circ \Sigma) \vdash_\Theta (k![v]; P_1) | (k?(b) \texttt{ in } P_2) | R : e'' + e_R} \text{ Type Par}$$
where
$$f = \texttt{fnEff}(\uparrow [a : T]e'; \alpha)$$
$$e'' = (e_{P_1} + e'\{a \leftarrow v\}) + (e_{P_2} \setminus e'\{a \leftarrow b\})$$
$$(e_{P_1} + e'\{a \leftarrow v\}) + (e_{P_2} \setminus e'\{a \leftarrow b\}) + e_R \leqslant e$$
$$\Gamma \asymp \Delta \qquad (\Gamma \circ \Delta) \cdot k : \bot_f \asymp \Sigma$$

From $\Delta \cdot b : T \cdot k : \overline{\alpha}\{a \leftarrow b\} \vdash_\Theta P_2 : e_{P_2}$ and $\Gamma \vdash_\Theta v : T$ and $\Delta \asymp \Gamma$ and the Substitution Lemma, we deduce

$$\Delta \cdot k : \overline{\alpha}\{a \leftarrow v\} \vdash_\Theta P_2\{b \leftarrow v\} : e_{P_2}\{b \leftarrow v\}$$

Next we construct the derivation:

$$\frac{\Gamma \cdot k : \alpha\{a \leftarrow v\} \vdash_\Theta P_1 : e_{P_1} \quad \Delta \cdot k : \overline{\alpha}\{a \leftarrow v\} \vdash_\Theta P_2\{b \leftarrow v\} : e_{P_2}\{b \leftarrow v\}}{(\Gamma \circ \Delta) \cdot k : \bot_{f'} \vdash_\Theta P_1 | P_2\{b \leftarrow v\} : e_{P_1} + e_{P_2}\{b \leftarrow v\}} \text{ Type Par}$$

where $f' = \texttt{fnEff}(\alpha\{a \leftarrow v\})$.

Finally, we introduce another application of Type Par:

$$\frac{(\Gamma \circ \Delta) \cdot k : \bot_{f'} \vdash_\Theta P_1 | P_2\{b \leftarrow v\} : e_{P_1} + e_{P_2}\{b \leftarrow v\} \quad \Sigma \vdash_\Theta R : e_R}{((\Gamma \circ \Delta) \cdot k : \bot_f) \circ \Sigma \vdash_\Theta P_1 | P_2\{b \leftarrow v\} | R : e_{P_1} + e_{P_2}\{b \leftarrow v\} + e_R} \text{ Type Par}$$

We are left to verify that $e_{P_1} + e_{P_2}\{b \leftarrow v\} + e_R \leqslant e$. We reason as follows:

$$\begin{array}{lll} & e_{P_1} + e_{P_2}\{b \leftarrow v\} + e_R & \\ = & e_{P_1} + (e_{P_2} \setminus e'\{a \leftarrow b\} + e'\{a \leftarrow b\})\{b \leftarrow v\} + e_R & \\ = & e_{P_1} + (e_{P_2} \setminus e'\{a \leftarrow b\})\{b \leftarrow v\} + e'\{a \leftarrow b\}\{b \leftarrow v\} + e_R & \\ = & e_{P_1} + e_{P_2} \setminus e'\{a \leftarrow b\} + e'\{a \leftarrow v\} + e_R & (b \notin \texttt{fn}(e_{P_2} \setminus e'\{a \leftarrow b\})) \\ \leqslant & e & \text{(hypothesis)} \quad \square \end{array}$$

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

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

Fig. 9. Types with effects for the ATM example.

$\mathsf{Client}(id, amt, a) = \texttt{request}\ a(k)\ \texttt{in}\ k![id];\ k \lhd \texttt{deposit};\ k![amt];$
$k?(bal)\ \texttt{in end}\ \langle id, amt, bal\rangle;\ \texttt{stop}$

$\mathsf{ATM}(a, b) = \texttt{accept}\ a(k)\ \texttt{in}\ k?(idA)\ \texttt{in}$
$k \rhd \{\ \texttt{deposit}: k?(amtA)\ \texttt{in}\ k![1000];\ \mathsf{ATM}[a, b]$
$\square\ \texttt{withdraw}: \texttt{request}\ b(h)\ \texttt{in}\ k?(amtA)\ \texttt{in}$
$h \lhd \texttt{withdraw};\ h![idA];\ h![amtA];\ h?(OKedAmtA)\ \texttt{in}$
$k![OKedAmtA];\ \mathsf{ATM}[a, b]\ \}$

$\mathsf{Bank}(b) = \texttt{accept}\ b(h)\ \texttt{in}$
$h \rhd \{\ \texttt{deposit}: h?(idB)\ \texttt{in}\ h?(amtB)\ \texttt{in}\ updateData;$
$\texttt{begin}\ \langle idB, amtB, balB\rangle;\ h![balB];\ \mathsf{Bank}[b]$
$\square\ \texttt{withdraw}: h?(idB)\ \texttt{in}\ h?(amtB)\ \texttt{in}$
$getOK\_AmtForIdB;\ h![OKedAmtB];\ \mathsf{Bank}[b]\ \}$

Fig. 10. The ATM example.

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 $P \xrightarrow{s} P'$ and $\mathtt{gn}(s) \cap \mathtt{dom}(\Gamma) = \emptyset$, then $\mathtt{ends}(s) \leqslant \mathtt{begins}(s) + e$.

Theorem 3.5 (Safety) If $\Gamma \vdash_\Theta P : (\!|\,|\!)$, then P is a safe process.

Let us return to the example of the ATM. By assigning the session names a and b the types indicated in Figure 9, the good ATM (when executed concurrently with Client and Bank) may be seen to be safe. Moreover, with this type assignment Example 1.3 is not safe according to our type system, as one might expect. Note that the necessary assertion labels are inserted as already explained in that example.

Example 3.6 (Deposit II (continued)) Consider the code of Figure 10. It consists of the ATM example of Figure 1 augmented with a `begin` assertion in the bank and an `end` assertion in the code of the client; the code of the `deposit` operation in ATM has been altered as suggested by Example 1.2 (note that it does not consult with the bank).

Also, let us take the following type expressions for the session names a and b, where two latent effects have been introduced to the session types introduced earlier.

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

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

As the reader may like to verify, both the client and the bank of Figure 10 are typable with the empty effect $(\!|\,|\!)$. Indeed, in the client, the credit $\langle id, amt, bal\rangle$

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

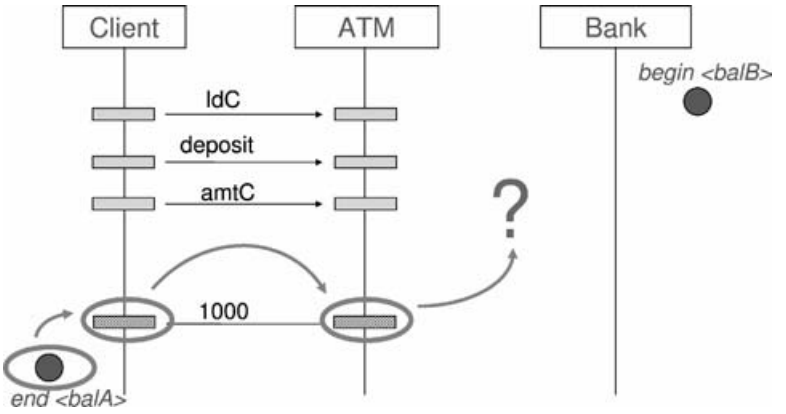

Fig. 11. Effects of Example 3.6.

introduced by the end assertion is paid for by the latent effect associated with the input operation $k?(bal)$; in the bank the credit $\langle idB, amtB, balB\rangle$ introduced by the latent effect associated to the output operation $h![balB]$ is paid for by the begin assertion just preceding it.

However, the credit $\langle idA, amtA, 1000\rangle$ introduced by the latent effect associated with the output operation $k![1000]$ in the code of ATM is not paid for. Thus the system resulting from executing all three components concurrently is unsafe since the net effect of the resulting system is the sum of the effects of each of its components (see rule Type Par in Figure 6). An illustration is given in Figure 11.

Let us consider an additional example including the use of catch and throw and the typings they entail.

Example 3.7 (Authenticator Example) Consider a server Server that serves up secrets in accordance with a given security level, and an authenticator Auth, trusted by Server to supply security clearances. Upon receiving an identifier from a client, Auth determines the security level of the client and passes the security level off to the server followed by the channel shared by Auth and the client. Upon receiving the channel, Server uses it to answer the client's question at the right level. The client's concern is that he gets the answer from the server that corresponds to the question. More formally, let

$$
\begin{aligned}
\mathsf{Server}(s) \;=\;& \texttt{accept}\ s(h)\ \texttt{in}\ h?(id)\ \texttt{in}\ h?(sec\_level)\ \texttt{in}\\
& \texttt{catch}\ h(k)\ \texttt{in}\ k?(question)\ \texttt{in}\\
& \texttt{begin}\ \langle question, getSecret(id, question, sec\_level)\rangle;\\
& k![getSecret(id, question, sec\_level)];\ \mathsf{Server}[s]\\
\mathsf{Auth}(a,s) \;=\;& \texttt{accept}\ a(k)\ \texttt{in}\ \mathsf{Auth}[a,s] \mid k?(id)\ \texttt{in}\ \texttt{request}\ s(h)\ \texttt{in}\\
& h![id]; h![getSecLevel(id)]; \texttt{throw}\ h[k]; \texttt{stop}\\
\mathsf{Client}(id, question, a) \;=\;& \texttt{request}\ a(k)\ \texttt{in}\ k![id];\ k![question];\ k?(secret)\ \texttt{in}\\
& \texttt{end}\ \langle question, secret\rangle;\ \texttt{stop}
\end{aligned}
$$

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

The system $\mathsf{Client}[id, question, a] \mid \mathsf{Auth}[a, s] \mid \mathsf{Server}[s]$ is typable with empty effects under the type assignments:

$$a : \sigma(\downarrow [id : \mathbf{Int}](\!|\,|\!); \downarrow [question : \mathbf{Int}](\!|\,|\!); \uparrow [secret : \mathbf{Int}](\!|\,\langle question, secret\rangle\,|\!); \mathbf{1})$$
$$s : \sigma(\downarrow [id : \mathbf{Int}](\!|\,|\!); \downarrow [sec\_level : \mathbf{Int}](\!|\,|\!); \downarrow [\downarrow [question : \mathbf{Int}](\!|\,|\!); \uparrow [secret : \mathbf{Int}](\!|\,\langle question, secret\rangle\,|\!)](\!|\,|\!); \mathbf{1})$$

If we run $\mathsf{Client}[id, question, a] \mid \mathsf{Auth}[a, s] \mid \mathsf{Server}[s]$, then the client will be served the secret answer to its question at the appropriate security level. Since the code typechecks with empty effect, the `end` in the code for Client is matched by the `begin` in the code for Server. Thus, we know that the answer received by the client from the server is the secret corresponding to the question asked. Since the client and Server communicate the question and answer on the same channel, it may not seem like much is being guaranteed. However, it is worth remembering that the client does not know that he is talking directly to the server, particularly since he started out talking to the authenticator, not the server.

The server in this setting it likely to be concerned that the person asking the question is the one that has the security clearance for the answer given. To answer this question we will augment the original code with additional assertions:

$$\begin{aligned}
\mathsf{Server}'(s) \;=\;& \texttt{accept}\ s(h)\ \texttt{in}\ h?(id)\ \texttt{in}\ h?(sec_level)\ \texttt{in}\\
& \texttt{catch}\ h(k)\ \texttt{in}\ k?(question)\ \texttt{in end}\ \langle id, question\rangle;\\
& \texttt{begin}\ \langle question, secret(question, sec_level)\rangle;\\
& k![secret(id, question, sec_level)];\ \mathsf{Server}\\
\mathsf{Client}'(id, question, a) \;=\;& \texttt{request}\ a(k)\ \texttt{in}\\
& \texttt{begin}\ \langle id, question\rangle k![id];\ k![question];\ k?(secret)\ \texttt{in}\\
& \texttt{end}\ \langle question, secret\rangle;\ \texttt{stop}
\end{aligned}$$

The new system $\mathsf{Client}'(id, question, a) \mid \mathsf{Auth}(a, s) \mid \mathsf{Server}'(s)$ is typable with empty effects under the modified type assignments:

$$a : \sigma(\downarrow [id : \mathbf{Int}](\!|\,|\!); \downarrow [question : \mathbf{Int}](\!|\,\langle id, question\rangle\,|\!); \uparrow [secret : \mathbf{Int}](\!|\,\langle question, secret\rangle\,|\!); \mathbf{1})$$
$$s : \sigma(\downarrow [id : \mathbf{Int}](\!|\,|\!); \downarrow [sec\_level : \mathbf{Int}](\!|\,|\!); \downarrow [\downarrow [question : \mathbf{Int}](\!|\,\langle id, question\rangle\,|\!); \uparrow [secret : \mathbf{Int}](\!|\,\langle question, secret\rangle\,|\!)](\!|\,|\!); \mathbf{1})$$

With the same process definitions, now let us consider the process

$$\mathsf{Client}'(id1, q1, a) \mid \mathsf{Client}'(id2, q2, a) \mid \mathsf{Client}'(id3, q3, a) \mid \mathsf{Client}'(id4, q4, a) \mid \mathsf{Auth}(a, s) \mid \mathsf{Server}'(s) \mid \mathsf{Server}'(s) \mid \mathsf{Server}'(s).$$

This process also type-checks with empty effects under the revised type assignments given above for a and b. However, this time, we don't know which client will get paired with which server, or which one will have to wait until one of the other clients finishes. If we assume that different copies of the server are capable of giving different answers, then different runs of this program can yield the clients receiving different answers to their questions. Still, the type systems guarantees that the client

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

receives an answer to its question, and that the server is issuing the answer to a client with appropriate authorization.

4 Conclusions

This paper defines a typed π-calculus that combines correspondence assertions and session types. Session types 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, ATM may decide either not to interact with Bank at all, or deposit an amount smaller than the one the client requested, and at the same time deposit the difference in some other account (creating an unintended message exchange with Bank). Session types are not expressive enough to distinguish these variants: In both 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 distinguish the "correct" ATM from the faulty or malicious ones.

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

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

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 $k :\downarrow [x : \mathbf{Int}]\langle x\rangle; \mathbf{1}$. The fact that $P \mid Q$ 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 Gordon and Jeffrey (2001a).

Another situation not captured by our system is exemplified by a process *Forwrd* 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. One could attempt to insert effects in the type associated to h_1 and h_2

$$\begin{aligned} PF &: \sigma(\downarrow [\alpha](\!|\langle\bullet\rangle|\!); \mathbf{1}) \\ FQ &: \sigma(\uparrow [\alpha](\!|\langle\bullet\rangle|\!); \mathbf{1}) \end{aligned}$$

where α is the type of l and "$\bullet$" is some dummy value, insert a corresponding begin $\langle\bullet\rangle$ assertion just before the throw operation in P and a corresponding

https://doi.org/10.1017/S095679680400543X Published online by Cambridge University Press

`end` $\langle\bullet\rangle$ assertion just after the catch operation in Q. The program $P[l]|\,Forwrd\,|\,Q$ is indeed safe, but the type assignment does not reflect our intentions. Indeed, if *Forwrd* executed a `throw` instruction 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 this example).

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

- When a deposit operation is requested by the client, correspondence assertions allow us to check that the account number that ATM communicates to Bank is exactly the same as the one punched in by the client as received by ATM. It would be interesting to consider studying a language of constraints in which such conditions may be formalized. In such a language, multi-sets and their operations become part of the object-language and a system of equations for solving constraints based on these expressions is required.
- Session types look much like processes. In (Igarashi & Kobayashi, 2004) 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 (Gordon & Melham, 1993) and using the development to encode and reason about security and networking protocols.

Acknowledgments

An extended abstract of this work was presented at the FOCLASA 2003 workshop.

This work was supported in part by the NSF Grant No. CCR-0220286 ITR:Secure Electronic Transactions and by the ARO under Award No. DAAD-19-01-1-0473.

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 would also like to thank the anonymous reviewers for their many useful suggestions and Healfdene Goguen for comments on previous drafts. This work was partially supported by The Stevens Technogenesis Fund, the NSF Grant No. CCR-0220286 ITR: Secure Electronic Transactions, the ARO Award No. DAAD-19-01-1-0473, and the NJCS&T Software Engineering for Distributed Computing and Networking grant.

References

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

Cervesato, I. and Pfenning, F. (2002) A linear logical framework. *Infor. & Computation*, **179**(1), 19–75.

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

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

Gay, S. and Hole, M. (1999) Types and subtypes for client-server interactions. *Proc. European Symposium on Programming Languages and Systems: Lecture Notes in Computer Science 1576*, pp. 74–90. Springer-Verlag.

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

Girard, J.-Y. (1987) Linear Logic. *Theor. Comput. Sci.* 1–102.

Gordon, A. and Jeffrey, A. (2003a) Authenticity by typing for security protocols. *J. Comput. Security*, **11**(4), 451–521.

Gordon, A. and Jeffrey, A. (2003b) Typing correspondence assertions for communication protocols. *Theor. Comput. Sci.* **300**, 379–409.

Gordon, M. J.C. and Melham, T. F. (1993) *Introduction to HOL: A theorem proving environment for higher-order logic*. Cambridge University Press.

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

Honda, K., Kubo, M. and Takeuchi, K. (1994) An interaction-based language and its typing system. *Proc. PARLE'94: Lecture Notes in Computer Science 817*, pp. 398–413. Springer-Verlag.

Honda, K., Vasconcelos, V. and Kubo, M. (1998) Language primitives and type discipline for structured communication-based programming. *Proc. ESOP'98: Lecture Notes in Computer Science*, pp. 122–138. Springer-Verlag.

Igarashi, A. and Kobayashi, N. (2004) A generic type system for the pi-calculus. *Theor. Comput. Sci.* **311**(1–3), 121–163.

Kobayashi, N. (1998) A partially deadlock-free type process calculus. *ACM Trans. Program. Lang. & Syst. (TOPLAS)*, **20**(2).

Kobayashi, N., Pierce, B. C. and Turner, D. N. (1999) Linearity and the pi-calculus. *ACM Trans. Program. Lang. & Syst.* **21**(5), 914–947.

Pierce, B. and Sangiorgi, D. (1996) Typing and subtyping for mobile processes. *Mathematical Structures in Comput. Science*, **6**(5).

Puntigam, F. (1996) Synchronization expressed in the types of communication channels. *Proc. EURO-PAR'96: Lecture Notes in Computer Science 1123*, pp. 762–769. Springer-Verlag.

Turner, D. (1995) *The polymorphic pi-calculus: Theory and implementation*. PhD thesis, University of Edinburgh.

Vallecillo, A., Vasconcelos, V. and Ravara, A. (2003) Typing the behavior of objects and component using session types. *Electr. Notes in Theor. Comput. Sci.* **68**(3).

Woo, T. and Lam, S. (1993) A semantic model for authentication protocols. *Proc. IEEE Symposium on Security and Privacy*, pp. 178–194.

Yoshida, N. (1996) Graph types for monadic mobile processes. *FST/TCS'16: Lecture Notes in Computer Science 1180*, pp. 371–386. Springer-Verlag.

Z.120, ITU-T Recommendation. (1996) *Message sequence chart (MSC)*.