Typechecking Safe Process Synchronization

Eduardo Bonelli, [a,b]

[a] *Stevens Institute of Technology*
[b] *LIFIA*

Adriana Compagnoni, [c]

[c] *Stevens Institute of Technology*

Elsa L. Gunter [d]

[d] *New Jersey Institute of Technology*

Abstract

Session types describe the interactions between two parties within multi-party communications. They constitute a communication protocol in the sense that the order and type of interactions between two parties are specified. For their part, correspondence assertions provide a mechanism for synchronization. When session types and correspondence assertions are combined, they are able to describe synchronization across different communication sessions, yielding a rich language for imposing expressive interaction patterns in multi-party communications.

This paper studies the typechecking problem for Iris, a typed π-calculus that combines session types and correspondence assertions. We define a typechecking algorithm and prove that it is sound and complete with respect to the typing rules. Furthermore, we show that the typing system satisfies the minimum effects property. Although session types have been extensively studied in the past few years, to our knowledge this is the first proof of decidability of typechecking for a type system with session types.

Keywords: Concurrency, π-calculus, communications, type systems, typechecking.

Contents

1	Introduction	3
1.1	Cryptography and Language-based Security	5
1.2	Related Work	6
1.3	Structure of the Paper	7
2	The Iris-Calculus	8
2.1	Syntax	8
2.2	The Type Discipline	9
3	An Example in Iris	16
4	Typechecking	20
5	Conclusions and Future Work	34

⋆ 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.

Email addresses: `ebonelli@cs.stevens-tech.edu` (Eduardo Bonelli,), `abc@cs.stevens-tech.edu` (Adriana Compagnoni,), `elsa@cis.njit.edu` (Elsa L. Gunter).

URLs: `http://guinness.cs.stevens-tech.edu/~ebonelli` (Eduardo Bonelli,), `http://www.cs.stevens-tech.edu/~abc` (Adriana Compagnoni,), `http://www.cs.njit.edu/~elsa` (Elsa L. Gunter).

[1] Faculty of Informatics, University of La Plata, La Plata, Argentina.

1 Introduction

Increasingly in our society we are coming to depend upon processors for monitoring and controlling devices in almost all aspects of our lives. In many instances the behavior of these processors is governed by communication with other processors, which may be other components of the same system or may be remotely located. Examples where such communication is critical can be found in space exploration, air traffic control, medical devices, banking, and electronic commerce. For each of these examples, errors in the software for these communications could lead to substantial financial loss, and, in some cases, the loss of human life. Therefore, it is of great importance to have high assurance that the software governing these communications and resulting decisions is correct.

Our approach applies to any situation where there is communication between many parties that can be factored into one-to-one communications. For example, a rover on Mars sending data to an unmanned spacecraft that communicates with a base on Earth can be factored into two communication sessions, one between the rover and the spacecraft and another one between the spacecraft and the base. Even if exchanges in both communications have to be interleaved, our framework allows such factorization.

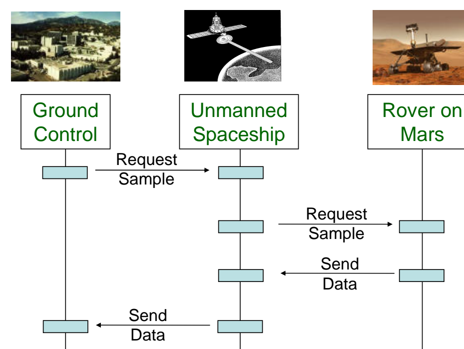

Fig. 1. Communication between Earth and Rover via Spacecraft

Session types allow us to describe the exchange of information between two parties. They describe the information being exchanged, in which order it is exchanged, what party sends it and what party receives it and what type the information has. Furthermore, by combining session types with correspondence assertions we can trace the origin of information. For example, we can automatically and statically verify whether for every possible execution of the system the data received by the base always comes from the rover.

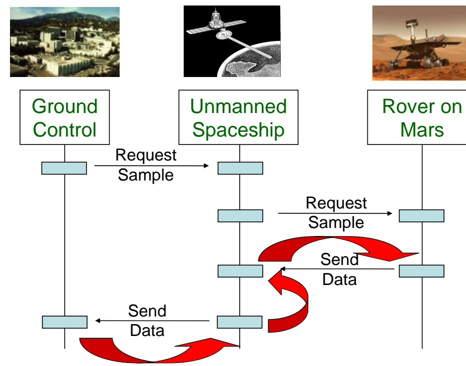

Fig. 2. Communication between Earth and Rover via Spacecraft

An autonomous surface rover that acts upon information received from a sensor is another example where verifying the origin of data is important. The data used by the rover should be fresh and not old data that could mistakenly be used to perform a move that could compromise the success of the mission. Our dependent session types system allows us to ensure that the data used by the rover is always new. We can automatically check that the sensor is always consulted before a move.

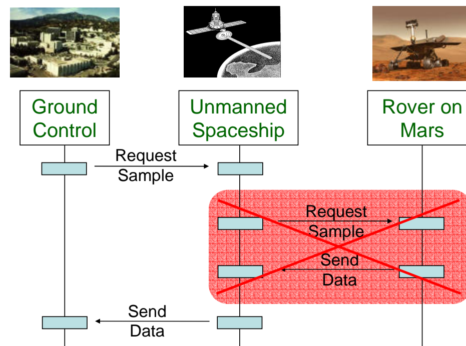

Fig. 3. Faulty Communication between Earth and Rover via Spacecraft

Programmer errors can include calling the wrong subroutine, causing an unmanned spacecraft not to obtain data from a rover. Even if the communication protocol between the rover and the spacecraft has been formally verified, not contacting the rover does not violate the communication protocol, but poses an undetected problem that can seriously compromise a mission. With dependent session types, we can verify that the data used by the spacecraft always comes from the sensor.

Closer to Earth, web services are becoming an increasingly important enabling technology for business software, and are based on the remote procedure call,

one of the key elements of our language-based security approach. As such, our approach is well-suited to provide a more secure framework for web services and the applications built with them.

Session types have attracted considerable attention in the past decade, motivated by the benefits that such type systems provide for the analysis of protocols. The initial proposal for session types was by Honda et al (Honda *et al.*, 1994). Natural extensions of this work that have been studied include subtypes (Gay & Hole, 1999) and bounded polymorphism (Hole & Gay, 2003). They have also been studied in the context of component-based software development (Vallecillo *et al.*, 2003) and reformulated in the λ-calculus with input/output operations (Gay *et al.*, 2003).

Iris, a statically typed language based on the π-calculus, extending (Honda *et al.*, 1998) with correspondence assertions (Gordon & Jeffrey, 2001a, 2003b), was first introduced in (Bonelli *et al.*, 2003a). There it was shown that the type system allows us to detect irregularities in concurrent communications such as the unauthorized modification of data, missing or avoided communications, and extra unintended communications. In this paper we continue the study of Iris by showding that typechecking is decidable.

In the earlier paper we developed an example of a Client, an ATM, and a Bank, and showed how the system succeeds in detecting a dishonest ATM. Imagine the Client selecting a deposit operation in the menu of the ATM machine. The ATM then contacts the Bank and performs the deposit on behalf of the Client, by selecting a deposit operation on the Bank's menu, and the Bank sends a balance back that the ATM forwards to the Client. In this case, the type system can detect if the ATM contacts the Bank, if the deposit order is transmitted faithfully by the ATM to the Bank (exact amount and correct bank account), and if the ATM fails to contact the Bank or if it contacts the Bank to performed an unauthorized deposit in the wrong account. Furthermore, it checks if the final balance that the Client receives comes from the Bank.

1.1 Cryptography and Language-based Security

Cryptography and language-based security can be considered complementary security approaches, where one deals with the static and exhaustive analysis of code, considering every possible execution path, while the other deals with the integrity of information in transit and authentication. Notice that our approach is not limited to financial operations or other situations in which secrecy must be maintained, but is applicable in any situation in which parallel processes synchronize to exchange information.

Origin of Information

Our analysis allows us to trace information back to its originating point. In contrast, cryptography can be used to authenticate who signed a message (using digital signatures), but it cannot tell us where the information originated, only who signed it, or just whose signature was used to sign it. We have to trust that the signed data is correct. Furthermore, cryptography adds the runtime overhead of encryption and decryption.

Combining our analysis with cryptography, we could guarantee that the correct data was received and that it was not corrupted in transit either by unauthorized modifications built in the code or by run-time alterations, such as feeding the wrong data into the communication channel.

The wrong data could have been accidentally signed and sent out. Our type system can identify where the data originated, without having to trust whoever signed the data for its integrity.

Detection of Unintended and Missing Interactions

Our system can detect if there is an extra interaction between two parties that was not supposed to be there, and it can also detect if an agent fails to contact another agent in a systematic way. In other words, we can detect if the code has a flaw that prevents the flow of information between agents.

This is a case where the accepted techniques for verification of communication protocols fall short, since a communication that does not take place, in the case of the ATM not contacting the Bank, does not violate any communication protocol. Furthermore, if the ATM performs an extra deposit in the wrong account, the communication protocol between the ATM and the Bank is still satisfied, yet the operation is undesirable. Cryptography cannot address this issue of missing or extra communications either: it only deals with the integrity of the information being exchanged and as an authentication mechanism between the communicating parties.

1.2 Related Work

Processes and types. This work may be included among others in which type systems for the π-calculus are studied (Pierce & Sangiorgi, 1993; Kobayashi, 1997; Kobayashi *et al.*, 1996; Turner, 1995). Subtyping is introduced in the setting of session types in (Gay & Hole, 1999); however, the concept of synchronization between sessions is not explored. The work (Yoshida, 1996) and (Puntigam, 1996) does not explore session types either: the first studies a typ-

ing scheme for processes based on graph types and the second a type system for restricting communication in concurrent objects; their relation to session types is discussed in (Honda *et al.*, 1998). While (Gordon & Jeffrey, 2001b) shares a fair amount in common with this work, there is a major difference. In (Gordon & Jeffrey, 2001b) dependencies in types are "horizontal" in the sense that in a type expression such as $\downarrow [x : T_1, y : T_2]$ the type of y may depend on the value of x, this being fixed for *all* communications over a channel of this type. However, since our setting is that of session types we allow "vertical" dependencies of the form $\downarrow [x : T_1]; \triangleright \{l_1 : \downarrow [y : T_2], l_2 : \downarrow [z : T_3]\}$ where y may depend on x and z does not. In the present work, dependency spans whole sessions. Recently, type systems where CCS-like processes are used for typing process expressions have appeared. The generic type system of (Igarashi & Kobayashi, 2001, 2004) is an example, although it does not incorporate correspondence assertions.Another approach is (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.

Typechecking and type inference. The first type system for the π-calculus is based on the notion of sort (Milner, 1999). In (Gay, 1993) a sort inference algorithm for the polyadic π-calculus is given. B. Pierce and D. Turner define a type checking and type inference algorithm for PICT, a concurrent programming language based on a polymorphic version of the π-calculus. The Cryptic Project, a joint project between A. Gordon and A. Jeffrey, includes an implementation of a type-checker for the language they developed in (Gordon & Jeffrey, 2001b,a, 2003b,a) that includes correspondence assertions and public and private data. Regarding the processes as models paradigm introduced by Chaki et al. in (Chaki *et al.*, 2002) there is also an implementation of a type-checker (called Piper) for their language. Typechecking for the common part of the generic type system in (Igarashi & Kobayashi, 2001, 2004) is discussed in that paper.

1.3 Structure of the Paper

Section 3 describes a detailed example of an electronic auction system in Iris. Section 2 defines Iris combining session types (Honda *et al.*, 1998) and correspondence assertions (Gordon & Jeffrey, 2001b). Section 2.2.1 presents a type system with effects for Iris. Section 4 defines a typechecking algorithm and establishes properties such as Soundness, Completeness, Minimum Types and Decidability of Typechecking. 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* $\ldots, -1, 0, 1, \ldots$, (branching) *labels* $l, l', \ldots$, and *process variables* written $X, Y, \ldots$. A *value* is an expression name or an integer constant and is denoted with letters $v, v', \ldots$. Assertion labels, written $L, L', \ldots$, are tuples of values and are written $\langle v_1, \ldots, v_n \rangle$. Process expressions, denoted with $P, Q, \ldots$, are defined as follows:

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

Process definitions D take the form $X_1[\overrightarrow{a_1 : T_1}] = P_1$ $\texttt{and} \ldots \texttt{and}$ $X_n[\overrightarrow{a_n : T_n}] = P_n$.

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

The `request` primitive requests a session on name a. When this session is established, the fresh private channel k shall be used for message interchange. The `accept` receives a request on the same name a and generates a new private channel for message interchange to be used once the session is established. The `request` and `accept` constructs each bind all free occurrences of the immediately following channel variable, k, in the subsequent process, P. The synchronous sending and receiving of messages is achieved with $k![v]; Q$ and $k?(x)$ `in` P respectively, although, as in (Honda *et al.*, 1998), a translation to an asynchronous calculus with branching is possible. Controlled sidestepping of linearity constraints on channel usage is achieved by means of the channel delegation constructs `throw` $k[k']; P$ and `catch` $k(k')$ `in` Q. Mech-

anisms for selection of a label and branching are available as $k \lhd l; P$ and $k \rhd \{l_1 : P_1 \Box \ldots \Box\, l_n : P_n\}$. The notation $P \,|\, Q$ has already been explained; we also use `stop` for inaction. We write $(\nu a : T)P$ or $(\nu k : \bot_{\{\alpha,\overline{\alpha}\}})P$ for the usual constructs for name hiding, where the former is for expression names and the latter for channel names. T denotes a type expression (Definition 2.1) and $\bot_{\{\alpha,\overline{\alpha}\}}$ is the "complete" channel type with communication protocol given by the channel type α. Note that $\bot_{\{\alpha,\overline{\alpha}\}} = \bot_{\{\overline{\alpha},\alpha\}}$. Definitions of processes are also allowed through the `def` D `in` P construct, possibly introducing recursion, including mutual recursion. 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 a process expression, written $\texttt{fn}(P)$, and that of an assertion label, likewise written $\texttt{fn}(e)$, are defined in the standard manner (see (Bonelli *et al.*, 2003b) for details). One case that is worth clarifying is that of process parameters:

$$\texttt{fn}(\overrightarrow{a : T}, b : T') \stackrel{def}{=} \texttt{fn}(\overrightarrow{a : T}) \cup (\texttt{fn}(T') \setminus \vec{a}).$$

Thus within process parameters, the names which occur leftmost in the protocol are available as bound variables in the types that occur to their right.

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

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

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

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

A *type* is either a plain type or a channel type; we use U, U_i to range over types. The set of free names of a type U, written $\mathtt{fn}(U)$, is defined as usual (see (Bonelli *et al.*, 2003b)). 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 $\overline{\alpha}$:

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

The types α and $\overline{\alpha}$ shall be assigned to the two endpoints of a communication session. Note that $\overline{\bot_{\{\alpha,\overline{\alpha}\}}}$ 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 $\mathbf{1}$. Each of these is accompanied by a latent *effect*. An effect is a multi-set of assertion labels; we use $(\!| \ldots |\!)$ for the multi-set constructor. Multiset subtraction is defined as $e \setminus e'$, the smallest multiset e'' such that $e \leq e' + e''$, where "+" is multiset union. The special channel type $\bot_{\{\alpha,\overline{\alpha}\}}$ models a channel that has not yet been opened and shared between two subprocesses of the current process.

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). Since there is a unique U such that $x : U \in \Gamma$ for any $x \in \mathtt{dom}(\Gamma)$, we may sometimes abbreviate $\Gamma \setminus x : U$ by $\Gamma \setminus x$. For any $x \in \mathtt{dom}(\Gamma)$, we will use $\Gamma(x)$ for the unique type such that $(x : \Gamma(x)) \in \Gamma$.

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 be *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. Note that since channel names may not appear in assertion labels (hence not in $\mathtt{fn}(U_i)$), types may only depend on names which are assigned plain types. Since interaction through channel names is restricted by linearity conditions in the sense of linear logic (Girard, 1987) (see explanation of Type Par rule below), this restriction states that we do not allow types depending on linear assumptions (we do however allow types depending on shared assumptions, that is, those of plain types). The intended application of our type discipline is not disturbed by such a restriction, and it is not clear whether the technical complications of the meta-theory resulting from lifting it outweigh its benefits. In fact this restriction already appears in other settings in which linear and intuitionistic assumptions coexist, such as the linear logical framework of (Cervesato & Pfenning, 2002). The last 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 structural rules fail (Remark 2.5).

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

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

The letter Θ stands for a *process protocol*: a set of expressions of the form $X_j : (\overrightarrow{a_j : T_j})$, for $j \in 1 \ldots n$, where the $X_1, \ldots, X_n$ are all distinct, and where each $\overrightarrow{a_j : T_j}$ 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, for each environment $\overrightarrow{a_j : T_j}$ in the process protocol, $\Gamma \cup ((\overrightarrow{a_j : T_j})\{\vec{a_j} \leftarrow \vec{b_j}\})$ is well-formed, for $\vec{b_j}$ with $\vec{b_j} \cap (\texttt{dom}(\Gamma) \cup \texttt{fn}(\overrightarrow{a_j : T_j})) = \{\ \}$.

The typing rules of Iris are presented in Figure 4. 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 righthand 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 the Type Rcv rule. Note, however, that this time the latent effect of the type of the parameter of the input (i.e. "b") 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. 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[2]. Channel and name restriction (for non-channel names) are typed as expected. Type Stop types the inaction `stop`; it requires all communication through channel names to have been completed. The Type Subsum rule allows increasing the required assertion obligations of a process term. The Type Par

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

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 in order 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 do not necessarily constitute well-formed environments. Once this notion of compatibility is in place we may define how two environments are combined through environment *composition*.

Definition 2.4 (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) $(\Gamma \cdot k : \alpha) \circ (\Gamma' \cdot k : \overline{\alpha}) = (\Gamma \circ \Gamma') \cdot k : \bot_{\{\alpha,\overline{\alpha}\}}$
(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 subscript of $\bot_{\{\alpha,\overline{\alpha}\}}$ in the second clause of Definition 2.4 records the dual channel types from which it arises, and hence the name dependencies of those dual channel types.

Lemma 2.2 If Γ and Γ' and well-formed environments, and $\Gamma \asymp \Gamma'$, then $\Gamma \circ \Gamma'$ is a well-formed environment.

Process variables are typed by the rule Type PVar. Here we need to check that the process variable is one that has already been introduced and that the arguments are used at the right types. Type Def explains how to type the definitions of process variables. This is one of the most verbose rules, owing to the treatment of mutual recursion. It is worth noting that we require that each process in the definition of a process variable must have an empty effect. In the presence of recursion, if the body of a definition were allowed to have a non-empty effect, it would potentially have to have any multiple of that effect, to cover any number of recursive calls.

The following two facts follow by induction over the rules of type derivation.

$$\frac{\Gamma \cdot a : \sigma(\alpha) \cdot k' : \alpha \vdash_\Theta P\{k \leftarrow k'\} : e \quad k' \notin \mathtt{dom}(\Gamma)}{\Gamma \cdot a : \sigma(\alpha) \vdash_\Theta \mathtt{accept}\ a(k)\ \mathtt{in}\ P : e} \text{ Type Acpt}$$

$$\frac{\Gamma \cdot a : \sigma(\alpha) \cdot k' : \overline{\alpha} \vdash_\Theta P\{k \leftarrow k'\} : e \quad k' \notin \mathtt{dom}(\Gamma)}{\Gamma \cdot a : \sigma(\alpha) \vdash_\Theta \mathtt{request}\ a(k)\ \mathtt{in}\ P : e} \text{ Type Requ}$$

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

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

$$\frac{\begin{array}{c}\Gamma \cdot c : T \cdot k : \alpha\{a \leftarrow c\} \vdash_\Theta P\{b \leftarrow c\} : e \\ \mathtt{fn}(e') \setminus \{a\} \subseteq \mathtt{dom}(\Gamma) \qquad c \notin \mathtt{fn}(e \setminus e'\{a \leftarrow c\}) \cup \mathtt{fn}(\Gamma)\end{array}}{\Gamma \cdot k :\downarrow [a : T]e'; \alpha \vdash_\Theta k?(b)\ \mathtt{in}\ P : e \setminus e'\{a \leftarrow c\}} \text{ Type Rcv}$$

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

$$\frac{\Gamma \cdot k : \alpha_j \vdash_\Theta P : e \quad (1 \leq j \leq n) \quad \mathtt{fn}(e') \cup \bigcup_{i=1}^{n} \mathtt{fn}(\alpha) \subseteq \mathtt{dom}(\Gamma)}{\Gamma \cdot k : \oplus\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\}e' \vdash_\Theta k \lhd l_j; P : e + e'} \text{ Type Sel}$$

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

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

$$\frac{\Gamma \cdot k' : \bot_{\{\alpha, \overline{\alpha}\}} \vdash_\Theta P\{k \leftarrow k'\} : e \quad k' \notin \mathtt{dom}(\Gamma)}{\Gamma \vdash_\Theta (\nu k : \bot_{\{\alpha, \overline{\alpha}\}})P : e} \text{ Type CRes}$$

$$\frac{\Gamma \cdot b : T \vdash_\Theta P\{a \leftarrow b\} : e \quad b \notin \mathtt{fn}(\Gamma) \cup \mathtt{fn}(e)}{\Gamma \vdash_\Theta (\nu a : T)P : e} \text{ Type NRes}$$

$$\frac{\Gamma \vdash_\Theta \diamond \quad \mathtt{ranCh}(\Gamma) \subseteq \{\mathbf{1}, \bot_{\{\alpha, \overline{\alpha}\}}\}}{\Gamma \vdash_\Theta \mathtt{stop} : (\!| |\!)} \text{ Type Stop}$$

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

$$\frac{\Gamma \vdash_\Theta P : e \quad \Gamma' \vdash_\Theta Q : e' \quad \Gamma \asymp \Gamma'}{\Gamma \circ \Gamma' \vdash_\Theta P | Q : e + e'} \text{ Type Par}$$

Fig. 4. Well-formed process expressions

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

$$\frac{\begin{array}{c} X_1, \ldots, X_n \text{ distinct}, \quad Y_1, \ldots Y_n \notin \mathtt{dom}(\Theta) \quad Y_1, \ldots, Y_n \text{ distinct} \\ \Psi = \Theta \cup \{(Y_1 : (\overrightarrow{a_1 : T_1})) \ldots, (Y_n : (\overrightarrow{a_n : T_n}))\} \quad \vec{b_i} \notin \mathtt{dom}(\Gamma) \\ \Gamma_i = \Gamma \setminus \mathtt{chan}(\Gamma) \ \cup \bigcup_{j=1}^{|b_i|} \{b_{i,j} : T_{i,j}\{(a_{i,1}, \ldots, a_{i,j-1}) \leftarrow (b_{i,1}, \ldots, b_{i,j-1})\}\} \\ \Gamma_i \vdash_\Psi P_i\{\vec{a_i} \leftarrow \vec{b_i}\}\{\vec{X} \leftarrow \vec{Y}\} : (\!|\,|\!) \quad i = 1, \ldots, n \qquad \Gamma \vdash_\Psi Q\{\vec{X} \leftarrow \vec{Y}\} : e \end{array}}{\Gamma \vdash_\Theta \mathtt{def}\ X_1(\overrightarrow{a_1 : T_1}) = P_1 \ldots \mathtt{and} \ldots X_n(\overrightarrow{a_n : T_n}) = P_n\ \mathtt{in}\ Q : e} \text{ Type Def}$$

Fig. 6. Well-formed process expressions (cont.)

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

$$\frac{\Gamma \vdash_\Theta \diamond}{\Gamma \vdash_\Theta () : ()} \text{ Wf PP Nil} \qquad \frac{\begin{array}{c} \Gamma \vdash_\Theta (\vec{v}) : (\overrightarrow{a : T}) \quad \Gamma \vdash_\Theta v : T'\{\vec{a} \leftarrow \vec{v}\} \\ b \notin \{\vec{a}\} \cup \mathtt{dom}(\Gamma) \end{array}}{\Gamma \vdash_\Theta (\vec{v}, v) : (\overrightarrow{a : T}, b : T')} \text{ Wf PP Cons}$$

Fig. 7. Well-formed values and process parameters

Lemma 2.3 If $\Gamma \vdash_\Theta P : e$, then $\mathtt{fn}(P) \cup \mathtt{fn}(e) \cup \mathtt{fn}(\Theta) \subseteq \mathtt{dom}(\Gamma)$.

Lemma 2.4 If $\Gamma \vdash_\Theta P : e$, then $\Gamma \vdash_\Theta \diamond$.

Remark 2.5 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 $k :\uparrow [a : T]e'; \alpha$ satisfy the condition of the Exchange Lemma, since neither one depends on the other. However, when we attempt to exchange $v : T$ and $k : \alpha\{a \leftarrow v\}$ in the upper

middle 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.

3 An Example in Iris

This section considers an example in Iris that illustrates some of the programming constructs available to the user. The example we develop (Figure 8) illustrates one way of modeling a simplified electronic auction system in Iris. The three main principals that integrate the system are: Auctioneer, Seller and Buyer. In a normal processing cycle a Seller contacts the Auctioneer informing of the product and initial bidding price desired. The Auctioneer then waits to receive biddings from interested buyers. After some fixed amount of time, the Auctioneer determines that the bidding process is over and assigns the product to the highest bidder.

Two additional processes integrate the system: SellerManager and BuyerManager. Once the Auctioneer has received notification of a Seller, including product and price information, she delegates all further interaction with it to the SellerManager process. Thus she becomes free to receive requests from buyers or new sellers. Likewise, once the Auctioneer receives notification from a Buyer, including product of interest and bid, she delegates all further interaction with the Buyer to the BuyerManager. The Auctioneer thus becomes available for interaction with other buyers and sellers.

In order to keep the example simple we assume that the Auctioneer can handle at most one seller at a time and that at least one buyer shall make a bid. In order to begin operating we assume that an initial seller and buyer have been placed, namely *dummySeller* and *dummyBuyer*. Also, we shall make use of an `if-then-else` instruction which, although not listed in the syntax of Iris, is a straightforward extension. In what follows we describe the full set of principals:

Auctioneer. The Auctioneer waits to receive requests for one of three operations:

- `sell`: This is invoked by a Seller. It reads the Seller's product and initial base price together with a session name *sSELL* to be used for further contact with the Seller. Since the auctioneer can handle at most one seller, it lets the seller manager know that it must cancel the previous seller (in turn the seller manager shall contact this seller to let her/him know). It also passes on *sSELL* to the SellerManager. After that, it informs the BuyerManager that a new product and base price is in effect.
- `bid`: This is invoked by a buyer. The Auctioneer reads in product, bid and

```
Auctioneer(sAuc, sBM, sSM) =
    accept sAuc(k) in
    k ▷ {sell: k?(prod, basePrice, sSELL) in request sSM(h)in h◁ cancel;
                h!(sSELL); request sBM(h) in h◁ newProduct;
                h!(prod, basePrice); Auctioneer[sAuc, sBM, sSM],
        □ bid: k?(prod, bid, sBUY) in request sBM(h)in h◁ newBidder;
                h!(prod, bid, sBUY); Auctioneer[sAuc, sBM, sSM],
        □ timeout: request sSM(h)in h◁ sold; request sBM(h) in h◁ bought;
                Auctioneer[sAuc, sBM, sSM] }

SellerManager(sSM, currSeller) =
    accept sSM(h) in
    h ▷ {sold: request currSeller(k)in k◁ sold;
                SellerManager[sSM, dummySeller],
        □ cancel: request currSeller(k)in k◁ cancel; h?(newSeller) in
                SellerManager[sSM, newSeller] }

BuyerManager(sBM, prod, currBid, currBuyer) =
    accept sBM(h) in
    h ▷ {newProduct: h?(prod,basePrice);
                BuyerManager[sBM, prod, basePrice, dummyBuyer],
        □ newBidder: h?(prod, bid, newBuyer) in
        if bid > currBid
            then request currBuyer(k)in k◁ outBidded;
                BuyerManager[sBM, prod, bid, newBuyer]
            else request newBuyer(k)in k◁ tooLow;
                BuyerManager[sBM, prod, currBid, currBuyer]
        □ bought: request currBuyer(k)in k◁ bought;
                BuyerManager[sBM, dummyProd, 0, dummyBuyer] }

Seller(sAuc, sSell, prod, price) = request sAuc(k) in k◁ sell; k![prod, price, sSell];
                                    accept sSell(k) in
                                    k ▷ {sold: stop,
                                        □ cancel: stop}

Buyer(sAuc, sBuy, prod, price) = request sAuc(k) in k◁ bid; k![prod, price, sBuy];
                                    accept sBuy(k) in
                                    k ▷ {outBidded: stop,
                                        bought: stop,
                                        tooLow: stop}
```

Fig. 8. Full code for the auction example.

contact information from the Buyer. Then it informs the buyer manager BuyerManager that a new bidder has arrived and passes on the bidder and the other data that was input to this manager.

- `timeout`: This operation is invoked when no further bidding time is left and hence the current highest bidder has successfully acquired the item sold. It informs the seller manager SellerManager and the buyer manager BuyerManager of this situation.

SellerManager. The seller manager acts as an accumulator which holds a session name to interact with the current Seller that the Auctioneer is dealing with. The Auctioneer instructs it to do two possible things:

- `sold`: tell the seller that her item has been sold, or
- `cancel`: tell the Seller that the auction has been canceled due to the arrival of a new seller and read in the new seller.

BuyerManager. The buyer manager acts as an accumulator which holds a session name to interact with the current Buyer that has placed the highest bid. It waits to receive one of the following selections:

- `newProduct`: this is selected by the Auctioneer and informs the Buyer that a new Seller has arrived and passes on the product and base price of this product.
- `newBidder`: this is selected by the Auctioneer when a new bidder has arrived. BuyerManager reads in the bid and compares it to its current highest bid: if the former is greater than the latter then it informs the current highest bidder (i.e. *currBuyer*) that it has been outbidded and recursively calls itself with the new bidder as a parameter; otherwise the new bidder is informed that her bid is too low and BuyerManager recursively calls itself with the current highest bidder as the highest bidder for the call.
- `bought`. this is selected by the Auctioneer to inform the buyer manager that the current highest bidder has successfully acquired the product.

Seller. This process defines the behavior of a Seller. She requests a session with the Auctioneer and lets her know that she is willing to sell a product *prod* at price *price*. Also, she lets the Auctioneer know how she may be reached for further interaction. She then waits to be informed whether her product was sold or the auction was canceled due to the arrival of some new seller.

Buyer. The Buyer requests a session with the Auctioneer and selects a bidding operation. She then sends the product she is interested in and the price she is willing to pay. Also, coordinates for further interaction are provided to the Auctioneer. She then awaits one of three possible replies:

- `outBidded`: In some later cycle a new bidder has outbidded her.
- `bought`: She has successfully bought the product.
- `tooLow`: Her initial bid was too low and thus rejected.

Note that in this simple example we do not take into account error capture and recovery, such as when a bidder attempts to make a bid for an item which has not been placed for selling.

We use the expression `def` D `in` Q to denote the concurrent execution of all

parties in the system, where the process declarations D are those described above and Q is:

$$\begin{array}{c}\mathsf{Auctioneer}[sAuc, sBM, sSM] \,|\, \mathsf{BuyerManager}[sBM, 0, 0, dummyBuyer] \,| \\ \mathsf{SellerManager}[sSM, dummySeller] \,| \\ \mathsf{Buyer}[sAuc, sBuy, prod, bid] \,|\, \mathsf{Seller}[sAuc, sSell, prod, price]\end{array}$$

This expression is well-typed in the pure theory of session types (Honda *et al.*, 1998) under the following type assumptions:

$$\begin{aligned}\Gamma = {} & sAuc : \sigma(\alpha), sBM : \sigma(\beta), sSM : \sigma(\gamma),\\ & dummyBuyer : \sigma(\oplus\{\texttt{outBidded} : \mathbf{1}, \texttt{bought} : \mathbf{1}, \texttt{tooLow} : \mathbf{1}\}),\\ & dummySeller : \sigma(\oplus\{\texttt{sold:1,cancel:}\mathbf{1}\}),\\ & sBuy : \sigma(\&\{\texttt{outBidded} : \mathbf{1}, \texttt{bought} : \mathbf{1}, \texttt{tooLow} : \mathbf{1}\}),\\ & sSell : \sigma(\&\{\texttt{sold} : \mathbf{1}, \texttt{cancel} : \mathbf{1}\}),\\ & prod : \mathbf{Int}, bid : \mathbf{Int}, price : \mathbf{Int}\end{aligned}$$

where the channel types α, β and γ are

$$\begin{aligned}\alpha = {} & \&\{\texttt{sell} :\downarrow [\mathbf{Int}, \mathbf{Int}, \sigma(\&\{\texttt{sold} : \mathbf{1}, \texttt{cancel} : \mathbf{1}\})]; \mathbf{1},\\ & \texttt{bid} :\downarrow [\mathbf{Int}, \mathbf{Int}, \sigma(\&\{\texttt{outBidded} : \mathbf{1}, \texttt{bought} : \mathbf{1}, \texttt{tooLow} : \mathbf{1}\})]; \mathbf{1},\\ & \texttt{timeout} : \mathbf{1}\}\\ \beta = {} & \&\{\texttt{newProduct} :\downarrow [\mathbf{Int}, \mathbf{Int}]; \mathbf{1},\\ & \texttt{newBidder} :\downarrow [\mathbf{Int}, \mathbf{Int}, \sigma(\oplus\{\texttt{outBidded} : \mathbf{1}, \texttt{bought} : \mathbf{1}, \texttt{tooLow} : \mathbf{1}\})]; \mathbf{1},\\ & \texttt{bought} : \mathbf{1}\}\\ \gamma = {} & \&\{\texttt{sold} : \mathbf{1}, \texttt{cancel} :\downarrow [\sigma(\oplus\{\texttt{sold} : \mathbf{1}, \texttt{cancel} : \mathbf{1}\})]; \mathbf{1}\}\end{aligned}$$

Note that the auction example is also typable in the type system introduced in Section 2 if we assume that all effects are empty $(\!|\,|\!)$.

We provide an informal explanation of the type assigned to the session name $sAuc$. This session name is used by the Auctioneer. The type $\sigma(\alpha)$ is a session type and is an abstraction of a pair of dual channel types, namely α and $\overline{\alpha}$. One endpoint of the communication is assumed to abide to the interaction pattern specified by α, while the other endpoint it assumed to abide to that specified by its dual. The & type constructor indicates that the Auctioneer must accept three operations `sell`, `bid` and `timeout`. If the first of these operations is invoked, then the Auctioneer must read in a triple consisting of an integer, another integer and a session name of type $\sigma(\&\{\texttt{sold} : \mathbf{1}, \texttt{cancel} : \mathbf{1}\})$. Similar comments apply to the `bid` operation. In the case of the `timeout` operation, no further interaction is expected on this channel.

Session types such as those of *sAuc*, *sBM* and *sSM* express how these long term communication abstractions behave *independently* of each other, even though they all belong to a common specification, namely that of the protocol specifying how Auctioneer, SellerManager, and the other parties should interact in order to carry out a specific operation (such as placing a bid). This fact may be witnessed as follows. Consider replacing the code for the bid operation of the Auctioneer by

Example 3.1 (Changing bids)

`bid`: $k?(prod, bid, sBUY)$ `in`
    `request` $sBM(h)$`in` $h \triangleleft$ `newBidder`;
    $h!(prod, bid - 10, sBUY)$; $\mathsf{Auctioneer}[sAuc, sBM, sSM]$,

This version of the `bid` operation places a smaller bid than the one originally communicated to the auctioneer by the bidder. Unfortunately, the resulting electronic auction system is declared typable by the pure theory of session types, under the *same* typing assumptions as the original system. Other examples of the lack of expressiveness of the pure theory of session types are described in Bonelli *et al.* (2003a).

The type system for Iris allows such badly behaved variants of the honest Auctioneer to be detected by introducing correspondence assertions into the picture and applying the typechecking algorithm described in this article. Indeed, in Bonelli *et al.* (2003a) a notion of *safe* process is P introduced following Gordon & Jeffrey (2001b,a, 2003b,a). Informally, it states that all `end` L assertions are corresponded by a `begin` L assertion, in every possible execution of P. Also, it is shown (Bonelli *et al.* (2003a)) that all processes which are typable with the empty effect $(\!|\,|\!)$ are safe. Example 3.1 may thus be addressed by the insertion of appropriate effects and then showing that the resulting code does not typecheck with the empty effect.

4 Typechecking

We define a typechecking function $Ch(\Gamma, \Theta, P)$, where Γ is an environment, Θ is a process protocol, and P is a process. The function $Ch(\Gamma, \Theta, P)$ is defined by recursion over the length of P, and will either return **fail**or the minimum possible effect for P.

We use several auxiliary functions:

- $ChEnv(\Gamma, \Theta)$, which checks the well-formation of contexts returning **true** if

and only if $\Gamma \vdash_\Theta \diamond$,

- $ChTy(\Gamma, \Theta, v, T)$ which checks the types of values returning **true** if and only if $\Gamma \vdash_\Theta v : T$, and
- $ChVec(\Gamma, \Theta, \vec{v}, \overrightarrow{a : T})$ which checks the well-formation of process parameters.

$ChEnv(\Gamma, \Theta)$ checks that the environment Γ and the derived environments $\Gamma \cup ((\overrightarrow{a_j : T_j})\{\vec{a_j} \leftarrow \vec{b_j}\})$ are well-formed, for all $\overrightarrow{a_j : T_j}$ in the process protocol Θ, and any $\vec{b_j}$ with $\vec{b_j} \cap (\mathtt{dom}(\Gamma) \cup \mathtt{fn}(\overrightarrow{a_j : T_j})) = \{\ \}$. Note that the choice of b_j does not matter, because if $\Gamma \cup ((\overrightarrow{a_j : T_j})\{\vec{a_j} \leftarrow \vec{b_j}\})$ is well-formed, then $\Gamma \cup ((\overrightarrow{a_j : T_j})\{\vec{a_j} \leftarrow \vec{b_j}\}\{\vec{b_j} \leftarrow \vec{c_j}\}) = \Gamma \cup ((\overrightarrow{a_j : T_j})\{\vec{a_j} \leftarrow \vec{c_j}\})$ is well-formed for any $\vec{c_j}$ with $\vec{c_j} \cap (\mathtt{dom}(\Gamma) \cup \mathtt{fn}(\overrightarrow{a_j : T_j})) = \{\ \}$. To show that an environment is well-formed requires checking conditions **C1**, **C2**, and **C3**. To check **C3**, we construct the directed graph with edges pointing from names in domain of the environment to each of the free names in the type the environment associates with it. (In the process, we can easily check conditions **C1** and **C2**.) Once we have constructed the graph, we apply any standard algorithm to check that it is cycle free. If v is a numerical constant $ChTy(\Gamma, \Theta, v, T)$ checks if $T = \mathbf{Int}$, otherwise it checks if $v : T$ is in the environment Γ, and then ccalls $ChEnv(\Gamma, \Theta)$.

We define $ChVec$ by recursion over the length of the process parameter:

Definition 4.1

- $ChVec(\Gamma, \Theta, (\ ), (\ )) = ChEnv(\Gamma, \Theta)$
- $ChVec(\Gamma, \Theta, (\vec{v}, v), (\overrightarrow{a : T}, b : T')) =$

$$\begin{cases} ChVec(\Gamma, \Theta, \vec{v}, \overrightarrow{a : T}) & \text{if } ChTy(\Gamma, \Theta, v, T'\{\vec{a} \leftarrow \vec{v}\}) = \textsf{true} \\ & \text{and } b \notin \{\vec{a}\} \cup \mathtt{dom}(\Gamma) \\ \textsf{false} & \text{otherwise.} \end{cases}$$

□

When defining the clause of Ch for parallel composition, it will be useful to have a few special-purpose definitions.

Definition 4.2 *Extended environments* extend plain environments by allowing channel names to be associated with plain types of the form $\sigma(\alpha)$ (session types). Given an extend environment Γ, let

$$\mathtt{domChoice}(\Gamma) = \{k \in \mathtt{domCh}(\Gamma) \mid \Gamma(k) = \sigma(\alpha) \text{ for some channel type } \alpha\}.$$

We will call a regular environment Γ' a *specialization* of an extended environment Γ if $\mathtt{dom}(\Gamma') = \mathtt{dom}(\Gamma)$, and for all $x \in \mathtt{dom}(\Gamma) \setminus \mathtt{domChoice}(\Gamma)$ we

have $\Gamma'(x) = \Gamma(x)$, and for all $k \in \texttt{domChoice}(\Gamma)$, if $\Gamma(k) = \sigma(\alpha)$, then either $\Gamma'(k) = \alpha$ or $\Gamma'(k) = \overline{\alpha}$. Let

$$\Sigma(\Gamma_i) = \{\Gamma'_i \mid \Gamma'_i \text{ is a specialization of } \Gamma_i\}.$$

Definition 4.3 Let P and Q be processes and Γ be an environment. We define the *split* of Γ with respect to P and Q by

- If $\texttt{fn}(P) \cup \texttt{fn}(Q) \not\subseteq \texttt{dom}(\Gamma)$, then $split(\Gamma, P, Q) = \textsf{fail}$.
- If there exists $k \in \texttt{fn}(P) \cap \texttt{fn}(Q)$ such that $\Gamma(k) \neq \bot_{\{\alpha,\overline{\alpha}\}}$ for any α, then $split(\Gamma, P, Q) = \textsf{fail}$.
- Otherwise, $split(\Gamma, P, Q) = (\Gamma_1, \Gamma_2)$ where Γ_1 and Γ_2 are extended environments defined by the following:
  - · $\texttt{dom}(\Gamma_1) \subseteq \texttt{dom}(\Gamma)$ and $\texttt{dom}(\Gamma_2) \subseteq \texttt{dom}(\Gamma)$.
  - · For all $a \in \texttt{domPl}(\Gamma)$, we have $a \in \texttt{dom}(\Gamma_1) \cap \texttt{dom}(\Gamma_2)$ and $\Gamma_1(a) = \Gamma_2(a) = \Gamma(a)$.
  - · For $k \in \texttt{fn}(P) \cap \texttt{fn}(Q)$ and $\Gamma(k) = \bot_{\{\alpha,\overline{\alpha}\}}$, we have $\Gamma_1(k) = \Gamma_2(k) = \sigma(\alpha)$.
  - · For all $k \in \texttt{fn}(P)$ but $k \notin \texttt{fn}(Q)$, we will have $k \notin \texttt{dom}(\Gamma_2)$ and $\Gamma_1(k) = \Gamma(k)$.
  - · For all $k \in \texttt{fn}(Q)$ but $k \notin \texttt{fn}(P)$, we will have $k \notin \texttt{dom}(\Gamma_1)$ and $\Gamma_2(k) = \Gamma(k)$.
  - · For all $k \in \texttt{dom}(\Gamma) \setminus (\texttt{fn}(P) \cup \texttt{fn}(Q))$, we will (arbitrarily) assign $\Gamma_1(k) = \Gamma(k)$, and have $k \notin \texttt{dom}(\Gamma_2)$.

These definitions are used in the clause defining $Ch(\Gamma, \Theta, P|\,Q)$. The function *split* is used to "split" the environment Γ into two, Γ'_1 and Γ'_2 such that $\Gamma = \Gamma'_1 \circ \Gamma'_2$. The difficulty is that when $\Gamma(k) = \bot_{\{\alpha,\overline{\alpha}\}}$, we may need to send $k : \alpha$ to one side and $k : \overline{\alpha}$ to the other, but we do not know which side is going to get which. The sets $\Sigma(\Gamma_1)$ and $\Sigma(\Gamma_2)$, where $(\Gamma_1, \Gamma_2) = split(\Gamma, P, Q)$, allow us to enumerate a sufficient set of possibilities for Γ'_1 and Γ'_2.

There were several arbitrary choices made in the definition of *split*. First, we could have sent $k : \Gamma(k)$ to Γ_2 for any or all of the $k \in \texttt{dom}(\Gamma) \setminus (\texttt{fn}(P) \cup \texttt{fn}(Q))$. Secondly, if $\Gamma(k) = \bot_{\{1,\overline{1}\}}$, then we had an additional option of assigning $k : 1$ to each of Γ_1 and Γ_2. The use of these arbitrary choices in the definition of Γ is justified by Lemma 4.9 and Lemma 4.10.

The function Ch is defined by induction on the length of the process it takes as its third argument.

Definition 4.4 (Typechecking Algorithm) If $\texttt{fn}(P) \not\subseteq \texttt{dom}(\Gamma)$, then $Ch(\Gamma, \Theta, P) = \textsf{fail}$. In all subsequent cases we will assume that $\texttt{fn}(P) \subseteq \texttt{dom}(\Gamma)$.

To ensure well-definedness, we assume that all classes of names are totally

ordered and that when choosing a fresh name we choose the least fresh name.

- **Case** $\texttt{accept}\ a(k)\ \texttt{in}\ P$**:**
  Let k' be a fresh channel variable not present in $\texttt{dom}(\Gamma)$.
  Then

$$Ch(\Gamma, \Theta, \texttt{accept}\ a(k)\ \texttt{in}\ P) =$$
$$\begin{cases} Ch(\Gamma \cdot k' : \alpha, \Theta, P\{k \leftarrow k'\}) & \text{if } \Gamma(a) = \sigma(\alpha) \\ \textsf{fail} & \text{otherwise.} \end{cases}$$

- **Case** $\texttt{request}\ a(k)\ \texttt{in}\ P$**:**
  Let k' be a fresh channel variable not present in $\texttt{dom}(\Gamma)$. Then

$$Ch(\Gamma, \Theta, \texttt{request}\ a(k)\ \texttt{in}\ P) =$$
$$\begin{cases} Ch(\Gamma \cdot k' : \overline{\alpha}, \Theta, P\{k \leftarrow k'\}) & \text{if } \Gamma(a) = \sigma(\alpha) \\ \textsf{fail} & \text{otherwise.} \end{cases}$$

- **Case** $\texttt{begin}\ \ L; P$**:**

$$Ch(\Gamma, \Theta, \texttt{begin}\ \ L; P) =$$
$$\begin{cases} Ch(\Gamma, \Theta, P) \setminus (\!|\, L \,|\!) & \text{if } Ch(\Gamma, \Theta, P) \neq \textsf{fail} \\ \textsf{fail} & \text{otherwise.} \end{cases}$$

- **Case** $\texttt{end}\ \ L; P$**:**

$$Ch(\Gamma, \Theta, \texttt{end}\ \ L; P) =$$
$$\begin{cases} Ch(\Gamma, \Theta, P) + (\!|\, L \,|\!) & \text{if } Ch(\Gamma, \Theta, P) \neq \textsf{fail} \\ \textsf{fail} & \text{otherwise.} \end{cases}$$

- **Case** $k![v]P$**:**

$$Ch(\Gamma, \Theta, k![v]P) =$$
$$\begin{cases} Ch((\Gamma \setminus k) \cdot k : \alpha\{a \leftarrow v\}, \Theta, P) + e\{a \leftarrow v\} & \\ & \text{if } \Gamma(k) =\uparrow [a : T]e; \alpha \text{ for some } a, T \text{ and } e \text{ with} \\ & \texttt{fn}(e) \setminus \{a\} \subseteq \texttt{dom}(\Gamma), \text{ and } ChTy(\Gamma, \Theta, v, T) \neq \\ & \textsf{fail}, \text{ and } Ch((\Gamma \setminus k) \cdot k : \alpha\{a \leftarrow v\}, \Theta, P) \neq \textsf{fail} \\ \textsf{fail} & \text{otherwise.} \end{cases}$$

- **Case** $k?(b)\ \texttt{in}\ P$**:**
  Let c be a fresh expression name not in $\texttt{fn}(\Gamma)$. Then

$Ch(\Gamma, \Theta, k?(b)\ \mathtt{in}\ P) =$

$$\begin{cases} Ch((\Gamma \setminus k) \cdot c : T \cdot k : \alpha\{a \leftarrow c\}, \Theta, P\{b \leftarrow c\}) \setminus e\{a \leftarrow c\} & \text{if } \Gamma(k) =\downarrow [a : T]e; \alpha \text{ for some } a, T \text{ and } e \text{ with } \mathtt{fn}(e) \setminus \{a\} \subseteq \mathtt{dom}(\Gamma) \text{ and } Ch((\Gamma \setminus k) \cdot c : T \cdot k : \alpha\{a \leftarrow c\}, \Theta, P\{b \leftarrow c\}) \neq \textsf{fail}, \text{ and } c \notin \mathtt{fn}\,(Ch((\Gamma \setminus k) \cdot c : T \cdot k : \alpha\{a \leftarrow c\}, \Theta, P\{b \leftarrow c\}) \setminus e\{a \leftarrow c\}) \\ \textsf{fail} & \text{otherwise.} \end{cases}$$

- **Case** $k \triangleright \{l_1 : P_1, \ldots, l_n : P_n\}$**:**

  $Ch(\Gamma, \Theta, k \triangleright \{l_1 : P_1, \ldots, l_n : P_n\}) =$

$$\begin{cases} (\bigvee_{i=1}^{n} Ch((\Gamma \setminus k) \cdot k : \alpha_i, \Theta, P_i)) \setminus e & \text{if } \Gamma(k) = \&\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\}e, \text{ for some } \alpha_1, \ldots, \alpha_n \text{ and } e \text{ with } \mathtt{fn}(e) \subseteq \mathtt{dom}(\Gamma), \text{ and } Ch((\Gamma \setminus k) \cdot k : \alpha_i, \Theta, P_i) \neq \textsf{fail} \text{ for } i = 1, \ldots, n, \text{ where } \bigvee_{i=1}^{n} e_i \text{ is the least upper bound of the effects } e_i\text{'s.} \\ \textsf{fail} & \text{otherwise.} \end{cases}$$

- **Case** $k \triangleleft l_j; P$**:**

  $Ch(\Gamma, \Theta, k \triangleleft l_j; P) =$

$$\begin{cases} Ch((\Gamma \setminus k) \cdot k : \alpha_j, \Theta, P) + e & \text{where } \Gamma(k) = \oplus\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\}e \text{ for some } l_1, \ldots, l_n \text{ with } l_j \in \{l_1, \ldots, l_n\}, \text{ some } \alpha_1, \ldots, \alpha_n \text{ and some } e, \text{ provided that either } Ch((\Gamma \setminus k) \cdot k : \alpha_j, \Theta, P) \neq \textsf{fail} \text{ and } \mathtt{fn}(e) \cup \bigcup_{i=1}^{n} \mathtt{fn}(\alpha_1) \subseteq \mathtt{dom}(\Gamma). \\ \textsf{fail} & \text{otherwise.} \end{cases}$$

- **Case** $\mathtt{throw}\ k[k']P$**:**

  $Ch(\Gamma, \Theta, \mathtt{throw}\ k[k']P) =$

$$\begin{cases} Ch((\Gamma \setminus k \setminus k') \cdot k : \alpha, \Theta, P) + e & \text{if } k \neq k' \text{ and } \Gamma(k) =\uparrow [\beta]e; \alpha \text{ and } \Gamma(k') = \beta \neq \mathbf{1} \text{ for some } \beta \text{ and } e \text{ with } \mathtt{fn}(e) \cup \mathtt{fn}(\beta) \subseteq \mathtt{dom}(\Gamma), \text{ and } Ch((\Gamma \setminus k \setminus k') \cdot k : \alpha, \Theta, P) \neq \textsf{fail} \\ \textsf{fail} & \text{otherwise.} \end{cases}$$

- **Case** $\texttt{catch}\ k(k')\ \texttt{in}\ P$**:**
  Let k'' be a fresh expression name not in $\texttt{dom}(\Gamma)$. Then

$$
Ch(\Gamma,\Theta,\texttt{catch}\ k(k')\ \texttt{in}\ P) = \begin{cases} Ch((\Gamma\setminus k)\cdot k'':\beta\cdot k:\alpha,\Theta,P\{k'\leftarrow k''\})\setminus e & \\ & \text{if } \Gamma(k) =\downarrow [\beta]e;\alpha \text{ for some } \beta \text{ and } e \text{ with } \texttt{fn}(e)\subseteq\texttt{dom}(\Gamma) \text{ and } Ch((\Gamma\setminus k)\cdot k'':\beta\cdot k:\alpha,\Theta,P\{k'\leftarrow k''\})\neq \textsf{fail} \\ \textsf{fail} & \text{otherwise.} \end{cases}
$$

- **Case** $(\nu k:\bot_{\{\alpha,\overline{\alpha}\}})P$**:**
  Let k' be a fresh name such that $k'\notin\texttt{dom}(\Gamma)$. Then

$$
Ch(\Gamma,\Theta,(\nu k:\bot_{\{\alpha,\overline{\alpha}\}})P) = Ch(\Gamma\cdot k':\bot_{\{\alpha,\overline{\alpha}\}},\Theta,P\{k\leftarrow k'\})
$$

- **Case** $(\nu a:T)P$**:**
  Let b be a fresh name such that $b\notin\texttt{fn}(\Gamma)$. Then

$$
Ch(\Gamma,\Theta,(\nu a:T)P) = \begin{cases} Ch(\Gamma\cdot b:T,\Theta,P\{a\leftarrow b\}) & \text{if } Ch(\Gamma\cdot b:T,\Theta,P\{a\leftarrow b\})\neq\textsf{fail} \text{ and } b\notin\texttt{fn}(Ch(\Gamma\cdot b:T,\Theta,P\{a\leftarrow b\})) \\ \textsf{fail} & \text{otherwise.} \end{cases}
$$

- **Case** $\texttt{stop}$**:**

$$
Ch(\Gamma,\Theta,\texttt{stop}) = \begin{cases} (\!|\,|\!) & \text{if } ChEnv(\Gamma,\Theta) = \textsf{true} \text{ and } \texttt{ranCh}(\Gamma)\subseteq\{\mathbf{1},\bot_{\{\alpha,\overline{\alpha}\}}\} \\ \textsf{fail} & \text{otherwise.} \end{cases}
$$

- **Case** $P|\,Q$**:**
  If $split(\Gamma,P,Q) = \textsf{fail}$, then $Ch(\Gamma,\Theta,P|\,Q)$. If $split(\Gamma,P,Q)\neq\textsf{fail}$, let $(\Gamma_1,\Gamma_2) = split(\Gamma,P,Q)$. Notice that $\texttt{domChoice}(\Gamma_1) = \texttt{domChoice}(\Gamma_2)$ for the extended environments Γ_1 and Γ_2 defined above. Also notice that the number of regular environments that are a specialization of a given extended environment Γ_i is $2^{|\texttt{domChoice}(\Gamma_i)|}\leq 2^{|\texttt{domCh}(\Gamma_i)|}$.

Then

$$Ch(\Gamma, \Theta, P|\, Q) =$$
$$\begin{cases} Ch(\Gamma'_1, \Theta, P) + Ch(\Gamma'_2, \Theta, Q) & \text{there exists } \Gamma'_1 \in \Sigma(\Gamma_1) \text{ and } \Gamma'_2 \in \Sigma(\Gamma_2) \text{ such that } Ch(\Gamma'_1, \Theta, P) \neq \textsf{fail} \text{ and } Ch(\Gamma'_2, \Theta, Q) \neq \textsf{fail} \text{ and for all } k \in \texttt{domChoice}(\Gamma_1) = \texttt{domChoice}(\Gamma_2),\ \Gamma'_1(k) = \overline{\Gamma'_2(k)} \\ \textsf{fail} & \text{otherwise.} \end{cases}$$

By Lemma 4.1, there is at most one specialization $\Gamma'_1 \in \Sigma(\Gamma_1)$ and at most one specialization $\Gamma'_2 \in \Sigma(\Gamma_2)$ such that $Ch(\Gamma'_1, \Theta, P) \neq \textsf{fail}$ and $Ch(\Gamma'_2, \Theta, Q) \neq \textsf{fail}$.

- **Case** $X[\vec{v}]$**:**

$$Ch(\Gamma, \Theta, X[v_1, \ldots, v_n]) =$$
$$\begin{cases} (\!|\,|\!) & \text{if } \texttt{ranCh}(\Gamma) \subseteq \{\mathbf{1}, \bot_{\{\alpha,\overline{\alpha}\}}\},\ X \in \texttt{dom}(\Theta),\ \Theta(X) = ((a_1 : T_1), \ldots, (a_m : T_m)) \text{ with } n = m, \text{ and } ChVec(\Gamma, \Theta, \vec{v}, \overrightarrow{a : T}) = \textsf{true} \text{ for all } i \text{ with } 1 \leq i \leq n. \\ \textsf{fail} & \text{otherwise.} \end{cases}$$

- **Case** $\texttt{def}\ X_1(\overrightarrow{a_1 : T_1}) = P_1 \ldots \texttt{and} \ldots X_n(\overrightarrow{a_n : T_n}) = P_n\ \texttt{in}\ Q$**:**
Let $Y_i, \ldots Y_n$ be fresh process variables such that $Y_i \notin \texttt{dom}(\Theta)$ for $1 \leq i \leq n$. Let $\{\vec{X} \leftarrow \vec{Y}\}$ mean the simultaneous substitution of Y_i for X_i for $1 \leq i \leq n$. For each i with $1 \leq i \leq n$, let $\vec{b_i}$ be a vector of distinct fresh variables such that $|\vec{b_i}| = |\vec{a_i}|$ and $b_{i,j} \notin \texttt{dom}(\Gamma)$ for $1 \leq j \leq |\vec{a_i}|$. Let $\Psi = \Theta \cup \{(Y_1 : \overrightarrow{a_1 : T_1}), \ldots, (Y_n, \overrightarrow{a_n : T_n})\}$, and for $1 \leq i \leq n$ let $\Gamma_i = \Gamma \setminus \texttt{chan}(\Gamma) \cup \bigcup_{j=1}^{|b_i|} \{b_{i,j} : T_{i,j}\{(a_{i,1}, \ldots, a_{i,j-1}) \leftarrow (b_{i,1}, \ldots, b_{i,j-1})\}\}$.

$$Ch(\Gamma, \Theta, \texttt{def}\ X_1(\overrightarrow{a_1 : T_1}) = P_1 \ldots \texttt{and} \ldots X_n(\overrightarrow{a_n : T_n}) = P_n\ \texttt{in}\ Q) =$$
$$\begin{cases} Ch(\Gamma, \Psi, Q\{\vec{X} \leftarrow \vec{Y}\}) & \text{if } Ch(\Gamma_i, \Psi, P_i\{a_i \leftarrow b_i\}) = (\!|\,|\!) \text{ for all } i \text{ with } 1 \leq i \leq n. \\ \textsf{fail} & \text{otherwise.} \end{cases}$$

□

In order to prove that Ch defines a typechecking algorithm we first need to show that it defines a total function. There are several points in the definition where choices are made, and we wish to show that the end result does not

depend on any particular choice. There are two kinds of choices. The first type of choice is of fresh names. To ensure well-definedness, we take the least fresh name in each case, but we need to know that a different choice does not affect the result of Ch. The second type of choice appears in the case of parallel composition $Q|\, R$, where we "split" our environment Γ into two extended environments Γ_1 and Γ_2, and then choose specializations Γ_1' and Γ_2' respectively, such that $Ch(\Gamma_1', \Theta, Q) \neq \textsf{fail}$ and $Ch(\Gamma_2', \Theta, R) \neq \textsf{fail}$. The next lemma shows that, if such Γ_1' and Γ_2' exist, they are unique.

Lemma 4.1 Let Γ and Γ' be two environments such that $\texttt{dom}(\Gamma) = \texttt{dom}(\Gamma')$ and for any $a \in \texttt{domPl}(\Gamma)$ we have $\Gamma(a) = \Gamma'(a)$, and for any $k' \in \texttt{domCh}(\Gamma)$, we either have $\Gamma(k') = \Gamma'(k')$ or $\Gamma(k') = \overline{\Gamma'(k')}$. If P is a process with $k \in \texttt{fn}(P)$ with $\Gamma'(k) = \overline{\Gamma(k)} \neq \Gamma(k)$, and $Ch(\Gamma, \Theta, P) \neq \textsf{fail}$ then $Ch(\Gamma', \Theta, P) = \textsf{fail}$.

Proof: There are two cases to consider.

(1) If $\texttt{fn}(P) \not\subseteq \texttt{dom}(\Gamma')$ then $Ch(\Gamma', \Theta, P) = \textsf{fail}$.
(2) If $\texttt{fn}(P) \subseteq \texttt{dom}(\Gamma') = \texttt{dom}(\Gamma)$, then we proceed by case analysis on the form of P, and the argument follows by induction on the length of P. We show here the most complicated cases.

- $P = h \triangleright \{l_1 : P_1, \ldots, l_n : P_n\}$: Since $Ch(\Gamma, \Theta, P) \neq \textsf{fail}$, we must have $\Gamma(h) = \&\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\}e$, for some $\alpha_1, \ldots, \alpha_n$ and e with $\texttt{fn}(e) \subseteq \texttt{dom}(\Gamma)$, and $Ch((\Gamma \setminus h) \cdot h : \alpha_i, \Theta, P_i) \neq \textsf{fail}$ for $i \in \{1, \ldots, n\}$. Since $k \in \texttt{fn}(P)$, we must have either $k = h$ or $k \in \texttt{fn}(P_i)$ for at least one $i \in \{1, \ldots, n\}$. If $\Gamma'(h) = \overline{\Gamma(h)}$, and in particular if $k = h$, then $\Gamma'(h) = \oplus\{l_1 : \overline{\alpha_1}, \ldots, l_n : \overline{\alpha_n}\}e \neq \&\{l_1 : \beta_1, \ldots, l_n : \beta_n\}e'$, for any $\beta_1, \ldots, \beta_n$ and e'. Therefore, $Ch(\Gamma', \Theta, h \triangleright \{l_1 : P_1, \ldots, l_n : P_n\}) = \textsf{fail}$. Thus, we may assume that $k \neq h$, $\Gamma'(h) = \Gamma(h)$, and $k \in \texttt{fn}(P_i)$ for some i. Then, by the induction hypothesis, since $Ch((\Gamma \setminus h) \cdot h : \alpha_i, \Theta, P_i) \neq \textsf{fail}$, we must have that $Ch((\Gamma' \setminus h) \cdot h : \alpha_i, \Theta, P_i) = \textsf{fail}$. Since $\Gamma'(h) = \&\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\}e$ but $Ch((\Gamma' \setminus h) \cdot h : \alpha_i, \Theta, P_i) = \textsf{fail}$, we again have that $Ch(\Gamma', \Theta, h \triangleright \{l_1 : P_1, \ldots, l_n : P_n\}) = \textsf{fail}$.
- $P = \texttt{throw}\ h[h']Q$: Since $Ch(\Gamma, \Theta, \texttt{throw}\ h[h']Q) \neq \textsf{fail}$ we have that $h \neq h'$ and $\Gamma(h) =\uparrow [\beta]e; \alpha$ and $\Gamma(h') = \beta$ for some α, β and e with $\texttt{fn}(e) \cup \texttt{fn}(\beta) \subseteq \texttt{dom}(\Gamma)$, and $Ch((\Gamma \setminus h \setminus h') \cdot h : \alpha, \Theta, Q) \neq \textsf{fail}$. If $\Gamma'(h) = \overline{\Gamma(h)}$, then $\Gamma'(h) =\downarrow [\beta]e; \overline{\alpha} \neq\uparrow [\beta']e'; \alpha'$ for any β', e' and α'. Therefore, $Ch(\Gamma', \Theta, \texttt{throw}\ h[h']Q) = \textsf{fail}$. If $\Gamma'(h') = \overline{\Gamma(h')} \neq \Gamma(h')$, then $\Gamma'(h') = \overline{\beta} \neq \beta$, and thus again $Ch(\Gamma', \Theta, \texttt{throw}\ h[h']Q) = \textsf{fail}$. Finally, if $\Gamma'(h) = \Gamma(h)$ and $\Gamma'(h') = \Gamma(h')$ (and hence $h \neq k \neq h'$), then by the induction hypothesis $Ch((\Gamma' \setminus h \setminus h') \cdot h' : \alpha, \Theta, Q) = \textsf{fail}$, and hence $Ch(\Gamma', \Theta, \texttt{throw}\ h[h']Q) = \textsf{fail}$.
- $P = Q|\, R$: Since $k \in \texttt{fn}(P)$, we must have that $k \in \texttt{fn}(Q) \cup \texttt{fn}(R)$. Since $Ch(\Gamma, \Theta, Q|\, R) \neq \textsf{fail}$, we have for all $h \in \texttt{fn}(Q) \cap \texttt{fn}(R)$ that $\Gamma(h) = \perp_{\{\alpha, \overline{\alpha}\}}$ for some α, and $\overline{\Gamma(h)}$ does not exist. Therefore, $k \notin \texttt{fn}(Q) \cap \texttt{fn}(R)$. If

$k \in \mathtt{fn}(Q) \setminus \mathtt{fn}(R)$, then in the extended environment Γ_1, $\Gamma_1(k) = \Gamma(k)$. Also, $\Gamma'_1(k) = \Gamma'(k)$ for the comparable extended environment Γ'_1. Let Π be a specialization of Γ_1 and Π' be a specialization of Γ'_1. Then $\mathtt{domPl}(\Pi) = \mathtt{domPl}(\Gamma) = \mathtt{domPl}(\Gamma') = \mathtt{domPl}(\Pi')$ and for all $a \in \mathtt{domPl}(\Pi)$ we have $\Pi(a) = \Gamma(a) = \Gamma'(a) = \Pi'(a)$. For channel names

$$\begin{aligned}\mathtt{domCh}(\Pi) &= (\mathtt{fn}(Q) \cap \mathtt{fn}(R) \cap \mathtt{domCh}(\Gamma)) \cup (\mathtt{domCh}(\Gamma) \setminus \mathtt{fn}(R)) \\ &= (\mathtt{fn}(Q) \cap \mathtt{fn}(R) \cap \mathtt{domCh}(\Gamma')) \cup (\mathtt{domCh}(\Gamma') \setminus \mathtt{fn}(R)) \\ &= \mathtt{domCh}(\Pi')\end{aligned}$$

To apply the induction hypothesis to $Ch(\Pi, \Theta, Q)$ and $Ch(\Pi', \Theta, Q)$, we need to know for all $h \in \mathtt{domCh}(\Pi)$ that either $\Pi(h) = \Pi'(h)$ or $\Pi(h) = \overline{\Pi'(h)}$, and that $\Pi(k) = \Pi'(k)$. For all $h \in \mathtt{fn}(Q) \cap \mathtt{fn}(R)$, $\Gamma(h) = \Gamma'(h) = \bot_{\{\alpha,\overline{\alpha}\}}$, so both $\Gamma_1(h) \in \{\sigma(\alpha), \sigma(\overline{\alpha})\}$ and $\Gamma'_1(h) \in \{\sigma(\alpha), \sigma(\overline{\alpha})\}$, so both $\Pi(h) \in \{\alpha, \overline{\alpha}\}$ and $\Pi'(h) \in \{\alpha, \overline{\alpha}\}$. Therefore, either $\Pi(h) = \Pi'(h)$ or $\Pi(h) = \overline{\Pi'(h)}$. For all $h \in \mathtt{domCh}(\Gamma) \setminus \mathtt{fn}(R)$ we have $\Gamma_1(h) = \Gamma(h)$ and $\Gamma'_1(h) = \Gamma'(h)$, so again either $\Pi(h) = \Pi'(h)$ or $\Pi(h) = \overline{\Pi'(h)}$. Therefore, for all $h \in \mathtt{domCh}(\Pi)$, we have either $\Pi(h) = \Pi'(h)$ or $\Pi(h) = \overline{\Pi'(h)}$. Moreover, since $k \in \mathtt{fn}(Q) \setminus \mathtt{fn}(R)$, we have that $\Pi(k) = \Gamma(k)$ and $\Pi'(k) = \Gamma'(k)$, so $\Pi(k) = \overline{\Pi'(k)}$. Therefore, by induction, we know that if $Ch(\Pi, \Theta, Q) \neq \textsf{fail}$, then $Ch(\Pi', \Theta, Q) = \textsf{fail}$. Since the choice of Π and Π' was arbitrary, we must have that if there exists $\Pi \in \Sigma(\Gamma_1)$ such that $Ch(\Pi, \Theta, Q) \neq \textsf{fail}$, then for all $\Pi' \in \Sigma(\Gamma'_1)$ we must have $Ch(\Pi', \Theta, Q) = \textsf{fail}$. Therefore, if $Ch(\Gamma, \Theta, Q|\, R) \neq \textsf{fail}$, then $Ch(\Gamma', \Theta, Q|\, R) = \textsf{fail}$.

The last case of $k \in \mathtt{fn}(R) \setminus \mathtt{fn}(Q)$ is much the same except we focus on R, Γ_2 and Γ'_2 instead of Q, Γ_1 and Γ'_1, and for channel names, we have

$$\begin{aligned}\mathtt{domCh}(\Pi) &= \mathtt{domCh}(\Gamma) \cap \mathtt{fn}(R) \\ &= (\mathtt{fn}(Q) \cap \mathtt{fn}(R) \cap \mathtt{domCh}(\Gamma)) \cup (\mathtt{domCh}(\Gamma) \cap (\mathtt{fn}(R) \setminus \mathtt{fn}(Q))) \\ &= (\mathtt{fn}(Q) \cap \mathtt{fn}(R) \cap \mathtt{domCh}(\Gamma')) \cup (\mathtt{domCh}(\Gamma') \cap (\mathtt{fn}(R) \setminus \mathtt{fn}(Q))) \\ &= \mathtt{domCh}(\Gamma') \cap \mathtt{fn}(R) = \mathtt{domCh}(\Pi')\end{aligned}$$

where Π is a specialization of Γ_2 and Π' is a specialization of Γ'_2. □

The previous lemma tells us that there is at most one specialization of the extended environments in the definition of $Ch(\Gamma, \Theta, P|\, Q)$ that can give a non-failure result. The next lemma tells us the effect that renaming with fresh names has on the result of Ch.

Lemma 4.2 (Renaming Lemma) Let $b_1, \ldots, b_m$ be distinct expression names with $b_i \notin \mathtt{fn}(\Gamma) \cup \mathtt{fn}(P), 1 \leq i \leq m$. Let k' be a channel name with $k' \notin \mathtt{dom}(\Gamma) \cup \mathtt{fn}(P)$. Let $Y_1, \ldots, Y_n = \vec{Y}$ be distinct process names such that $Y_j \notin \mathtt{dom}(\Theta) \cup \mathtt{fn}(P), 1 \leq j \leq m$. Then, for any m distinct expression names $a_1, \ldots, a_m$, any channel name k, and any distinct process names

$X_1, \ldots, X_n = \vec{X}$, we have that

(1) $ChEnv(\Gamma\{\vec{a} \leftarrow \vec{b}\}, \Theta\{\vec{a} \leftarrow \vec{b}\}) = ChEnv(\Gamma\{k \leftarrow k'\}, \Theta)$
$= ChEnv(\Gamma, \Theta\{\vec{X} \leftarrow \vec{Y}\}) = ChEnv(\Gamma, \Theta)$,

(2) $ChTy(\Gamma\{\vec{a} \leftarrow \vec{b}\}, \Theta\{\vec{a} \leftarrow \vec{b}\}, v\{\vec{a} \leftarrow \vec{b}\}, T\{\vec{a} \leftarrow \vec{b}\})$
$= ChTy(\Gamma\{k \leftarrow k'\}, \Theta, v, T)$
$= ChTy(\Gamma, \Theta\{\vec{X} \leftarrow \vec{Y}\}, v, T) = ChTy(\Gamma, \Theta, v, T)$,

(3) $ChVec(\Gamma\{\vec{a} \leftarrow \vec{b}\}, \Theta\{\vec{a} \leftarrow \vec{b}\}, v\{\vec{a} \leftarrow \vec{b}\}, \overrightarrow{c:T}\ \{\vec{a} \leftarrow \vec{b}\})$
$= ChVec(\Gamma\{k \leftarrow k'\}, \Theta, \vec{v}, \overrightarrow{a:T}) = ChVec(\Gamma, \Theta\{\vec{X} \leftarrow \vec{Y}\}, \vec{v}, \overrightarrow{c:T})$
$= ChVec(\Gamma, \Theta, \vec{v}, \overrightarrow{a:T})$

(4) $Ch(\Gamma, \Theta\{\vec{X} \leftarrow \vec{Y}\}, P\{\vec{X} \leftarrow \vec{Y}\}) = Ch(\Gamma\{k \leftarrow k'\}, \Theta, P\{k \leftarrow k'\})$
$= Ch(\Gamma, \Theta, P)$.

(5) $Ch(\Gamma, \Theta, P) = \textsf{fail}$ if and only if $Ch(\Gamma\{a \leftarrow b\}, \Theta\{a \leftarrow b\}, P\{a \leftarrow b\}) = \textsf{fail}$, and if $Ch(\Gamma, \Theta, P) \neq \textsf{fail}$, then $Ch(\Gamma\{a \leftarrow b\}, \Theta\{a \leftarrow b\}, P\{a \leftarrow b\}) = Ch(\Gamma, \Theta, P)\{a \leftarrow b\}$

Proof: The proof of the Renaming Lemma is by induction on the definition of $ChEnv$, $ChTy$, $ChVec$, and Ch. It is long, and fairly unnoteworthy. It is omitted here. □

Proposition 4.3 (Well-definedness of Ch) Ch is a total function.

Proof: By Lemmas 4.1, 4.2, and 4.4, Ch defines a function. Furthermore, Ch is a total function because the size of the third argument (process P) decreases in every recursive call. □

The following lemmas let us know that the existence of certain judgments assures us of other related judgments.

Lemma 4.4 If $k \notin \texttt{fn}(P)$ but $k \in \texttt{dom}(\Gamma)$ and $\Gamma \vdash_\Theta P : e$, then $\Gamma(k) = 1$ or there exists an α such that $\Gamma(k) = \bot_{\{\alpha,\overline{\alpha}\}}$.

Lemma 4.5 If $\Gamma \vdash_\Theta P : e$ and $k \notin \texttt{fn}(P)$, then $\Gamma \setminus k \vdash_\Theta P : e$.

Lemma 4.6 If $\Gamma \vdash_\Theta P : e$ and $k \notin \texttt{dom}(\Gamma)$, then $\Gamma \cdot k : 1 \vdash_\Theta P : e$ and $\Gamma \cdot k : \bot_{\{\alpha,\overline{\alpha}\}} \vdash_\Theta P : e$ for all channel types α.

The corresponding facts for Ch are below.

Lemma 4.7 If $Ch(\Gamma, \Theta, P) \neq \textsf{fail}$, then $\texttt{fn}(Ch(\Gamma, \Theta, P)) \subseteq \texttt{fn}(\Gamma)$.

Lemma 4.8 If $k \notin \texttt{fn}(P)$ but $k \in \texttt{dom}(\Gamma)$ and $Ch(\Gamma, \Theta, P) \neq \textsf{fail}$, then $\Gamma(k) = 1$ or there exists an α such that $\Gamma(k) = \bot_{\{\alpha,\overline{\alpha}\}}$.

Lemma 4.9 If $k \notin \mathtt{fn}(P)$, then $Ch(\Gamma \setminus k, \Theta, P) = Ch(\Gamma, \Theta, P)$.

Lemma 4.10 If $k \notin \mathtt{fn}(\Gamma) \cup \mathtt{fn}(P)$, then $Ch(\Gamma \cdot k : 1, \Theta, P) = Ch(\Gamma, \Theta, P)$ and $Ch(\Gamma \cdot k : \bot_{\{\alpha,\overline{\alpha}\}}, \Theta, P) = Ch(\Gamma, \Theta, P)$ for all channel types α.

Lemma 4.11 Let Γ_1 and Γ_2 be environments such that $\Gamma_1 \asymp \Gamma 2$, and let $\Gamma = \Gamma_1 \circ \Gamma_2$. Suppose that $Ch(\Gamma_1, \Theta, P) \neq \textsf{fail}$ and $Ch(\Gamma_2, \Theta, q) \neq \textsf{fail}$ and $split(\Gamma, P, Q) \neq \textsf{fail}$ for some processes P and Q. Let $(\Pi_1, \Pi_2) = split(\Gamma, P, Q)$. Then there exist $\Gamma_1' \in \Sigma(\Gamma_1)$ and $\Gamma_2' \in \Sigma(\Gamma_2)$ such that $\Gamma_1' \asymp \Gamma 2'$ and $\Gamma = \Gamma_1' \circ \Gamma_2'$ and $Ch(\Gamma_1', \Theta, P) = Ch(\Gamma_1, \Theta, P)$ and $Ch(\Gamma_2', \Theta, Q) = Ch(\Gamma_2, \Theta, Q)$.

Proof: Notice that because $\Gamma_1 \circ \Gamma_2 = \Gamma$ we have that $\mathtt{domPl}(\Gamma_1) = \mathtt{domPl}(\Gamma_2) = \mathtt{domPl}(\Gamma) = \mathtt{domPl}(\Pi_1) = \mathtt{domPl}(\Pi_2)$ and $\Gamma_1(a) = \Gamma_2(a) = \Gamma(a) = \Pi_1(a) = \Pi_2(a)$, and we have $\mathtt{dom}(\Gamma_1) \cap \mathtt{dom}(\Gamma_2) \subseteq \mathtt{domChoice}(\Pi_1) = \mathtt{domChoice}(\Pi_2)$, and for all $k \in \mathtt{domChoice}(\Pi_1)$ we have that $\Gamma_i(k) \in \{\alpha, \overline{\alpha}\}$ for $i = 1, 2$. Let

$$
\begin{aligned}
Bad(\Gamma_i) = \; & (\mathtt{domCh}(\Gamma_i) \setminus \mathtt{domCh}(\Pi_i)) \cup (\mathtt{domCh}(\Pi_i) \setminus \mathtt{domCh}(\Gamma_i)) \\
& \cup \{x \in \mathtt{dom}(\Gamma_i) \setminus \mathtt{domChoice}(\Pi_i) | \Gamma_1(k) \neq \Pi_i(k)\}
\end{aligned}
$$

Notice that $\Gamma_i \in \Sigma(\Pi_i)$ if and only if $Bad\Gamma_i = \emptyset$. The proof is by induction on the combined length of $Bad(\Gamma_1) \cup Bad(\Gamma_2)$. Observe that $\mathtt{domCh}(\Pi_1) = \mathtt{domCh}(\Gamma) \setminus (\mathtt{fn}(Q) \setminus \mathtt{fn}(P))$ and $\mathtt{domCh}(\Pi_2) = \mathtt{domCh}(\Gamma) \cap \mathtt{fn}(Q)$.

Suppose there exists $k \in \mathtt{domCh}(\Gamma_1) \setminus \mathtt{domCh}(\Pi_1)$. Then $k \in \mathtt{fn}(Q) \setminus \mathtt{fn}(P)$. Since $k \notin \mathtt{fn}(P)$, by Lemma 4.9, $Ch(\Gamma_1 \setminus k, \Theta, P) = Ch(\Gamma_1, \Theta, P)$. Since $k \in \mathtt{fn}(Q) \subseteq \mathtt{domCh}(\Gamma_2)$ and $k \in \mathtt{domCh}(\Gamma_1)$ and $\Gamma_1 \asymp \Gamma_2$, we must have that $\Gamma_2(k) = \overline{\Gamma_1(k)}$. But by Lemma 4.8, either $\Gamma_1(P) = 1$ or $\Gamma_1(P) = \bot_{\{\alpha,\overline{\alpha}\}}$ for some α. However, $\overline{\bot_{\{\alpha,\overline{\alpha}\}}}$ does not exist. Therefore, $\Gamma_1(k) = 1 = \Gamma_2(k)$, and $\Gamma(k) = 1$. Then $(\Gamma_1 \setminus k) \asymp \Gamma_2$ and $(\Gamma_1 \setminus k) \circ \Gamma_2 = \Gamma$. Moreover, $|Bad(\Gamma_1 \setminus k) \cup Bad(\Gamma_2)| < |Bad(\Gamma_1) \cup Bad(\Gamma_2)|$, so by induction we there exist $\Gamma_1' \in \Sigma(\Pi_1)$ and $\Gamma_2' \in \Sigma(\Pi_2)$ such that $\Gamma_1' \asymp \Gamma_2'$ and $\Gamma = \Gamma_1' \circ \Gamma_2'$ and $Ch(\Gamma_1', \Theta, P) = Ch(\Gamma_1, \Theta, P)$ and $Ch(\Gamma_1', \Theta, P) = Ch(\Gamma_1 \setminus k, \Theta, P)$ and $Ch(\Gamma_2', \Theta, Q) = Ch(\Gamma_2, \Theta, Q)$. Since $Ch(\Gamma_1 \setminus k, \Theta, P) = Ch(\Gamma_1, \Theta, P)$, we are done in this case.

Suppose there exists $k \in \mathtt{domCh}(\Gamma_2) \setminus \mathtt{domCh}(\Pi_2)$. Then $k \notin \mathtt{fn}(Q)$. By Lemma 4.8, $\Gamma_2(k) \in \{1, \bot_{\{\alpha,\overline{\alpha}\}}\}$. If $k \in \mathtt{domCh}(\Gamma_1)$, then the argument proceeds as before. Therefore, we only need to consider $k \notin \mathtt{domCh}(\Gamma_1) \cup \mathtt{fn}(Q)$. Since $k \notin \mathtt{domCh}(\Gamma_1)$ and $\Gamma = \Gamma_1 \circ \Gamma_2$, we have the $\Gamma(k) = \Gamma_2(k)$. Then by Lemmas 4.10 and 4.9, $Ch(\Gamma_1 \cdot \Gamma(k), \Theta, P) = Ch(\Gamma_1, \Theta, P)$ and $Ch(\Gamma_2 \setminus k, \Theta, Q) = Ch(\Gamma_2, \Theta, Q)$. Moreover, $(\Gamma_1 \cdot \Gamma(k)) \asymp (\Gamma_2 \setminus k)$ and $\Gamma = (\Gamma_1 \cdot \Gamma(k)) \circ (\Gamma_2 \setminus k)$. Therefore, by induction, as above we are done in this case.

The case of $k \in \mathtt{domCh}(\Pi_1) \setminus \mathtt{domCh}(\Gamma_1)$ is basically the same as the previous case. The case of $k \in \mathtt{domCh}(\Pi_2) \setminus \mathtt{domCh}(\Gamma_2)$ can't happen because

$\mathtt{domCh}(\Pi_2) \subseteq \mathtt{fn}(Q)\mathtt{dom}(\Gamma_2)$. If $k \in \mathtt{domCh}(\Gamma_i) \setminus \mathtt{domChoice}(\Pi_i)$, then $\Gamma(k) \neq \bot_{\{\alpha,\overline{\alpha}\}}$ and $k \notin \mathtt{domCh}(\Gamma_1) \cap \mathtt{domCh}(\Gamma_2)$ and $\Gamma_i(k) = \Gamma(k)$. If $k \in \mathtt{fn}(P) \cup \mathtt{fn}(Q)$, then $\Gamma_i(k) = \Pi_i(k)$. If $k \notin \mathtt{fn}(P) \cup \mathtt{fn}(Q)$, then $\Gamma_i(k) = 1$. Moreover, since $\Gamma(k) = 1$ and $\Pi_i(k) \neq 1$, $k \notin \mathtt{domCh}(\Pi_i)$, and we must have $i = 2$. Then, as before we use Lemmas 4.10 and 4.9 to move k from $\mathtt{domCh}(\Gamma_2)$ to $\mathtt{domCh}(\Gamma_1)$.
□

We now show that if the a process P is typable in a given environment Γ and definitions Θ, then the algorithm does not fail (Completeness Proposition 4.12); we also show that if the algorithm does not fail with inputs Γ,Θ, and P, then P has effect returned by the algorithm under Γ and Θ (Soundness Proposition 4.13). Finally, we show that the effect found by the algorithm is the least effect (Minimum Effects 4.14).

Proposition 4.12 (Completeness) If $\Gamma \vdash_\Theta P : e$, then $Ch(\Gamma, \Theta, P) \neq \textsf{fail}$ and $Ch(\Gamma, \Theta, P) \leq e$.

Proof: The proof follows by induction on the derivation of $\Gamma \vdash_\Theta P : e$. We consider a case analysis on the last rule applied in the derivation. We give a representative sample here.

- Type Acpt: If the last rule applied in the derivation of $\Gamma \vdash_\Theta P : e$ is Type Acpt, then $P = \mathtt{accept}\ a(k)\ \mathtt{in}\ Q$ and $\Gamma(a) = \sigma(\alpha)$ for some expression name a, process Q, and channel type α. Moreover, there exists $k' \notin \mathtt{dom}(\Gamma)$ such that $\Gamma \cdot k' : \alpha \vdash_\Theta Q\{k \leftarrow k'\} : e$ by a smaller derivation. Therefore, by induction we have that $Ch(\Gamma \cdot k' : \alpha, \Theta, Q) \neq \textsf{fail}$ and $Ch(\Gamma \cdot k' : \alpha, \Theta, Q) \leq e$. Since $Ch(\Gamma \cdot k' : \alpha, \Theta, Q) \neq \textsf{fail}$ we have that $\mathtt{fn}(Q) \subseteq \mathtt{dom}(\Gamma) \cup \{k'\}$. Since $\Gamma(a) = \sigma(\alpha)$, by the definition of Ch, we have that $Ch(\Gamma, \Theta, \mathtt{accept}\ a(k)\ \mathtt{in}\ Q) = Ch(\Gamma \cdot k'' : \alpha, \Theta, Q\{k \leftarrow k''\})$ for some $k'' \notin \mathtt{dom}(\Gamma)$. If $k'' = k'$, we are done with this case, so suppose $k'' \neq k'$. By Lemma 2.4, since $\Gamma \vdash_\Theta P : e$, we have that Γ is well-formed, and hence $\mathtt{fn}(\Gamma) \subseteq \mathtt{dom}(\Gamma)$. Therefore, $k'' \notin \mathtt{fn}(\Gamma)$ and $k'' \notin \mathtt{fn}(Q) \subseteq \mathtt{dom}(\Gamma)$, and hence $\Gamma\{k'' \leftarrow k'\} = \Gamma$ and $Q\{k'' \leftarrow k'\} = Q$. Thus we have

  $$\begin{aligned}
  &Ch(\Gamma, \Theta, \mathtt{accept}\ a(k)\ \mathtt{in}\ Q)\\
  &= Ch(\Gamma \cdot k'' : \alpha, \Theta, Q\{k \leftarrow k''\})\\
  &= Ch((\Gamma \cdot k'' : \alpha)\{k'' \leftarrow k'\}, \Theta, Q\{k \leftarrow k''\}\{k'' \leftarrow k'\})\\
  &= Ch(\Gamma \cdot k' : \alpha, \Theta, Q\{k \leftarrow k'\}) \leq e.
  \end{aligned}$$

  This finishes the case of Type Acpt. The case of Type Requ is completely analogous.
- Type Bgn and Type End: Suppose $P = \mathtt{begin}\ L; Q$ or $P = \mathtt{end}\ L; Q$. In either case, we have that $\Gamma \vdash_\Theta Q : e'$, where $e = e' \setminus (\!| L |\!)$ in the $\mathtt{begin}$ case, and $e = e' + (\!| L |\!)$ in the $\mathtt{begin}$ case. By induction we have that $Ch(\Gamma, \Theta, Q) \neq \textsf{fail}$ and $Ch(\Gamma, \Theta, Q) \leq e$. Therefore, in each

case $Ch(\Gamma,\Theta,\texttt{begin }L;Q) = Ch(\Gamma,\Theta,Q) \setminus (\!| L |\!) \leq e' \setminus (\!| L |\!) = e$, or $Ch(\Gamma,\Theta,\texttt{end }L;Q) = Ch(\Gamma,\Theta,Q) + (\!| L |\!) \leq e' + (\!| L |\!) = e$,

- Type Snd: If Type Snd is the last rule applied, then $P = k![v]Q$ for some k, v and Q, and $\Gamma(k) = \uparrow [a:T]e';\alpha$ for some a, T, e', and α with $e'\{a \leftarrow v\} \leq e$, and $\Gamma \setminus k \vdash_\Theta v : T$ and $\texttt{fn}(e') \setminus \{a\} \subseteq \texttt{dom}(\Gamma)$ and $(\Gamma \setminus k) \cdot k : \alpha\{a \leftarrow v\} \vdash_\Theta Q : e \setminus e'$. Therefore, by induction we have that $Ch((\Gamma \setminus k) \cdot k : \alpha\{a \leftarrow v\}, \Theta, Q) \neq$ **fail** and $Ch((\Gamma \setminus k) \cdot k : \alpha\{a \leftarrow v\}, \Theta, Q) \leq e \setminus e'\{a \leftarrow v\}$, and by the definition of Ch, we have $Ch(\Gamma,\Theta,k![v]Q) \neq$ **fail** and

$$\begin{aligned} Ch(\Gamma,\Theta,k![v]Q) &= Ch((\Gamma \setminus k) \cdot k : \alpha\{a \leftarrow v\}, \Theta, Q) + e'\{a \leftarrow v\} \\ &\leq (e \setminus e'\{a \leftarrow v\}) + e'\{a \leftarrow v\} \\ &= e \qquad \text{because } e'\{a \leftarrow v\} \leq e. \end{aligned}$$

- Type Rcv: In this case we must have $P = k?(b)\ \texttt{in}\ Q$ for some k, b and Q, and $\Gamma(k) = \downarrow [a:T]e'$ for some a, T and e' with $\texttt{fn}(e') \setminus \{a\} \subseteq \texttt{dom}(\Gamma)$, and $(\Gamma \setminus k) \cdot c : T \cdot k : \alpha\{a \leftarrow c\} \vdash_\Theta Q\{b \leftarrow c\} : e''$ for some c and e'' such that $c \notin \texttt{fn}(e'' \setminus e'\{a \leftarrow c\}) \cup \texttt{fn}(\Gamma)$ and such that $e = e'' \setminus e'\{a \leftarrow c\}$. By induction, we have $Ch((\Gamma \setminus k) \cdot c : T \cdot k : \alpha\{a \leftarrow c\}, \Theta, Q\{b \leftarrow c\}) \neq$ **fail** and $Ch((\Gamma \setminus k) \cdot c : T \cdot k : \alpha\{a \leftarrow c\}, \Theta, Q\{b \leftarrow c\}) \leq \mathrm{e}''$ Since $Ch((\Gamma \setminus k) \cdot c : T \cdot k : \alpha\{a \leftarrow c\}, \Theta, Q\{b \leftarrow c\}) \neq$ **fail**, by Lemma 4.2, we have that

$$\begin{aligned} &Ch((\Gamma \setminus k) \cdot c' : T \cdot k : \alpha\{a \leftarrow c\}', \Theta, Q\{b \leftarrow c\}') \\ &= Ch((\Gamma \setminus k) \cdot c : T \cdot k : \alpha\{a \leftarrow c\}\{c \leftarrow c'\}, \Theta\{c \leftarrow c'\}, Q\{b \leftarrow c\}\{c \leftarrow c'\}) \\ &= (Ch(\Gamma,\Theta,Q\{b \leftarrow c\}))\{c \leftarrow c'\} \leq e''\{c \leftarrow c''\} \end{aligned}$$

for any $c' \notin \texttt{fn}(\Gamma)$. Also, since $c \notin \texttt{fn}(e'' \setminus e'\{a \leftarrow c\}) \cup \texttt{fn}(\Gamma)$, we have $c'' \notin \texttt{fn}(e''\{c \leftarrow c'\} \setminus e'\{a \leftarrow c\}') \cup \texttt{fn}(\Gamma)$ and hence $c' \notin Ch((\Gamma \setminus k) \cdot c' : T \cdot k : \alpha\{a \leftarrow c\}', \Theta, Q\{b \leftarrow c\}') \setminus e'\{a \leftarrow c\}'$. Therefore, by the definition of Ch, $Ch(\Gamma,\Theta,k?(b)\ \texttt{in}\ Q) \neq$ **fail** and

$$\begin{aligned} &Ch(\Gamma,\Theta,k?(b)\ \texttt{in}\ Q) \\ &= Ch((\Gamma \setminus k) \cdot c' : T \cdot k : \alpha\{a \leftarrow c'\}, \Theta, Q\{b \leftarrow c\}') \setminus e'\{a \leftarrow c\}' \\ &\leq e''\{c \leftarrow c'\} \setminus e'\{a \leftarrow c'\} \\ &\leq (e'' \setminus e'\{a \leftarrow c\})\{c \leftarrow c'\} \ = e'' \setminus e'\{a \leftarrow c\} \quad \text{because } c \notin \texttt{fn}(e'' \setminus e'\{a \leftarrow c\}) \\ &= e \end{aligned}$$

- Type Par: If Type Par was the last rule to be applied, then $P = Q|\,R$ and there exist environments Γ_1 and Γ_2 and effects e_1 and e_2 such that $\Gamma_1 \vdash_\Theta Q : e_1$ and $\Gamma_2 \vdash_\Theta R : e_2$ and $\Gamma_1 \asymp \Gamma_2$ and $\Gamma = \Gamma_1 \circ \Gamma_2$ and $e = e_1 + e_2$. By the inductive hypothesis, we have that $Ch(\Gamma_1,\Theta,Q) \neq$ **fail**, $Ch(\Gamma_1,\Theta,Q) \leq \mathrm{e}_1$, $Ch(\Gamma_2,\Theta,R) \neq$ **fail**, and $Ch(\Gamma_2,\Theta,R) \leq \mathrm{e}_2$.

Since $Ch(\Gamma_1, \Theta, Q) \neq \textsf{fail}$ and $Ch(\Gamma_2, \Theta, R) \neq \textsf{fail}$, we have $\mathtt{fn}(Q) \subseteq \mathtt{dom}(\Gamma_1)$ and $\mathtt{fn}(R) \subseteq \mathtt{dom}(\Gamma_2)$. Since $\Gamma = \Gamma_1 \circ \Gamma_2$, we have that $\mathtt{fn}(Q) \cup \mathtt{fn}(R) \subseteq \mathtt{dom}(\Gamma)$ and for each $k \in \mathtt{fn}(Q) \cap \mathtt{fn}(R)$ there exists an α such that $\Gamma(k) = \bot_{\{\alpha,\overline{\alpha}\}}$. Therefore $split(\Gamma, Q, R) \neq \textsf{fail}$. Let $(\Pi_1, \Pi_2) = split(\Gamma, Q, R)$. By Lemma 4.11 we have there exist $\Gamma'_1 \in \Sigma(\Pi_1)$ and $\Gamma'_2 \in \Sigma(\Pi_2)$ such that $\Gamma'_1 \asymp \Gamma'_2$ and $Ch(\Gamma'_1, \Theta, Q) = Ch(\Gamma_1, \Theta, Q)$ and $Ch(\Gamma'_2, \Theta, R) = Ch(\Gamma_2, \Theta, R)$. Since $\Gamma'_1 \asymp \Gamma'_2$, we have that $\Gamma'_1(k) = \overline{\Gamma'_2(k)}$ for all $k \in \mathtt{domChoice}(\Gamma_1)$. Therefore, by the definition of Ch we have that $Ch(\Gamma, \Theta, Q|\, R) \neq \textsf{fail}$ and

$$\begin{aligned} Ch(\Gamma, \Theta, Q|\, R) &= Ch(\Gamma'_1, \Theta, Q) + Ch(\Gamma'_2, \Theta, R) \\ &= Ch(\Gamma_1, \Theta, Q) + Ch(\Gamma_2, \Theta, R) \\ &\leq e_1 + e_2 = e \end{aligned}$$

□

Proposition 4.13 (Soundness) If $Ch(\Gamma, \Theta, P) \neq \textsf{fail}$, then $\Gamma \vdash_\Theta P : Ch(\Gamma, \Theta, P)$.

Proof: The proof follows by induction on the definition of $Ch(\Gamma, \Theta, P)$. We show here a few representative cases.

- $\mathtt{accept}\ a(k)\ \mathtt{in}\ P$: By the definition of Ch, $Ch(\Gamma, \Theta, \mathtt{accept}\ a(k)\ \mathtt{in}\ P) = Ch(\Gamma \cdot k' : \alpha, \Theta, P\{k \leftarrow k'\})$, where $k' \notin \mathtt{dom}(\Gamma)$ and $\Gamma(a) = \sigma(\alpha)$. By the induction hypothesis, $\Gamma \cdot k' : \alpha \vdash_\Theta P\{k \leftarrow k'\} : Ch(\Gamma \cdot k' : \alpha, \Theta, P\{k \leftarrow k'\})$. By the definition of Ch, and applying Type Acpt, $\Gamma \cdot a : \sigma(\alpha) \vdash_\Theta \mathtt{accept}\ a(k)\ \mathtt{in}\ P : Ch(\Gamma, \Theta, \mathtt{accept}\ a(k)\ \mathtt{in}\ P)$.
- $\mathtt{begin}\ L; P$: By the induction hypothesis, $\Gamma \vdash_\Theta P : Ch(\Gamma, \Theta, P)$. Since, $\mathtt{fn}(P) \in \mathtt{dom}(\Gamma)$, it follows that $\mathtt{fn}(L) \in \mathtt{dom}(\Gamma)$. By the rule Type Bgn, $\Gamma \vdash_\Theta \mathtt{begin}\ L; P : Ch(\Gamma, \Theta, P) \setminus (\!| L |\!)$, and by the definition of Ch, $\Gamma \vdash_\Theta \mathtt{begin}\ L; P : Ch(\Gamma, \Theta, \mathtt{begin}\ L; P)$.
- $P|\, Q$: By the definition of Ch, $Ch(\Gamma, \Theta, P|\, Q) = Ch(\Gamma'_1, \Theta, P) + Ch(\Gamma'_2, \Theta, Q)$, for some Γ'_1 and Γ'_2. By construction of Γ'_1 and Γ'_2, it follows that $\Gamma'_1 \asymp \Gamma'_2$ and $\Gamma = \Gamma'_1 \circ \Gamma'_2$, and by the induction hypothesis, $\Gamma'_1 \vdash_\Theta P : Ch(\Gamma'_1, \Theta, P)$ and $\Gamma'_2 \vdash_\Theta Q : Ch(\Gamma'_2, \Theta, Q)$. Finally, by the rule Type Par, the result follows. □

Corollary 4.14 (Minimum Effects) If $\Gamma \vdash_\Theta P : e$, then $\Gamma \vdash_\Theta P : Ch(\Gamma, \Theta, P)$ and $Ch(\Gamma, \Theta, P) \leq e$.

Proof: The result holds immediately from Soundness (Proposition 4.13) and Completeness (Proposition 4.12). □

We can now state our main result.

Corollary 4.15 (Decidability of Typechecking) Given Γ, Θ, P and e it is decidable whether $\Gamma \vdash_\Theta P : e$.

Proof: We first call $Ch(\Gamma, \Theta, P)$ that always terminates, by Proposition 4.3. If $Ch(\Gamma, \Theta, P) = \textsf{fail}$, by Completeness (Proposition 4.12), $\Gamma \vdash_\Theta P : e$ is not derivable. If $Ch(\Gamma, \Theta, P) \neq \textsf{fail}$, we check the multiset inclusion $Ch(\Gamma, \Theta, P) \leq e$ which is also decidable. If $Ch(\Gamma, \Theta, P) \leq e$ holds, then by Soundness (Proposition 4.13) and Type Subsum, $\Gamma \vdash_\Theta P : e$. If $Ch(\Gamma, \Theta, P) \not\leq e$, by Completeness (Proposition 4.12), $\Gamma \vdash_\Theta P : e$ is not derivable. □

5 Conclusions and Future Work

A session type describes the interactions between two parties within multiparty communications. It is a communication protocol describing the order and type of interactions between two parties. Iris is a typed π-calculus resulting from a combination of session types with correspondence assertions that takes session types a step further. Indeed, not only does Iris allow the description of the exchange protocol, but also the synchronization between parties that may not participate in the same session.

This paper studies the typechecking problem for Iris. We define a typechecking algorithm $Ch(\Gamma, \Theta, P)$ that checks whether process P is typable under the typing assumptions in Γ. If P is typable under Γ, it returns the least effect for P, and otherwise it returns **fail**. Although session types have been extensively studied in the past few years, to our knowledge this is the first proof of decidability of typechecking for a type system with session types. A related open problem that we are currently investigating is the decidability of *type inference*, where type unification has to be considered in the presence of equations such as those defining the dual of a channel type.

Iris allows us to express the relationship between the information being sent at its origin and the information being received at the intended destination. If we stay within a decidable fragment (such as linear arithmetic) we can capture a considerable family of communication and data exchange patterns: in a large percentage of the cases where data is transferred, we are interested in seeing the exact same data at both ends, and many other cases involve very simple linear arithmetic transformations. For example, frequently an ATM is allowed to charge a processing fee for a transaction, and then the relation between the amount entered by the Client and that received by the Bank will not be identical, but will satisfy a simple linear arithmetic equation. To address this issue we are considering the extension of Iris with arithmetic.

If we allow general arithmetic, which is not decidable, we can expect to define

a sound semi-decision procedure: An algorithm without false positives or false negatives. If the algorithm says yes, then all information can be traced back to its sources. If the algorithm says no, the algorithm will exhibit a path showing that the data is not coming from the intended origin. If the algorithm fails to terminate, then we cannot deduce any information.

Future work also 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: We are grateful to the Laboratory for Secure Systems group at Stevens for interesting discussions, and in particular to Georgi Babayan, Pablo Garralda, and Ricardo Medel for inspiring brainstorming sessions. We also thank Healfdene Goguen for comments and suggestions on previous drafts. This work was partially supported by The Stevens Technogenesis Fund, the NSF Grant No. CCR-0220286 ITR: Secure Electronic Transactions, and the ARO Award No. DAAD-19-01-1-0473.

References

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

Bonelli, Eduardo, Compagnoni, Adriana, & Gunter, Elsa. (2003a). Correspondence assertions for process synchronization in concurrent communications. Brogi, Antonio, Jacquet, Jean-Marie, & Pimentel, Ernesto (eds), *FOCLASA 2003. 2nd International Workshop on Foundations of Coordination Languages and Software Architectures*. Electronic Notes in Theoretical Computer Science, vol. 91, no. 2. Marseille, France: Elsevier Science. To appear.

Bonelli, Eduardo, Compagnoni, Adriana, & Gunter, Elsa. (2003b). *Correspondence assertions for process synchronization in concurrent communications*. Tech. rept. 2003-7. Department of Computer Science, Stevens Institute of Technology.

Cervesato, Iliano, & Pfenning, Frank. (2002). A linear logical framework. *Information and Computation*, **179**(1), 19–75.

Chaki, Sagar, Rajamani, Sriram, & Rehof, Jakob. (2002). Types as models: Model checking message-passing programs. *Pages 45–57 of: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM.

Gay, Simon. (1993). A sort inference algorithm for the polyadic pi-calculus. *Proc. of the 20th ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*. ACM Press.

Gay, Simon, & Hole, Malcolm. (1999). Types and subtypes for client-server interactions. *Pages 74–90 of: Proceedings of the European Symposium on Programming Languages and Systems*. LNCS, no. 1576. Springer-Verlag.

Gay, Simon, Vasconcelos, Vasco, & Ravara, Antonio. (2003). *Session types for inter-process communication*. Tech. rept. TR-2003-133. Department of Computing Science, University of Glasgow.

Girard, Jean-Yves. (1987). Linear Logic. *Theoretical Computer Science*, 1–102.

Gordon, Andrew, & Jeffrey, Alan. (2001a). Authenticity by typing for security protocols. *Pages 145–159 of: 14th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press.

Gordon, Andrew, & Jeffrey, Alan. (2001b). Typing correspondence assertions for communication protocols. *Seventeenth Conference on the Mathematical Foundations of Programming Semantics (MFPS 2001)*. ENTCS, no. 45. Elsevier.

Gordon, Andrew, & Jeffrey, Alan. (2003a). Authenticity by typing for security protocols. *Journal of computer security*, **11**(4), 451–521.

Gordon, Andrew, & Jeffrey, Alan. (2003b). Typing correspondence assertions for communication protocols. *Theoretical Computer Science*, **300**, 379–409.

Gordon, Michael J.C., & Melham, Thomas F. (1993). *Introduction to HOL: A theorem proving environment for higher-order logic*. Cambridge: CUP.

Hole, Malcolm, & Gay, Simon. (2003). *Bounded polymorphism in session types*. Tech. rept. TR-2003-132. Department of Computing Science, University of Glasgow.

Honda, Kohei, Kubo, Makoto, & Takeuchi, Kaku. (1994). An interaction-based language and its typing system. *Pages 398–413 of: Proceedings of PARLE'94*. LNCS, no. 817. Springer-Verlag.

Honda, Kohei, Vasconcelos, Vasco, & Kubo, Makoto. (1998). Language primitives and type discipline for structured communication-based programming. *Pages 122–138 of: Proceedings of ESOP'98*. LNCS. Springer-Verlag.

Igarashi, Atsushi, & Kobayashi, Naoki. (2001). A generic type system for the pi-calculus. *Pages pp.128–141 of: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM.

Igarashi, Atsushi, & Kobayashi, Naoki. (2004). A generic type system for the pi-calculus. *Theoretical Computer Science*, **311**(1–3), 121–163.

Kobayashi, Naoki. (1997). A partially deadlock-free type process calculus. *Pages 128–139 of: Proceedings of the Twelth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press.

Kobayashi, Naoki, Pierce, Benjamin, & Turner, David N. (1996). Linearity in the pi-calculus. *Pages 358–371 of: Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*.

Milner, Robin. (1999). *Communicating and mobile systems: the π-calculus*. Cambridge University Press.

Pierce, Benjamin, & Sangiorgi, Davide. (1993). Typing and subtyping for mobile processes. *Pages 376–385 of: Proceedings of the eighth annual ieee symposium on logic in computer science*. IEEE Computer Society Press.

Puntigam, Franz. (1996). Synchronization expressed in the types of communication channels. *Pages 762–769 of: Proceedings of the EURO-PAR'96*. LNCS, no. 1123. Springer-Verlag.

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

Vallecillo, Antonio, Vasconcelos, Vasco, & Ravara, António. (2003). Typing the

behavior of objects and component using session types. *Electronic Notes in Theoretical Computer Science*, **68**(3).

Yoshida, Nobuko. (1996). Graph types for monadic mobile processes. *Pages 371–386 of: FST/TCS'16.* LNCS, no. 1180. Springer-Verlag.