ELSEVIER

Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 138 (2005) 3–22

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Typechecking Safe Process Synchronization

Eduardo Bonelli[1,3]

Stevens Institute of Technology and LIFIA [6]

Adriana Compagnoni[1,4]

Stevens Institute of Technology

Elsa Gunter[1,2,5]

University of Illinois at Urbana - Champaign

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, type systems, typechecking.

[1] Supported in part by NSF Grant No. CCR-0220286 ITR:Secure Electronic Transactions.
[2] Supported in part by the ARO under Award No. DAAD-19-01-1-0473.
[3] Email:ebonelli@cs.stevens-tech.edu
[4] Email:abc@cs.stevens-tech.edu
[5] Email:egunter@cs.uiuc.edu
[6] Faculty of Informatics, University of La Plata, La Plata, Argentina

doi:10.1016/j.entcs.2005.05.002

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. It is of great importance to have high assurance that the software governing these communications and resulting decisions is correct.

The approach presented in this work applies to any situation where there is communication between multiple parties that can be factored into one-to-one communications. Session types [17,18] allow one 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, what party receives it, and the type of the information. However, session types alone fall short in restricting process interaction. For example, a small system involving processes Client, ATM and Bank is developed in [1,4]. It illustrates situations which cannot be captured by session types, including:

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

Combining session types with correspondence assertions [23,12,14] increases considerably the expressiveness of the interaction patterns that may be imposed on processes. Examples of properties that may now be addressed are [1,4]:

- the balance that Client receives always comes from Bank, and the amount to be deposited received by Bank always comes from Client.
- ATM should behave as a forwarder that does not alter the data received from Bank or Client.
- we can detect that ATM is attempting a deposit not instructed by Client or tries to deposit a smaller amount than the one specified by Client.

Iris, a statically typed language based on the π-calculus that extends [18] with correspondence assertions [11,14], was first introduced in [1]. 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 showing that typechecking is decidable.

Related work. Regarding session types, they 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 K. Honda et al. [17]. Natural extensions of this work that have been studied include subtypes [8] and bounded polymorphism [16]. They have also been studied in the context of component-based software development [22] and reformulated in the λ-calculus with input/output operations [9]. Regarding type-checking and inference, in [7] 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 [12,11,14,13] that includes correspondence assertions and public and private data. Regarding the processes as models paradigm introduced by Chaki et al. in [6] 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 [19,20] is discussed in that paper. Recently, a type-inference system called TyPiCal has been developed by N. Kobayashi that permits lock-freedom analysis, deadlock-freedom analysis, useless-code elimination, and information flow analysis [21].

2 The Iris-Calculus

2.1 Syntax

We assume given a set of *names* $x, y, z, \ldots$ We distinguish two distinct kinds of names: *expression names*, written $a, b, c, \ldots$ (which range over sessions and integers); and *channel names*, written $k, h, k', \ldots$ We also have *integer constants* $\ldots, -1, 0, 1, \ldots$ and (branching) *labels* $l, l', \ldots$ 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 [7]

[7] In our technical report [3] we also deal with a process declaration construct `def` D `in` P where D takes the form $X_1[\overline{a_1 : T_1}] = P_1$ `and` $\ldots$ `and` $X_n[\overline{a_n : T_n}] = P_n$ and a process call construct $X[\overline{v}]$. These have been omitted due to space restrictions.

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

Remark 2.1 Parentheses are binding constructs. Two process expressions which differ only in the names of their bound names are called α-equivalent and 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 $P\{k \leftarrow k'\}$. 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 [2] for details).

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 [18], a translation to an asynchronous calculus with branching is possible. Controlled side-stepping of linearity constraints on channel usage is achieved by means of the channel delegation constructs `throw` $k[k'];P$ and `catch` $k(k')$ `in` Q. Mechanisms for selection of a label and branching are available as $k \lhd l;P$ and $k \rhd \{l_1:P_1 \Box \ldots \Box\ l_n:P_n\}$. The notation $P \,|\, Q$ stands for the concurrent execution of P and Q; 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 (Def. 2.2) and $\bot_{\{\alpha,\overline{\alpha}\}}$ is the "complete" channel type with communication protocol given by the channel type α. Note that $\bot_{\{\alpha,\overline{\alpha}\}} = \bot_{\{\overline{\alpha},\alpha\}}$. The `begin` and `end` assertions shall be used as type directives in the type system for Iris (Sect. 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 operational semantics of Iris in the form of a reduction relation on processes is given in Fig. 1. As usual, it relies on the notion of structural congruence whose definition in Iris is standard.

2.2 The Type Discipline

2.2.1 Session types and effects

The type system assigns an effect to a process under a given set of type assumptions. The effect of a process reflects its pending obligations. An

$(\texttt{accept}\ a(k)\ \texttt{in}\ P_1) \mid (\texttt{request}\ a(k)\ \texttt{in}\ P_2) \longrightarrow (\nu k : \bot_{\{\alpha,\overline{\alpha}\}})(P_1 \mid P_2)$	Trans Link
$(k![v]; P_1) \mid (k?(a)\ \texttt{in}\ P_2) \longrightarrow P_1 \mid P_2\{a \leftarrow v\}$	Trans Comm
$(k \lhd l_i; P) \mid (k \rhd \{l_1 : P_1 \Box \ldots \Box l_n : P_n\}) \longrightarrow P \mid P_i,\ \text{if } i \in 1..n$	Trans Brnch
$(\texttt{throw}\ k[k']; P_1) \mid (\texttt{catch}\ k(k'')\ \texttt{in}\ P_2) \longrightarrow P_1 \mid P_2\{k'' \leftarrow k'\}$	Trans Catch
$\texttt{begin}\ L; P \longrightarrow P$	Trans Begin
$\texttt{end}\ L; P \longrightarrow P$	Trans End
$P \longrightarrow P' \Rightarrow (\nu x : U)P \longrightarrow (\nu x : U)P'$	Trans Res
$P \longrightarrow P' \Rightarrow P \mid Q \longrightarrow P' \mid Q$	Trans Par
$P \equiv P', P' \longrightarrow Q', Q' \equiv Q \Rightarrow P \longrightarrow Q$	Trans $\equiv$

Fig. 1. Unlabelled Reduction Semantics for Iris

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

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

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

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

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 $\texttt{fn}(U)$, is defined as usual (see [2]). The base type **Int** is the type of integer constants. Session types are represented as $\sigma(\alpha)$ and may informally be seen to denote a pair consisting of a channel type α and its dual $\overline{\alpha}$:

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

The types α and $\overline{\alpha}$ shall be assigned to the two endpoints of a communication session. Note that $\overline{\bot_{\{\alpha,\overline{\alpha}\}}}$ 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 subtrac-

tion is defined as $e \setminus e'$, the smallest multiset e'' such that $e \leq e' + e''$, where "+" is multiset union. Multiset join is defined as $e \vee e'$, the smallest multiset e'' such that $e \leq e''$ and $e' \leq e''$. The special channel type $\bot_{\{a,\overline{a}\}}$ 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 2.3 [Depends on] $x_i : U_i$ *depends directly on* $x_j : U_j$ *in* Γ (written $(x_j : U_j) \hookrightarrow_d (x_i : U_i)$), if $x_j \in \mathtt{fn}(U_i)$. We say $x_i : U_i$ *depends on* $x_j : U_j$ *in* Γ if $x_i : U_i \hookrightarrow x_j : U_j$, where $\hookrightarrow$ denotes the transitive closure of $\hookrightarrow_d$.

An environment is *well-formed* if it satisfies the following three conditions:

C1. For each $x \in \mathtt{domPl}(\Gamma)$, x is an expression name, and for each $y \in \mathtt{domCh}(\Gamma)$, y is a channel name.

C2. For each $i \in 1..n$, $\mathtt{fn}(U_i) \subseteq \mathtt{dom}(\Gamma) \setminus \{x_i\}$.

C3. The relation $\hookrightarrow$ is irreflexive, that is, $x_i : U_i \not\hookrightarrow x_i : U_i$ for all $x_i : U_i \in \Gamma$.

The first condition, **C1**, requires that only channel types be assigned to channel names, and only plain types be assigned to expression names. Condition **C2** requires that all free names in types assigned by Γ must be declared within Γ. Note that since channel names may not appear in assertion labels, and 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 [10] (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 [5]. 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 [3].

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

$$\Gamma \vdash \diamond \qquad \text{well-formed environment } \Gamma$$

$$\Gamma \vdash v : T \qquad \text{well-typed value } v \text{ of type } T$$

$$\Gamma \vdash P : e \qquad \text{well-typed process } P \text{ with effect } e$$

The typing rules of Iris are presented in Fig. 2. The rules Type Acpt and Type Requ introduce a new channel name in the environment, thus guaranteeing that a private channel is being used for the session. Note that dual channel types are used for the requesting and accepting parties. Type Bgn and Type End affect process effects by eliminating or adding a new assertion label. The rules Type Snd and Type Rcv allow the typing of the communication primitives for sending and receiving data. Note that data is sent and received over channels only. Also, note that the type of k in the upper right-hand judgement of Type Snd is $\alpha\{a \leftarrow v\}$, reflecting the fact that the "rest" of the channel type, namely α, may depend on the output value v. 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[8]. Channel and name restriction (for non-channel names) are typed as expected. Type Stop types the inaction `stop`; it requires all communication through channel names to have been completed. The Type Subs rule allows increasing the required assertion obligations of a process term.

The Type Par rule types the parallel execution of two processes. A channel may be used 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* (Fig. 4). Note that the notion of compatibility makes sense for two sets of assumptions that 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* (Fig. 4). The subscript of $\bot_{\{\alpha,\overline{\alpha}\}}$ in the second clause of the definition of composition of environments (Fig. 4) records the dual channel types from which it arises, and hence the name dependencies of those dual channel types.

3 Typechecking

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

- $ChEnv(\Gamma)$, which checks the well-formation of contexts returning **true** if and only if $\Gamma \vdash \diamond$,
- $ChTy(\Gamma, v, T)$ which checks the types of values returning **true** if and only if $\Gamma \vdash v : T$.

$ChEnv(\Gamma)$ checks that the environment Γ is well-formed. This 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, $ChTy(\Gamma, v, T)$ checks if $T = \mathbf{Int}$; otherwise it checks if $v : T$ is in the environment Γ, and then calls $ChEnv(\Gamma)$.

[8] Technically, this allows us to correct a problem present in [18], namely the failure of Subject Congruence.

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash \diamond \quad \mathtt{ranCh}(\Gamma) \subseteq \{\mathbf{1}, \bot_{\{\alpha,\overline{\alpha}\}}\}}{\Gamma \vdash \mathtt{stop} : (\!|\,|\!)}\ \text{Type Stop} \qquad \frac{\Gamma \vdash P : e \quad e \leq e' \quad \mathtt{fn}(e') \subseteq \mathtt{dom}(\Gamma)}{\Gamma \vdash P : e'}\ \text{Type Subs}$$

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

Fig. 2. Well-typed process expressions

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

Fig. 3. Well-typed values

(i) $\emptyset \asymp \emptyset$

(ii) $\Gamma \asymp \Gamma'$ implies
- (a) $\Gamma \cdot a : T \asymp \Gamma' \cdot a : T$
- (b) $\Gamma \cdot k : \alpha \asymp \Gamma' \cdot k : \overline{\alpha}$
- (c) $\Gamma \cdot k : \alpha \asymp \Gamma'$, if $k \notin \mathtt{dom}(\Gamma')$
- (d) $\Gamma \asymp \Gamma' \cdot k : \alpha$, if $k \notin \mathtt{dom}(\Gamma)$

(i) $\emptyset \circ \emptyset = \emptyset$

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

(iii) $(\Gamma \cdot k : \alpha) \circ (\Gamma' \cdot k : \overline{\alpha}) = (\Gamma \circ \Gamma') \cdot k : \bot_{\{\alpha,\overline{\alpha}\}}$

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

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

Fig. 4. Compatible environments and composition of environments

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

Definition 3.1 *Extended environments* extend plain environments by allowing channel names to be associated with plain types of the form $\sigma(\alpha)$ (session types). Given an extended 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 $k \in \mathtt{domChoice}(\Gamma)$, 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 3.2 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 $\mathtt{fn}(P) \cup \mathtt{fn}(Q) \not\subseteq \mathtt{dom}(\Gamma)$, then $\mathit{split}(\Gamma, P, Q) = \mathbf{fail}$.
- If there exists $k \in \mathtt{fn}(P) \cap \mathtt{fn}(Q)$ such that $\Gamma(k) \neq \bot_{\{\alpha,\overline{\alpha}\}}$ for any α, then $\mathit{split}(\Gamma, P, Q) = \mathbf{fail}$.
- Otherwise, $\mathit{split}(\Gamma, P, Q) = (\Gamma_1, \Gamma_2)$ where Γ_1 and Γ_2 are extended environments defined by the following:
  - $\mathtt{dom}(\Gamma_1) \subseteq \mathtt{dom}(\Gamma)$ and $\mathtt{dom}(\Gamma_2) \subseteq \mathtt{dom}(\Gamma)$.
  - For all $a \in \mathtt{domPl}(\Gamma)$, $a \in \mathtt{dom}(\Gamma_1) \cap \mathtt{dom}(\Gamma_2)$ and $\Gamma_1(a) = \Gamma_2(a) = \Gamma(a)$.
  - For $k \in \mathtt{fn}(P) \cap \mathtt{fn}(Q)$ and $\Gamma(k) = \bot_{\{\alpha,\overline{\alpha}\}}$, $\Gamma_1(k) = \Gamma_2(k) = \sigma(\alpha)$.
  - For all $k \in \mathtt{fn}(P)$ but $k \notin \mathtt{fn}(Q)$, $k \notin \mathtt{dom}(\Gamma_2)$ and $\Gamma_1(k) = \Gamma(k)$.
  - For all $k \in \mathtt{fn}(Q)$ but $k \notin \mathtt{fn}(P)$, $k \notin \mathtt{dom}(\Gamma_1)$ and $\Gamma_2(k) = \Gamma(k)$.
  - For all $k \in \mathtt{dom}(\Gamma) \setminus (\mathtt{fn}(P) \cup \mathtt{fn}(Q))$, we will (arbitrarily) assign $\Gamma_1(k) = \Gamma(k)$, and have $k \notin \mathtt{dom}(\Gamma_2)$.

These definitions are used in the clause defining $Ch(\Gamma, P|\, Q)$. The function *split* is used to divide the environment Γ into two extended environments, Γ_1 and Γ_2, such that for appropriate $\Gamma'_1 \in \Sigma(\Gamma_1)$ and $\Gamma'_2 \in \Sigma(\Gamma_2)$, $\Gamma = \Gamma'_1 \circ \Gamma'_2$. 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 $k \in \mathtt{dom}(\Gamma) \setminus (\mathtt{fn}(P) \cup \mathtt{fn}(Q))$. Secondly, if $\Gamma(k) = \bot_{\{\mathrm{I},\overline{\mathrm{I}}\}}$, 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 the fact that they do not alter the result of the type checking function Ch (see [3] for a proof of this fact).

3.1 Defining the Typechecking Function Ch

We define $Ch(\Gamma, P)$ by induction on the length of P. 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.

If $\mathtt{fn}(P) \not\subseteq \mathtt{dom}(\Gamma)$, then $Ch(\Gamma, P)$ is defined to return **fail**. In all subsequent cases we will assume that $\mathtt{fn}(P) \subseteq \mathtt{dom}(\Gamma)$. In most cases the definition of Ch can be read-off from the type rules. For example, if P is $\mathtt{request}\ a(k)\ \mathtt{in}\ Q$, then we let k' be a fresh channel variable not present in $\mathtt{dom}(\Gamma)$ and define $Ch(\Gamma, \mathtt{request}\ a(k)\ \mathtt{in}\ P)$ as:

- $Ch(\Gamma \cdot k' : \overline{\alpha}, P\{k \leftarrow k'\})$ if $\Gamma(a) = \sigma(\alpha)$, and
- **fail**, otherwise.

The exception is the case of parallel composition, which requires further attention. If $split(\Gamma, P, Q)$ returns **fail**, then $Ch(\Gamma, P|\, Q)$ is defined to return **fail** too. Otherwise, if $split(\Gamma, P, Q) \neq$ **fail**, let $(\Gamma_1, \Gamma_2) = split(\Gamma, P, Q)$. Notice that $\mathtt{domChoice}(\Gamma_1) = \mathtt{domChoice}(\Gamma_2)$ 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 $2^{|\mathtt{domChoice}(\Gamma_i)|} \leq 2^{|\mathtt{domCh}(\Gamma_i)|}$. We define $Ch(\Gamma, P|\, Q)$ as:

- $Ch(\Gamma'_1, P) + Ch(\Gamma'_2, Q)$, if there exists $\Gamma'_1 \in \Sigma(\Gamma_1)$ and $\Gamma'_2 \in \Sigma(\Gamma_2)$ such that $Ch(\Gamma'_1, P) \neq$ **fail** and $Ch(\Gamma'_2, Q) \neq$ **fail** and for all $k \in \mathtt{domChoice}(\Gamma_1) = \mathtt{domChoice}(\Gamma_2)$, $\Gamma'_1(k) = \overline{\Gamma'_2(k)}$, and
- **fail**, otherwise.

Note that 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 $Ch(\Gamma'_1, P) \neq$ **fail** and $Ch(\Gamma'_2, Q) \neq$ **fail**.

3.2 Properties of Ch

There are several points in the definition of Ch where one of two kinds of choices are made: The first is the choice of fresh names and the second 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 $Ch(\Gamma'_1, Q) \neq \textbf{fail}$ and $Ch(\Gamma'_2, R) \neq \textbf{fail}$. Nonetheless, the following result holds.

Proposition 3.3 (Well-definedness of Ch) *Ch is a total function.*

Proof. This relies on two main lemmas. The first one states the choice of fresh name does not affect the output of Ch. The second one states that if the aforementioned specializations Γ'_1 and Γ'_2 exist, then they are unique. Finally, it is noted that the size of the third argument (process P) decreases in every recursive call. □

For the proof of completeness we may assume that the type derivation of $\Gamma \vdash P : e$ does not include applications of Type Subs. This follows from the observation that if $\Gamma \vdash P : e$, then for some $e' \leq e$, $\Gamma \vdash P : e'$ is derivable without using Type Subs.

Proposition 3.4 (Completeness) *If $\Gamma \vdash P : e$, then $Ch(\Gamma, P) \neq \textbf{fail}$ and $Ch(\Gamma, P) \leq e$.*

Proof. By induction on the derivation of $\Gamma \vdash P : e$. All cases follow from standard lemmas except for the parallel composition case. This case requires the following result whose proof is simple but tedious.

Lemma 3.5 *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 $Ch(\Gamma_1, P) \neq \textbf{fail}$ and $Ch(\Gamma_2, q) \neq \textbf{fail}$ and $split(\Gamma, P, Q) \neq \textbf{fail}$ for some processes P and Q. Let $(\Pi_1, \Pi_2) = split(\Gamma, P, Q)$. Then 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 $Ch(\Gamma'_1, P) = Ch(\Gamma_1, P)$ and $Ch(\Gamma'_2, Q) = Ch(\Gamma_2, Q)$.*

The proof of the parallel composition case proceeds as follows: Suppose 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 $\Gamma_1 \vdash Q : e_1$ and $\Gamma_2 \vdash R : e_2$ 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 $Ch(\Gamma_1, Q) \neq \textbf{fail}$, $Ch(\Gamma_1, Q) \leq e_1$, $Ch(\Gamma_2, R) \neq \textbf{fail}$, and $Ch(\Gamma_2, R) \leq e_2$.

Since $Ch(\Gamma_1, Q) \neq \textbf{fail}$ and $Ch(\Gamma_2, R) \neq \textbf{fail}$, we have $\mathtt{fn}(Q) \subseteq \mathtt{dom}(\Gamma_1)$ and $\mathtt{fn}(R) \subseteq \mathtt{dom}(\Gamma_2)$. Also, 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 $split(\Gamma, Q, R)$ is defined and we may take $(\Pi_1, \Pi_2) = split(\Gamma, Q, R)$. By Lemma 3.5 there exist $\Gamma'_1 \in \Sigma(\Pi_1)$ and $\Gamma'_2 \in \Sigma(\Pi_2)$ such that $\Gamma'_1 \asymp \Gamma'_2$ and $Ch(\Gamma'_1, Q) = Ch(\Gamma_1, Q)$ and $Ch(\Gamma'_2, R) = Ch(\Gamma_2, R)$. 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 $Ch(\Gamma, Q|\, R) \neq \textbf{fail}$ and

$$Ch(\Gamma, Q|\, R) = Ch(\Gamma'_1, Q) + Ch(\Gamma'_2, R) = Ch(\Gamma_1, Q) + Ch(\Gamma_2, R) \leq e_1 + e_2 = e$$

□

Proposition 3.6 (Soundness) *If* $Ch(\Gamma, P) \neq \textbf{fail}$, *then* $\Gamma \vdash P : Ch(\Gamma, P)$.

Proof. By induction on the definition of $Ch(\Gamma, P)$. We show two sample cases.

- $\mathtt{accept}\ a(k)\ \mathtt{in}\ P$: By the definition of Ch, $Ch(\Gamma, \mathtt{accept}\ a(k)\ \mathtt{in}\ P) = Ch(\Gamma \cdot k' : \alpha, P\{k \leftarrow k'\})$, where $k' \notin \mathtt{dom}(\Gamma)$ and $\Gamma(a) = \sigma(\alpha)$. By the induction hypothesis, $\Gamma \cdot k' : \alpha \vdash P\{k \leftarrow k'\} : Ch(\Gamma \cdot k' : \alpha, P\{k \leftarrow k'\})$. By the definition of Ch, and applying Type Acpt, $\Gamma \cdot a : \sigma(\alpha) \vdash \mathtt{accept}\ a(k)\ \mathtt{in}\ P : Ch(\Gamma, \mathtt{accept}\ a(k)\ \mathtt{in}\ P)$.
- $P|\, Q$: By the definition of Ch, $Ch(\Gamma, P|\, Q) = Ch(\Gamma'_1, P) + Ch(\Gamma'_2, Q)$, 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, $\Gamma'_1 \vdash P : Ch(\Gamma'_1, P)$ and $\Gamma'_2 \vdash Q : Ch(\Gamma'_2, Q)$. Finally, by the rule Type Par, the result follows.

□

Corollary 3.7 (Minimum Effects) *If* $\Gamma \vdash P : e$, *then* $\Gamma \vdash P : Ch(\Gamma, P)$ *and* $Ch(\Gamma, P) \leq e$.

Proof. The result holds immediately from Soundness (Proposition 3.6) and Completeness (Proposition 3.4). □

We can now state our main result.

Corollary 3.8 (Decidability of Typechecking) *Given* Γ, Θ, P *and* e *it is decidable whether* $\Gamma \vdash P : e$.

Proof. We first call $Ch(\Gamma, P)$ that always terminates, by Proposition 3.3. If $Ch(\Gamma, P) = \textbf{fail}$, by Completeness (Proposition 3.4), $\Gamma \vdash P : e$ is not derivable. If $Ch(\Gamma, P) \neq \textbf{fail}$, we check the multiset inclusion $Ch(\Gamma, P) \leq e$ which is also decidable. If $Ch(\Gamma, P) \leq e$ holds, then by Soundness (Proposition 3.6) and Type Subs, $\Gamma \vdash P : e$. If $Ch(\Gamma, P) \not\leq e$, by Completeness (Proposition 3.4), $\Gamma \vdash P : e$ is not derivable. □

4 Conclusions and Future Work

A session type describes the interactions between two parties within multi-party 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. Iris allows the description of the exchange protocol, and 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 $Ch(\Gamma, P)$ 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 [15] and using the development to encode and reason about security and networking protocols.

Acknowledgement

We are grateful to the LSS group at Stevens for interesting discussions. We also thank Georgi Babayan, Pablo Garralda, Healfdene Goguen and Ricardo Medel for comments and suggestions.

References

[1] Bonelli, E., A. Compagnoni and E. Gunter, *Correspondence assertions for process synchronization in concurrent communications*, in: A. Brogi, J.-M. Jacquet and E. Pimentel, editors, *FOCLASA 2003. 2nd International Workshop on Foundations of Coordination Languages and Software Architectures*, Electronic Notes in Theoretical Computer Science **97** (2003), pp. 175–195.

[2] Bonelli, E., A. Compagnoni and E. Gunter, *Correspondence assertions for process synchronization in concurrent communications*, Technical Report 2003-8, Department of Computer Science, Stevens Institute of Technology (2003).

[3] Bonelli, E., A. Compagnoni and E. Gunter, *Typechecking safe process synchronization*, Technical Report 2004-3, Department of Computer Science, Stevens Institute of Technology (2004).

[4] Bonelli, E., A. Compagnoni and E. Gunter, *Correspondence assertions for process synchronization in concurrent communications*, Journal of Functional Programming, Special issue on Language-Based Security **15** (2005).

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

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

[7] Gay, S., *A sort inference algorithm for the polyadic pi-calculus*, in: *Proc. of the 20th ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages* (1993), pp. 429–438.

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

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

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

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

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

[13] Gordon, A. and A. Jeffrey, *Authenticity by typing for security protocols*, Journal of Computer Security **11** (2003), pp. 451–521.

[14] Gordon, A. and A. Jeffrey, *Typing correspondence assertions for communication protocols*, Theoretical Computer Science **300** (2003), pp. 379–409.

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

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

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

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

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

[20] Igarashi, A. and N. Kobayashi, *A generic type system for the pi-calculus*, Theoretical Computer Science **311** (2004), pp. 121–163.

[21] Kobayashi, N., *Type-based analyzer for the pi-calculus*, posted on MOCA mailing list (23 of July, 2004).

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

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

A An Example in Iris

The example in this section (Fig. A.1) models a simplified electronic auction system in Iris. The three main principals of the system are: Auctioneer, Seller and Buyer. In a normal processing cycle Seller contacts Auctioneer informing of the product and initial bidding price desired. Auctioneer then waits to receive biddings from interested buyers. After some fixed amount of time, Auctioneer determines that the bidding process is over and assigns the product to the highest bidder.

Two additional processes participate in the system: SellerManager and BuyerManager. Once 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 Auctioneer receives notification from a buyer, including product of interest and bid, she delegates all further interaction with the buyer to the BuyerManager. Auctioneer thus becomes available for interaction with other buyers and sellers.

In order to keep the example simple we assume that Auctioneer can handle at most one seller at a time and that at least one buyer shall make a bid. For the same reason, 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. In order to begin operating we assume that an initial seller and buyer have been placed, namely *dummySeller* and *dummyBuyer*. For this

example we use a version of Iris extended with booleans and `if-then-else`, such extensions being straightforward to accomodate.

In what follows we describe the full set of principals:

Auctioneer. 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. Auctioneer reads in product, bid and 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 Auctioneer is dealing with. 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 Auctioneer and informs that a new seller has arrived and passes on the product and base price of this product.
- `newBidder`: selected by 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`. selected by Auctioneer to inform the buyer manager that the current highest bidder has successfully acquired the product.

```
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. A.1. Full code for the auction example.

Seller. This process defines the behavior of a seller. She requests a session with Auctioneer and lets her know that she is willing to sell a product *prod* at price *price*. Also, she lets 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 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 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.

The full system is depicted as follows

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

and is well-typed in the pure theory of session types [18] under the following type assumptions:

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

where the channel types α, β and γ are:

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

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

We provide an informal explanation of the type assigned to the session name $sAuc$. This session name is used by 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 is assumed to abide to that specified by its dual. The "&" type constructor indicates that Auctioneer must accept three operations: `sell`, `bid` and `timeout`. If the first of these operations is invoked, then 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 Auctioneer by:

Example A.1 [Changing bids]

```
bid: k?(prod, bid, sBUY) in
     request sBM(h)in h◁newBidder;
```

$h!(prod, bid - 10, sBUY)$; 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 [1].

The type system for Iris detects such badly behaved variants of the honest auctioneer by introducing correspondence assertions and applying the type-checking algorithm described in this article. Indeed, in [1] a notion of *safe* process is P introduced following [12,11,14,13]. 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 [1] that all processes which are typable with the empty effect $(\!|\,|\!)$ are safe. Example A.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. Briefly, this is achieved by first inserting a begin assertion with label $\langle prod, price, sBuy\rangle$ in Buyer just before its data on k is sent. Then, an end assertion with label $\langle prod, bid, newBuyer\rangle$ is placed in the `newBidder` operation of the BuyerManager, just after these names are read in. Finally, we augment the channel types α and β with appropriate effects:

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