Principal Type Scheme for Session Types

Álvaro Tasistro *tasistro@ort.edu.uy*
Universidad ORT Uruguay
11100, Montevideo, Uruguay

Ernesto Copello *copello@ort.edu.uy*
Universidad ORT Uruguay
11100, Montevideo, Uruguay

Nora Szasz *szasz@ort.edu.uy*
Universidad ORT Uruguay
11100, Montevideo, Uruguay

Abstract

Session types as presented in [1] model communication between processes as a structure of dialogues. The dialogues are specified by sequences of types of messages, where each type describes the format and direction of the message. The resulting system imposes a type discipline that guarantees compatibility of interaction patterns between processes of a well-typed program. The system is polymorphic in Curry's style, but no formal treatment of this aspect has been provided yet. In this paper we present a system assigning type schemes to programs and an algorithm of inference of the principal type scheme of any typable program for a significant fragment of the calculus which allows delegation of communication, i.e. transmission of channels. We use classical syntax for variables and channels, i.e. just one sort of names in each case for either bound of free occurrences. We prove soundness and completeness of the algorithm, working on individual terms rather than on α-equivalence classes. The algorithm has been implemented in Haskell and partially checked in the proof assistant Agda.

Keywords: types, principal type scheme, type inference algorithm.

1. INTRODUCTION

Systems of (dyadic) session types allow to structure programs which consist of communicating processes as networks of *dialogues*. Each such dialogue is called a *session* and is carried out through a specific sort of communication entity called a *channel*. Channels are created by a special kind of interaction occurring at ordinary ports, which we explain at once: using syntax close to that in the original presentation of session types [2], we write ***acc*** *a(k).P* to represent a process that is willing to *accept* a session at port *a*. This can interact with concurrent ***req*** *a(k').Q* which can be thought of *requesting* such session. As a consequence of the interaction, a new channel is created that will communicate the continuation processes *P* and *Q*. In these processes, the names *k* and *k'* will (respectively) represent the two *ends* of the newly created channel. Thus, and as a consequence of the dialogue restriction, each channel end in the system belongs to one and only one process.
Once the channel is created, the session takes place, i.e. a sequence of messages is interchanged. The system of types allows characterizing each session as a sequence of *message formats*, where each format specifies the direction and type of contents of the message. Such characterization is a *session type*. A process like *P* or *Q* above can in turn be characterized by the (session) types of its (free) channels, which are determined by the actions performed by the process at each of its channel ends. Let us call the set of channel types of a process its *typing*. Now, in ***acc*** *a(k).P* and ***req*** *a(k).P* the name *k* becomes bound and the process ceases to depend on it; that is to say, the typing of **acc** *a(k).P* shall not mention *k* anymore. The port *a* is, however, assigned the type of *k*. And thus it becomes in principle possible to check whether two processes ***acc*** *a(k).P* and ***req*** *a(k').Q* that expect to establish a session through *a* do indeed hold compatible

interactions. Compatible means actually *dual*, i.e. an output in one process must be mirrored by an input in the other, and with contents of the same type. Thereby, type correctness ensures absence of compatibility errors in communication and freedom from interference of third parties in the dialogues. The safety property can be characterized as freedom from dead-lock in the case that sessions do not overlap.

As already said, session types appeared in [3,2]. Next to that, [4] allowed the transmission of channels in sessions, i.e. the possibility of implementing *delegation*: a process can then delegate a session to another process that takes over the dialogue. It does so by sending the second process the corresponding channel end (which is then definitely lost by the original process.)

The system of types was later refined in [1], following ideas of [5]. That is the system that we shall consider in this paper. The problem to study is that of *type inference*, i.e. of performing type checking even when (some) type declarations are omitted. Actually the system in [1] is polymorphic in Curry's style and admits a definition of *principal type scheme*. The principal type scheme can be computed for each well-formed process, even without any type declaration of variables, channels or ports. Such is the contribution of this paper: a formal treatment of session type polymorphism, in which we give an inference algorithm and prove its soundness and completeness. We here carry out the work for a fragment of the original type system including channel delegation but not choice or recursion. These restrictions are not essential, as we shall indicate. We have implemented our algorithm in Haskell and in the proof assistant Agda [6], in which the proof of soundness has been completely fomalized.

This kind of work has not been done elsewhere, as far as we know. In [5] a type checking (not inference) algorithm is given and its soundness proven, for a system more expressive than the one considered here, since it allows for subtyping in the session types. In [7] a simple version of session types is studied. in which only two implicit channels can be simultaneously used. As a consequence of this, delegation of channels is not possible. The type system is also simplified not allowing recursive types. In this work type safety is proven and an OCaml implementation of an inference algorithm is presented, for which proofs of some basic results are given.

There are some other related works that embed session types in other programming languages. In [8] and [9] session types are implemented in Haskell, making use of its powerful polymorphic type system and type classes with functional dependencies [10]. In the first work only one channel is implemented and soundness of the embedded system is proven. In the second one multiple channels are allowed but no soundness property is given. In [11] a more general technique is given to embed session types with multiple channels, thus earning more portability in the host language. In particular, any polymorphic language can be used as host. Soundness is proven but only for one channel.

The rest of the paper is organized as follows: in the next section we introduce the process language and the type system, adapted from the one in [1]. We notice it is polymorphic and then introduce the notion of type scheme, employing type variables. In section 3 we formulate our algorithm of type inference which computes, for every typable program, its principal type scheme, i.e. a type scheme assignable to the program and from which every other typing of the program can be obtained as a type substitution instance. We give detailed proofs of soundness and completeness of the algorithm. We expose conclusions and remaining work in section 4.

2. SESSION TYPES

Syntax of processes is as follows:

P : − 0 | k!e ; P | k?x. P | k!!k' ; P | k??k'. P | acc a(k). P | req a(k). P | P |Q

We now informally explain their meaning. In what follows *channel* and *channel end* are used interchangeably:
- *0* is the inactive process.
- In the term *k!e ; P*, *k* is a channel end and *e* an expression whose value is data to be sent along *k*. Then the process continues behaving as *P*.
- In *k?x. P*, data is received in the channel end *k*. The variable *x* is bound in the term.
- The term *k!!k' ; P* sends the channel end *k'* along the channel *k* and then becomes *P*.
- Dually, *k??k'. P* receives a channel in the channel end *k*. The name *k'* becomes bound in the term.
- The meaning of *acc a(k). P* and *req a(k). P* is related to session initiation and has already been explained. The name *k* is bound in both terms.
- Finally, *P |Q,* is the parallel composition of processes *P* and *Q*.

As usual, we assume denumerable sets of channel names and of variables. Also, as is evident from the syntax above, we assume a class of data expressions to be specified separately.
The syntax has been chosen so as to include those cases that are essential for the study of compatibility of interaction. In this regard, the only constructs that could be said missing are the choice operators. But consideration of these adds only technical difficulties that lie somehow beside the problem we are interested in. Also for completing a sufficiently expressive language we should include recursion or replication. We shall comment on this later.

We now turn to the consideration of types. We shall assume that an appropriate type system exists for the data expressions, whose properties are to be stated when necessary. Let for the moment δ stand for data types. Then session types are as follows:

$$\alpha,\beta : - 1 \mid \uparrow\delta ; \alpha \mid \downarrow\delta ; \alpha \mid \uparrow\alpha ; \beta \mid \downarrow\alpha ; \beta$$

i.e. they are finite sequences of message formats, each of which specifies the direction (↑ = out, ↓ = in) and type of the contents of the message (type of data or type of a channel being sent or received in delegation). *1* stands for impossibility of communication. Should we consider recursion, we would have to allow for (finite descriptions of) infinite sequences. Also if we considered choice there would have to be a branching construct.

The *dual* $\underline{\alpha}$ of a type α is defined as follows:

$$\underline{1} = 1$$
$$\underline{\uparrow\delta; \alpha} = \downarrow\delta; \underline{\alpha}$$
$$\underline{\downarrow\delta; \alpha} = \uparrow\delta; \underline{\alpha}$$
$$\underline{\uparrow\alpha; \beta} = \downarrow\alpha; \underline{\beta}$$
$$\underline{\downarrow\alpha; \beta} = \uparrow\alpha; \underline{\beta}$$

A typing judgement is of the form $\Gamma; \Pi \vdash P \triangleright \Delta$, where:

- *P* is the program being typed.
- Δ is the channel context, recording the types of the free channels of the program *P*.
- Γ is the data context, containing the declarations of the free (data) variables of *P*.
- Π is the port context, with the declarations of the ordinary ports of P.

Data contexts Γ are finite partial functions from data variables to data types. The application of function *F* to argument *a* will be written *F a*. The *union* of two data contexts Γ, Γ' is still a valid data context when $\Gamma\ x = \Gamma'\ x$ for every variable *x* which is defined (declared) in both Γ and Γ'. Port contexts Π are, similarly, finite partial functions from sort names to session types.
Channel contexts Δ are instead total functions from channel names to channel types, *1* almost everywhere. This choice proves to be convenient and reflects the fact that unused and unusable (inactive) channels are indistinguishable. In particular, ***1*** is the constant function everywhere equal to *1*. Two channel typings Δ and Δ' are *disjoint*, to be written $\Delta \ / \ \Delta'$ iff for every channel *k*, at

least one of Δk and $\Delta' k$ is *1*. The *union* of two disjoint channel typings, to be written $\Delta.\Delta'$, is such that for every channel *k*, $(\Delta.\Delta')k$ is the sum of Δk and $\Delta' k$, where sum has *1* as (left and right) identity element. *Overriding* a function *F* with a pair *(a, b)* is written $F \prec\!+ a \to b$ and gives value *F x* for every $x \neq a$, whereas it gives value *b* for argument *a*. In the case of channel and data contexts, we will write : in overridings instead of the symbol →. When treating channel contexts it will prove sometimes convenient to use a notation for a strong form of overriding to be written · and that can be called *extension*. Specifically, $\Delta \cdot k : \alpha$ means the overriding of Δ with the pair *(k,* α*)* but requiring further $\Delta k = 1$.

The type system is exposed in Figure 1.

$$\textit{inact:}\ \frac{}{\Gamma;\ \Pi\ \vdash\ 0 \rhd \mathbf{1}}$$

$$\textit{snd:}\ \frac{\Gamma \vdash e :\ \delta \qquad \Gamma;\ \Pi \vdash P \rhd \Delta}{\Gamma\ ;\ \Pi \vdash k!e\ ;\ P \rhd \Delta \prec\!+ k{:}\uparrow\delta\ ;\ \Delta k}$$

$$\textit{rcv:}\ \frac{\Gamma \prec\!+ x{:}\ \delta\ ;\ \Pi \vdash P \rhd \Delta}{\Gamma\ ;\ \Pi \vdash k?x.\ P \rhd \Delta \prec\!+ k{:}\downarrow\delta\ ;\ \Delta k}$$

$$\textit{thrw:}\ \frac{\Gamma\ ;\ \Pi \vdash P \rhd \Delta}{\Gamma\ ;\ \Pi \vdash k!!k'\ ;\ P \rhd \Delta \prec\!+ k{:}\uparrow\alpha\ ;\ \Delta k \cdot k'{:}\alpha}$$

$$\textit{ctch:}\ \frac{\Gamma\ ;\ \Pi \vdash P \rhd \Delta \cdot k'{:}\alpha}{\Gamma\ ;\ \Pi \vdash k??k'.\ P \rhd \Delta \prec\!+ k{:}\downarrow\alpha\ ;\ \Delta k}$$

$$\textit{acc:}\ \frac{\Gamma\ ;\ \Pi \vdash P \rhd \Delta \cdot k{:}\ \Pi a}{\Gamma;\ \Pi \vdash acc\ a(k).\ P \rhd \Delta}$$

$$\textit{req:}\ \frac{\Gamma\ ;\ \Pi \vdash P \rhd \Delta \cdot k{:}\ \underline{\Pi a}}{\Gamma;\ \Pi \vdash req\ a(k).\ P \rhd \Delta}$$

$$\textit{conc:}\ \frac{\Gamma\ ;\ \Pi \vdash P \rhd \Delta \qquad \Gamma\ ;\ \Pi \vdash Q \rhd \Delta'}{\Gamma;\ \Pi \vdash P \mid Q \rhd \Delta.\Delta'}\ \Delta/\Delta'$$

FIGURE 1: The Type System

We proceed to explain the rules:

- First, the rule *inact* establishes that any channel is *completed* (no longer usable) in process *0.*
- The next *snd* rule corresponds to the event of *sending* data through the channel *k*. We assume the existence of a type system for data expressions in which it is possible to type these under declarations of its variables, which are of course the variables of our programs. That explains the first premise of the rule. The second premise types the continuation process *P*, and then the conclusion updates the typing of *P* with the new type of *k*, obtained by prefixing $\uparrow\delta$ to the type sequence characterizing *k* in the continuation process.
- The third rule *rcv* corresponds to *receiving* data through a channel. A variable *x* is used and its declaration updates the data context in the typing of the continuation process *P*. The type declared to *x* is of course the type of the data received in the resulting typing in the conclusion of the rule.
- Next comes the rule *thrw* corresponding to sending (*throwing*) a channel end through a channel. The thrown channel end must be named with a fresh identifier, i.e. one not occurring in the continuation process P. This reflects the fact that the channel end will no longer belong to the process that just threw it over. In the rule the condition is imposed by the use of the extension operator · in the conclusion. Notice that the rule can be applied for whatever type is associated to the thrown channel.

- The next rule *ctch* is for receiving (*catching*) a channel sent over by a communicating process. The name *k'* used to represent the received channel end becomes bound, which is reflected in the fact that it cannot appear in the typing Δ. Syntactically, this is enforced by use of the extension operator in the premise. In fact, the names *k* and *k'* could coincide (since after all no restriction should be placed in the choice of the name *k'*). In this case the resulting typing of the process at hand depends on *k* and reflects that it becomes unusable after the catching.
- The rule *acc* is for *accepting* a session. The channel end *k* becomes bound and similar considerations as above apply. But there is a detail to comment, which concerns the type assigned to *k* in order to type the continuation process. This type is the same that the port *a* has in the port context Π. This means two things: firstly, *a* has to be declared in Π, and this ought to be made explicit as a side condition to the rule. The reason why we have omitted this has to do only with brevity of the presentation and will become clearer below. Secondly, the rule reflects that the typing of the ordinary ports is the type of the channel end created by interaction at that port.
- This is to be linked to the next, dual, rule *req* for typing a *request* of session. What we require in this case is that the channel behaves in a manner dual to the type of the port, and that will make it dual to the type of the opposite end of the channel created at the interaction of acceptance and request. That is to say, in a parallel composition of an acceptance and a request of a session the port is typed uniformly in both cases, and it is the channel ends which have to receive dual types.
- Finally, the rule *conc* of *concurrent composition* of processes requires that no channel end belongs to more than one process (disjointness of the channel typings) and that all variables and ports are uniformly typed in both processes.

Notice that no type declaration is required in the syntax of terms for any of the variables, ports or channels. This is coincident with the formulation in [1], of which the system presented here is a slight variant. The system is thus polymorphic à la Curry. Examples of polymorphic terms are: *acc a(k).k?x.k!x; 0* and *acc a(k).acc b(k'). . . . k!!k'; P* .
This motivates the investigation of type schemes. We therefore consider a denumerable set of *type variables* a^t and define the (session) *type schemes* as follows:

$$\alpha : - a^t \mid \underline{a}^t \mid 1 \mid \uparrow\delta\ ;\ \alpha \mid \downarrow\delta;\ \alpha \mid \uparrow\alpha\ ;\ \alpha \mid \downarrow\alpha\ ;\ \alpha$$

The scheme $\underline{a}^t$ is there just to stand as the dual of type variable a^t (and its dual is of course a^t).

We then consider the type system given before with two modifications: first, we assign type schemes in place of types; and, secondly, only for the sake of simplicity of the treatment, we shall consider port contexts as total functions from port names to type schemes. The (new) port contexts shall be built by successive instantiations from an original context in which every port name has associated a different type variable. We call this the *void* or *purely generic* port context and write it Ω. It can be implemented by assuming that each port name *a* can be encoded uniquely as a type variable a^t. We define $dom\ \Pi = \{a \mid \Pi a \neq a^t\}$ for port context Π. This set will always be finite. We further define two port contexts Π and Π' to be *compatible* when for every port *a*, either $\Pi a = \Pi' a$ or one of them is a type variable. For compatible port contexts Π and Π', define the *union* $\Pi.\Pi'$ to take, for each port *a*, the common value at *a* of Π and Π' if such is the case, or the more instantiated one otherwise. We insist in that these considerations are only for simplicity in the treatment to be presented below and are not essential to it. Otherwise, the system with type schemes is exactly like the one above. In particular, the rules are the same as displayed in Figure 1.

We have also to consider *type substitutions*. These are finite partial functions from type variables to type schemes and their action on type schemes is defined in the obvious way. We only have to remark that substitution of type scheme α for a^t in $\underline{a}^t$ yields $\underline{\alpha}$.

Finally, we assume that a similar extension to type schemes can be applied to the system of typing of data expressions. Then the following two basic results are obtained, provided they hold too for the system of data expressions:

Lemma 1 (Weakening). If $\Gamma; \Pi \vdash P \rhd \Delta$ and $\Gamma \subseteq \Gamma'$ then $\Gamma'; \Pi \vdash P \rhd \Delta$.

Proof. Immediate induction on the type system. Use that $\Gamma \prec\!+ x: \delta \subseteq \Gamma' \prec\!+ x: \delta$ if $\Gamma \subseteq \Gamma'$.

Lemma 2 (Closure under type substitution). If $\Gamma; \Pi \vdash P \rhd \Delta$ then for any type substitution θ, $\Gamma\theta; \Pi\theta \vdash P \rhd \Delta\theta$.

Proof. Induction on the type system. Use that for any type substitution σ and type scheme α, $\underline{\alpha}\sigma = \underline{\alpha\sigma}$.

3. TYPE INFERENCE

An inference algorithm for the given type system is displayed in Figure 2.
We make use of the form of judgement $\Gamma; \Pi \leftarrow P \rightarrow \Delta$ with the obvious meaning, i.e. given program P the algorithm infers (if possible) the contexts Γ, Π and Δ. Further, as shall be proven presently, the typing inferred in case of success is the most general that can be assigned to P in the type system, i.e. it is the principal type scheme of P. This means that every other typing of P can be obtained from the one inferred by applying to this a suitable type substitution.
A simple inspection reveals that for each program P the inferred typing is unique up to the choice of the type variables used to construct it. The type variables are introduced in the (conclusions of the) rules rcv_2 and *thrw,* and as will be shown, the choice of particular names is immaterial once certain basic conditions of freshness are ensured, namely that the names are fresh w.r.t. the set of type variables used in each rule's premise. This allows us to make the following convention in order to simplify the presentation: in rules with two premises, no type variable is used in both premises. And in rules in which we introduce type variables, these are fresh w.r.t. the set of type variables used in the premises.

$$inact: \ \frac{}{\varnothing; \Omega \leftarrow 0 \rightarrow \mathbf{1}}$$

$$snd: \ \frac{\Gamma \leftarrow e \rightarrow \delta \qquad \Gamma'; \Pi \leftarrow P \rightarrow \Delta}{\Gamma\theta \cup \Gamma'\theta; \Pi\theta \leftarrow k!e\ ; P \rightarrow \Delta\theta \prec\!+ k: \hat{\ }\delta\theta\ ; (\Delta\theta)k} \ \Gamma \cup^{\theta} \Gamma'$$

$$rcv_1: \ \frac{\Gamma\ ; \Pi \leftarrow P \rightarrow \Delta}{\Gamma \setminus x\ ; \Pi \leftarrow k?x.\ P \rightarrow \Delta \prec\!+ k:\downarrow\Gamma x\ ; \Delta k} \ x \in dom\ \Gamma$$

$$rcv_2: \ \frac{\Gamma\ ; \Pi \leftarrow P \rightarrow \Delta}{\Gamma\ ; \Pi \leftarrow k?x.\ P \rightarrow \Delta \prec\!+ k:\downarrow \mathsf{a}^{\mathsf{t}}\ ; \Delta k} \ x \notin dom\ \Gamma$$

$$thrw: \ \frac{\Gamma\ ; \Pi \leftarrow P \rightarrow \Delta}{\Gamma\ ; \Pi \leftarrow k!!k'\ ; P \rightarrow \Delta \prec\!+ k:\uparrow \mathsf{a}^{\mathsf{t}}\ ; \Delta k \cdot \mathsf{k}':\mathsf{a}^{\mathsf{t}}}$$

$$ctch: \ \frac{\Gamma\ ; \Pi \leftarrow P \rightarrow \Delta}{\Gamma\ ; \Pi \leftarrow k??k'\ ; P \rightarrow \Delta \setminus k' \prec\!+ k:\downarrow\Delta k'\ ; (\Delta \setminus k')k}$$

$$acc: \ \frac{\Gamma\ ; \Pi \leftarrow P \rightarrow \Delta}{\Gamma\theta; \Pi\theta \leftarrow acc\ a(k).\ P \rightarrow \Delta\theta \setminus k} \ \Pi a \cup^{\theta} \Delta k$$

$$req: \ \frac{\Gamma\ ; \Pi \leftarrow P \rightarrow \Delta}{\Gamma\theta; \Pi\theta \leftarrow req\ a(k).\ P \rightarrow \Delta\theta \setminus k} \ \Pi a \cup^{\theta} \underline{\Delta k}$$

$$conc:\ \frac{\Gamma\ ;\ \Pi \leftarrow P \rightarrow \Delta \qquad \Gamma'\ ;\ \Pi' \leftarrow Q \rightarrow \Delta'}{\Gamma\theta \cup \Gamma'\theta;\ \Pi\theta.\Pi'\theta \leftarrow P \mid Q \rightarrow \Delta\theta.\Delta'\theta}\ \ \Delta/\Delta'\ ,\ (\Gamma,\Pi)\ \cup^{\theta}\ (\Gamma',\Pi')$$

FIGURE 2: The Inference Algorithm

We now explain the rules. The general idea is of course to infer the minimal and most general contexts that fit the given program.
First, the rule *inact* assigns to *0* the void contexts.
In the *snd* (send) rule use is made of inference of type of data expressions –we assume such algorithm to be available– which gives the first premise. The second premise corresponds to the (recursive) inference of typing of the continuation process *P*. Then the condition for success of the rule is that the two inferred contexts Γ and Γ' unify, i.e. that the (data) type schemes at their common variables unify. This is (we assume) standard first order unification, which is decidable and yields in case of success a most general unifier θ. This is what is expressed by the side condition $\Gamma \cup^{\theta} \Gamma'$ to the rule. The conclusion obtains immediately by realizing that every context has to be instantiated by θ and Γ and Γ' have to be put together. Besides, the typing of the process at hand has to be updated with the type inferred for the channel *k*.
For the *rcv* (receive) rule there are two subcases. Once the continuation process *P* has been recursively typed one checks whether the data variable *x* is used in *P* or not. In the first case, the type assigned to *x* in the data context is recorded in the type of the channel as being received. If otherwise the variable is not used in *P*, then any type does since any value can be received. Therefore we update the channel *k* with the mark of input of a fresh type variable. According to the convention given above, this variable can be any one not occurring in the premise of the rule. A situation entirely similar to this last subcase arises in the next rule, in which the thrown out channel *k'* can be typed with any type whatsoever.
In the rule *catch* the point is to delete the bound name *k'* so that it does not occur in the resulting channel context. The rest of the manipulation has to do with considering the case in which the names *k* and *k'* coincide.
In the rest of the rules the novelty is the use of a unification algorithm over session types. This is expressed in the side conditions to the rules. Now session types are also first order trees if data types are and therefore such algorithm exists under the assumptions that we have established. This is actually the point on which all our development rests.

We can now prove the full correctness of our algorithm. For this we have to suppose correct the algorithm of data type inference.

Proposition 3 (Soundness of Type Inference). If $\Gamma;\ \Pi \leftarrow P \rightarrow \Delta$ then $\Gamma;\ \Pi \vdash P \triangleright \Delta$.

Proof. By induction on the rules of the inference algorithm.
Case *inact*: Immediate.
Case *snd*: Assume $\Gamma \vdash e : \delta$ (soundness of expression type inference) and $\Gamma'\ ;\ \Pi \vdash P \triangleright \Delta$ (induction hypothesis). Assume further $\Gamma \cup^{\theta} \Gamma'$ (side condition to the rule in the inference algorithm.) We then know both $\Gamma\theta \vdash e : \delta\theta$ and $\Gamma'\theta;\ \Pi\theta \vdash P \triangleright \Delta\theta$ because of the property of preservation of typing under type substitution in both type systems (expressions and session types). Now, since Γ and Γ' unify under θ, $\Gamma\theta \cup \Gamma'\theta$ is defined and, by weakening of both type systems, we get $\Gamma\theta \cup \Gamma'\theta \vdash e : \delta\theta$ and $\Gamma\theta \cup \Gamma'\theta;\ \Pi\theta \vdash P \triangleright \Delta\theta$. Hence, by rule *snd* of the session type system, $\Gamma\theta \cup \Gamma'\theta;\ \Pi\theta \vdash k!e;\ P \triangleright \Delta \prec\!\!+ k : \uparrow \delta\theta;\ (\Delta\theta)k$, as required.
Case rcv₁: Assume $\Gamma;\ \Pi \vdash P \triangleright \Delta$ (induction hypothesis) and $x \in \mathrm{dom}\ \Gamma$ (side condition to the rule.) We then know $\Gamma = \Gamma \setminus x \prec\!\!+ x : \Gamma x$ and therefore, because of the rule *rcv* of the type system, we have $\Gamma \setminus x;\ \Pi \vdash k?x.P \triangleright \Delta \prec\!\!+ k : \downarrow \Gamma x;\ \Delta k$, as required.
Case rcv₂: Assume $\Gamma;\ \Pi \vdash P \triangleright \Delta$ (induction hypothesis) and the side condition $x \notin \mathrm{dom}\ \Gamma$. *Also,* as indicated before, assume a^t fresh in $(\Gamma,\ \Pi,\ \Delta)$. Then $\Gamma \subseteq \Gamma \prec\!\!+ x : a^t$ and therefore by weakening,

$\Gamma \prec\!\!+ x: a^t ; \Pi \vdash P \rhd \Delta$. Now, using rule *rcv* of the type system, we arrive at the desired $\Gamma; \Pi \vdash k?x.P \rhd \Delta \prec\!\!+ k: \downarrow a^t ; \Delta k$.
Case thrw: Immediate. Notice that the side condition needs not be used.
Case ctch: Immediate once one writes $\Delta = (\Delta \setminus k') \cdot k' : \Delta k'$.
Case acc: Assume $\Gamma; \Pi \vdash P \rhd \Delta$ and side condition $\Pi a \cup^{\theta} \Delta k$. By preservation of typing under type substitutions we know $\Gamma\theta; \Pi\theta \vdash P \rhd \Delta\theta$. Now $\Delta\theta = (\Delta\theta \setminus k) \cdot k: (\Delta\theta)k$. And $(\Delta\theta)k = (\Delta k)\theta =$ (since $\Pi a \cup^{\theta} \Delta k) = (\Pi a)\theta = (\Pi\theta)a$, whence the required $\Gamma\theta; \Pi\theta \vdash acc\ a(k).P \rhd \Delta\theta \setminus k$ by use of the rule *acc* of the type system.
Case req: Identical to the preceding one.
Case conc: Use the unification side condition, preservation of typing under type substitution, and weakening, just the same as in case *snd*.

Proposition 4 (Completeness of Type Inference). $\Gamma; \Pi \vdash P \rhd \Delta$ implies $\Gamma_1; \Pi_1 \leftarrow P \rightarrow \Delta_1$ for contexts $\Gamma_1, \Pi_1, \Delta_1$ and type substitution θ such that $\Gamma_1\theta \subseteq \Gamma$, $\Pi_1\theta = \Pi$ and $\Delta_1\theta = \Delta$.

Proof. By induction on the rules of the type system.
Case inact: Define $\theta a^t = \Pi a$ for every $a \in dom\ \Pi$.
Case snd: Assume $\Gamma_1 \leftarrow e \rightarrow \delta_1$ with $\Gamma_1\theta \subseteq \Gamma$ and $\delta_1\theta = \delta$ for appropriate type substitution θ (this corresponds to completeness of the data expression type inference system.) Assume the induction hypothesis, i.e. $\Gamma_1' ; \Pi_1' \leftarrow P \rightarrow \Delta_1'$ with $\Gamma_1'\theta' \subseteq \Gamma$, $\Pi_1'\theta' = \Pi$ and $\Delta_1'\theta' = \Delta$ for appropriate type substitution θ'. Assume further that the type variables employed in Γ_1 and Γ_1' are disjoint. Hence without loss of generality we can also take θ and θ' to possess disjoint domains. Now the union σ of these two substitutions makes both $\Gamma_1\sigma \subseteq \Gamma$ and $\Gamma_1'\sigma \subseteq \Gamma$, which means that there is a subcontext of Γ that is a type substitution instance of both Γ_1 and Γ_1'. Hence these two have a most general unifier ζ and we can apply rule *snd* of the inference algorithm to obtain $\Gamma_1\zeta \cup \Gamma_1'\zeta ; \Pi_1'\zeta \leftarrow k!e ; P \rightarrow \Delta_1'\zeta \prec\!\!+ k : \uparrow \delta\zeta ; (\Delta_1'\zeta)k$. Also because ζ is the m.g.u. of Γ_1 and Γ_1', we know that there exists ζ' such that $\sigma = \zeta\zeta'$. Further, since θ' is the subset of σ acting on the type variables of Γ_1', Π_1' and Δ_1', we have $\Gamma_1'\theta' = \Gamma_1'\zeta\zeta'$ and similarly for Π_1' and Δ_1'. Therefore in the inference above we have what is required to prove, namely $(\Gamma_1\zeta \cup \Gamma_1'\zeta)\zeta' = \Gamma_1\zeta\zeta' \cup \Gamma_1'\zeta\zeta' = \Gamma_1\theta \cup \Gamma_1'\theta' \subseteq \Gamma$, $\Pi_1'\zeta\zeta' = \Pi_1'\theta' = \Pi$ and $[\Delta_1'\zeta \prec\!\!+ k : \uparrow\delta\zeta ; (\Delta_1'\zeta)k]\zeta' = \Delta \prec\!\!+ k : \uparrow\delta; \Delta k$, where the latter can be easily checked by just distributing the substitution ζ'.
Case rcv: Assume the induction hypothesis, i.e. $\Gamma_1; \Pi_1 \leftarrow P \rightarrow \Delta_1$ with $\Gamma_1\theta \subseteq \Gamma \prec\!\!+ x: \delta$, $\Pi_1\theta = \Pi$ and $\Delta_1\theta = \Delta$ for appropriate type substitution θ.
If now $x \in dom\ \Gamma_1$ then we can apply rule rcv_1 of the inference algorithm to get $\Gamma_1 \setminus x; \Pi_1 \leftarrow k?x.P \rightarrow \Delta_1 \prec\!\!+ k : \downarrow\Gamma_1 x; \Delta_1 k$. Moreover, we have $(\Gamma_1 \setminus x)\theta \subseteq \Gamma$, which follows from $\Gamma_1\theta \subseteq \Gamma \prec\!\!+ x: \delta$, and, by hypothesis, $\Pi_1\theta = \Pi$. Finally, $[\Delta_1 \prec\!\!+ k: \downarrow\Gamma_1 x; \Delta_1 k]\theta = \Delta \prec\!\!+ k: \downarrow\delta; \Delta k$, which can be checked by distributing θ and using $\Delta_1\theta = \Delta$ as well as $(\Gamma_1 x)\theta = (\Gamma_1\theta)x = \delta$.
If otherwise $x \notin dom\ \Gamma_1$, we choose a sufficiently fresh type variable a^t and apply rule rcv_2 to get $\Gamma_1; \Pi_1 \leftarrow k?x.P \rightarrow \Delta_1 \prec\!\!+ k: \downarrow a^t ; \Delta_1 k$ and taking $\theta' = \theta \cdot a^t \rightarrow \delta$, the required conditions $\Gamma_1\theta' \subseteq \Gamma$, $\Pi_1\theta' = \Pi$, and $[\Delta_1 \prec\!\!+ k: \downarrow a^t ; \Delta_1 k]\theta' = \Delta \prec\!\!+ k: \downarrow\delta; \Delta k$ all hold.
Notice that here is a place where we introduce type variables into the inferred type scheme. It should be clear that the procedure works whatever freshness conditions are imposed to type variable a^t besides the basic one we have agreed upon, namely that a^t is fresh in the contexts Γ_1, Π_1 and Δ_1.
Case thrw: Similar to the last case above. A sufficiently fresh type variable is introduced.
Case ctch: Immediate once one notes that $(\Delta \cdot x : \alpha) \setminus x = \Delta$.
Case acc: Assume the induction hypothesis, i.e. $\Gamma_1 ; \Pi_1 \leftarrow P \rightarrow \Delta_1$, with $\Gamma_1\theta \subseteq \Gamma$, $\Pi_1\theta = \Pi$ and $\Delta_1\theta = \Delta \cdot k: \Pi a$. Notice that $(\Pi_1 a)\theta = (\Pi_1\theta)a = \Pi a = (\Delta_1\theta)k = (\Delta_1 k)\theta$. Therefore, $\Pi_1 a \cup^{\zeta} \Delta_1 k$ and $\theta = \zeta\zeta'$. We can then apply rule *acc* of the inference algorithm to obtain $\Gamma\zeta ; \Pi\zeta \leftarrow acc\ a(k).\ P \rightarrow \Delta\zeta \setminus k$. And, besides, $\Gamma_1\zeta\zeta' \subseteq \Gamma$, $\Pi_1\zeta\zeta' = \Pi$ and $(\Delta_1\zeta \setminus k)\zeta' = \Delta\zeta\zeta' \setminus k = \Delta$, as required.
Case req: Identical to the preceding one.
Case conc: Similar to first case *snd*.

Soundness and completeness, together with unicity of inference of typing (up to choice of type variables) give the result on *principal type scheme*. Actually a principal type scheme for P is, by

definition, one that is assignable to P and that satisfies the conditions exposed in the completeness theorem for all the typings $\Gamma; \Pi \vdash P \triangleright \Delta$ assignable to P.

4. CONCLUSION

The classical result of (implicitly) simply typed λ calculus, of existence and effective computability of principal type scheme of any typable term can be extended to session types. This fact has been mentioned in passing in [4] and [1] but only now has it been proven formally. What remains for us to make this result complete is to extend our present development to types of choice (branching) and recursive types. Of these, the latter seem to constitute the interesting problem. But, as pointed out above, the key point on which the given algorithm and proofs rest is the existence of a unification algorithm for session types. And, as remarked out in e.g. [12], this algorithm can be extended to unification of regular trees with all other details of the proof holding without modifications.
We have also formalized a great part of the present development in the proof assistant Agda [6], which implements a version of constructive type theory. Besides, we have implemented the inference algorithm in Haskell. All this is available in [13] and we expect to soon complete the formalization of the whole development. Notice that the treatment presented in this paper does not depend on identifying α-convertible terms and is therefore amenable to direct formalization.

Acknowledgements Ernesto Copello was partially supported by a graduate student scholarship from ANII (Agencia Nacional de Investigación e Innovación), Uruguay.

5. REFERENCES

[1] N. Yoshida and V. T. Vasconcelos. “Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication”. In *1st International Workshop on Security and Rewriting Techniques*, volume 171(4) of *ENTCS*, pages 73–93. Elsevier, 2007.

[2] K. Takeuchi, K. Honda, and M. Kubo. “An interaction-based language and its typing system”. In Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou, and Sergios Theodoridis, editors, *PARLE*, volume 817 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 1994.

[3] K. Honda. “Types for dyadic interaction”. In Eike Best, editor, *CONCUR’93*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer Berlin / Heidelberg, 1993. 10.1007/3-540-57208-2_35.

[4] K. Honda, V. T. Vasconcelos, and M. Kubo. “Language primitives and type disciplines for structured communication-based programming”. In *ESOP’98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[5] S. J. Gay and M. Hole. “Subtyping for session types in the pi calculus”. *Acta Inf.*, pages 191–225, 2005.

[6] U. Norell. “Towards a practical programming language based on dependent type theory”. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[7] L. G. Mezzina. “How to infer finite session types in a calculus of services and sessions”. In *Proceedings of the 10th international conference on Coordination models and languages*, COORDINATION’08, pages 216–231, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] M. Neubauer and P. Thiemann. “An implementation of session types”. In Bharat Jayaraman, editor, *PADL*, volume 3057 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2004.

[9] M. Sackman and S. Eisenbach. “Session Types in Haskell: Updating Message Passing for the 21st Century”. Technical report, June 2008.

[10] M. P. Jones. “Type classes with functional dependencies”. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, ESOP ’00, pages 230–244, London, UK, 2000. Springer-Verlag.

[11] R. Pucella and J. A. Tov. “Haskell session types with (almost) no class”. *SIGPLAN Not.*, 44(2):25–36, September 2008.

[12] F. Cardone and M. Coppo. “Type inference with recursive types: Syntax and semantics”. *Inf. Comput.*, 92(1):48–80, 1991.

[13] Ernesto Copello. Inferencia de tipos de sesión. Master’s thesis, Universidad ORT Uruguay, 2012.