# Types and Subtypes for Client-Server Interactions

Simon Gay[1] and Malcolm Hole[2]

[1] Department of Computer Science, Royal Holloway, University of London, Egham, Surrey, TW20 0EX, UK. Email: <S.Gay@dcs.rhbnc.ac.uk>
[2] Same address as first author. Email: <M.Hole@dcs.rhbnc.ac.uk>

**Abstract.** We define an extension of the $\pi$-calculus with a static type system which supports high-level specifications of extended patterns of communication, such as client-server protocols. Subtyping allows protocol specifications to be extended in order to describe richer behaviour; an implemented server can then be replaced by a refined implementation, without invalidating type-correctness of the overall system. We use the POP3 protocol as a concrete example of this technique.

## 1 Introduction

Following its early success as a framework for the investigation of the foundations of various concurrent programming styles, the $\pi$-calculus [12] has also become established as a vehicle for the exploration of type systems for concurrent programming languages [2, 7, 9, 11, 15, 21]. Inter-process communication in the $\pi$-calculus is based on point-to-point transmission of messages along named channels, and many proposed type systems have started from the assignment of types to channels, so that the type of a channel determines what kind of message it can carry. Because messages can themselves be channel names, this straightforward idea leads to detailed specifications of the intended uses of channels within a system. This line of research has not been purely theoretical: the Pict programming language [16] is directly based on the $\pi$-calculus and has a rich type system which incorporates subtyping on channel types and higher-order polymorphism.

Honda et al. [5, 20] have proposed $\pi$-calculus-like languages in which certain channels can be given *session types*. Such a channel, which we will call a *session channel*, is not restricted to carrying a single type of message for the whole of its lifetime; instead, its type specifies a sequence of message types. Some of the messages might indicate choices between a range of possibilities, and different choices could lead to different specifications of the types of subsequent messages; session types therefore have a branching structure. One application of session types is the specification of complex protocols, for example in client-server systems. The main contribution of the present paper is to add subtyping to a system of session types, and show that this strengthens the application to the specification of client-server protocols. The differences in syntax between

our language and the $\pi$-calculus are minimal; all the special treatment of session channels is handled by the typing rules. We anticipate that this will make it easier to achieve our next goal of incorporating our type system into a modified version of the Pict language and compiler.

Consider a server for mathematical operations, which initially offers a choice between addition and negation. A client must choose an operation and then send the appropriate number of arguments to the server, which responds by sending back the result. All communications take place on a single session channel called $x$, whose session type is

$$S = \&\langle \mathsf{plus}\colon ?[\mathsf{int}] \,.\, ?[\mathsf{int}] \,.\, ![\mathsf{int}] \,.\, \mathsf{end}, \mathsf{negate}\colon ?[\mathsf{int}] \,.\, ![\mathsf{int}] \,.\, \mathsf{end} \rangle.$$

More precisely, this is the type of the server side of the channel. The $\&\langle \ldots \rangle$ constructor specifies that a choice is offered between, in this case, two options, labelled plus and negate. Each label leads to a type which describes the subsequent communication on $x$; note that the two branches have different types, in which $?[\mathsf{int}]$ indicates receiving an integer, $![\mathsf{int}]$ indicates sending an integer, . is the sequencing constructor, and end indicates the end of the interaction. The client side of the channel $x$ has a dual or complementary type, written $\overline{S}$. Explicitly,

$$\overline{S} = \oplus\langle \mathsf{plus}\colon ![\mathsf{int}] \,.\, ![\mathsf{int}] \,.\, ?[\mathsf{int}] \,.\, \mathsf{end}, \mathsf{negate}\colon ![\mathsf{int}] \,.\, ?[\mathsf{int}] \,.\, \mathsf{end} \rangle.$$

The $\oplus\langle \ldots \rangle$ constructor specifies that the client makes a choice between plus and negate. Again, each label is followed by a type which describes the subsequent interaction; the pattern of sending and receiving is the opposite of the pattern which appears on the server side.

An implementation of a maths server must use $x$ in accordance with the type $S$, and an implementation of a client must use $x$ in accordance with the type $\overline{S}$. These requirements can be enforced by static typechecking, and it is then guaranteed that no communication errors will occur at runtime. When the client chooses a label, it is guaranteed to be one of the labels offered by the server; when the client sends a subsequent message, it is guaranteed to be of the type expected by the server; and similarly when the server sends a message.

The typing rules to be introduced in Section 3 will allow the derivation of

$$x : S^1 \vdash \mathsf{server}$$

where

$$\begin{aligned} \mathsf{server} = x \triangleright \{\, & \mathsf{plus}\colon x\,?\,[a : \mathsf{int}] \,.\, x\,?\,[b : \mathsf{int}] \,.\, x\,!\,[a+b] \,.\, \mathbf{0}, \\ & \mathsf{negate}\colon x\,?\,[a : \mathsf{int}] \,.\, x\,!\,[-a] \,.\, \mathbf{0}\}. \end{aligned}$$

The operation $\triangleright$ allows a message on $x$ to choose between the listed alternatives. The labels are the same as those in $S$, and the pattern of inputs $(x\,?\,[a : \mathsf{int}] \ldots .)$ and outputs $(x\,!\,[a+b] \ldots .)$ matches that in $S$. The usage annotation of 1 on $S$ indicates that only one side of $x$ is being used by server.

One possible definition of a client is

$$\mathsf{client} = x \triangleleft \mathsf{negate} \,.\, x\,!\,[2] \,.\, x\,?\,[a : \mathsf{int}] \,.\, \mathbf{0}$$

and we can derive the typing judgement

$$x : \overline{S}^1 \vdash \mathsf{client}.$$

Note the use of $\triangleleft$ to select from the available options, and that the subsequent pattern of inputs and outputs matches the specification in $\overline{S}$. This client does not do anything with the value received from the server; more realistically, $\mathbf{0}$ would be replaced by some continuation process which used $a$.

The client and the server can be put in parallel using the typing rule T-Par for parallel composition:

$$\frac{x : S^1 \vdash \mathsf{server} \quad x : \overline{S}^1 \vdash \mathsf{client}}{x : S^2 \vdash \mathsf{server} \mid \mathsf{client}}$$

where the usage annotation 2 on $S$ indicates that both sides of $x$ are being used.

The usage annotations are necessary in order to ensure that each side of $x$ is only used by one process. The system $\mathsf{server} \mid \mathsf{client} \mid \mathsf{client}$ is erroneous because both clients are trying to use $x$ to communicate with the same server. If this system is executed, one client will use $x$ to choose either $\mathsf{plus}$ or $\mathsf{negate}$. After that, the server expects to receive an integer on $x$, but the other client will again use $x$ to choose between $\mathsf{plus}$ and $\mathsf{negate}$. This is a runtime type error of the kind that the type system is designed to avoid. We will see later that

$$\frac{x : \overline{S}^1 \vdash \mathsf{client} \quad x : \overline{S}^1 \vdash \mathsf{client}}{x : \overline{S}^1 \vdash \mathsf{client} \mid \mathsf{client}}$$

is not a valid application of the typing rule T-Par.

To avoid runtime type errors we must ensure that session channels are used linearly [3]. Our typing rules use techniques similar to those of Kobayashi et al. [9] to enforce linearity. The type system also allows non-session types to be specified, and there are no restrictions on how many processes may use them. For example, $y : \hat{}[\mathsf{int}]$ is a channel which can be used freely to send or receive integers.

How, then, can we implement a server which can be used by more than one client? The solution is for each client to create a session channel which it will use for its own interaction with the server. The server consists of a replicated thread process; each thread receives a session channel along a channel called $\mathsf{port}$ and uses it to interact with a client.

$$\begin{aligned}
\mathsf{thread} &= \mathsf{port}\,?\,[x : S^1]\,.\,\mathsf{server} \\
\mathsf{newserver} &= !\mathsf{thread} \\
\mathsf{client1body} &= y \triangleleft \mathsf{negate} : y\,!\,[2]\,.\,y\,?\,[a : \mathsf{int}]\,.\,\mathbf{0} \\
\mathsf{client1} &= (\nu y : S^2)\mathsf{port}\,!\,[y]\,.\,\mathsf{client1body} \\
\mathsf{client2body} &= z \triangleleft \mathsf{plus} : z\,!\,[1]\,.\,z\,!\,[2]\,.\,z\,?\,[b : \mathsf{int}]\,.\,\mathbf{0} \\
\mathsf{client2} &= (\nu z : S^2)\mathsf{port}\,!\,[z]\,.\,\mathsf{client2body}
\end{aligned}$$

Now

$$\mathsf{port} : \hat{}[S^1] \vdash \mathsf{newserver} \mid \mathsf{client1} \mid \mathsf{client2}$$

is a valid typing judgement. Because port does not have a session type, it can be used by all three processes. When this system is executed, client1 sends the local channel $y$ to one copy of thread; the standard $\pi$-calculus scope extrusion allows the scope of $y$ to expand to include that copy of thread, and then client1body and server have a private interaction on $y$. Similarly client2 and another copy of thread use the private channel $z$ for their communication. Notice that $y$ is a session channel with usage 2 in client1, and indeed both sides of $y$ are used: the side whose type is $S$ is used by being sent away on port, and the side whose type is $\overline{S}$ is used for communication by client1body.

The final ingredient of our type system is subtyping. On non-session types, subtyping is defined exactly as in Pierce and Sangiorgi's type system for the $\pi$-calculus [15]: if $\forall i \in \{1,\ldots,n\}.T_i \leqslant U_i$ then $\hat{}[T_1,\ldots,T_n] \leqslant ?[U_1,\ldots,U_n]$ and $\hat{}[U_1,\ldots,U_n] \leqslant ![T_1,\ldots,T_n]$. Channels whose type permits both input and output ($\hat{}$) can be used in positions where just input or just output is required. We also have $?[T_1,\ldots,T_n] \leqslant ?[U_1,\ldots,U_n]$ and $![U_1,\ldots,U_n] \leqslant ![T_1,\ldots,T_n]$; recall that input behaves covariantly and output behaves contravariantly.

Subtyping on sequences $T_1.\cdots.T_n$ is defined pointwise, again with ? acting covariantly and ! acting contravariantly. More interesting is the definition of subtyping for branch and choice types. If a process needs a channel of type $\&\langle l_1{:}T_1,\ldots,l_n{:}T_n\rangle$ which allows it to offer a choice from $\{l_1,\ldots,l_n\}$, then it can safely use a channel of type $\&\langle l_1{:}T_1,\ldots,l_m{:}T_m\rangle$, where $m \leqslant n$, instead. The channel type prevents the process from ever receiving labels $l_{m+1},\ldots,l_n$ but every label that can be received will be understood. Furthermore, a channel of type $\&\langle l_1{:}S_1,\ldots,l_m{:}S_m\rangle$ can be used if each $S_i \leqslant T_i$, as this means that after the choice has been made the continuation process uses a channel of type $S_i$ instead of $T_i$ and this is safe. In Section 5 we will see how subtyping can be used to decribe modifications to the specification of a server.

The remainder of the paper is organised as follows. Section 2 defines the syntax of processes and types, and some basic operations on type environments. The typing rules are presented in Section 3. Section 4 defines the operational semantics of the language and states the main technical results leading to type soundness. Section 5 uses our type system to specify the POP3 protocol, and discusses the role of subtyping. Finally we discuss related work, and outline our future plans, in Section 6.

## 2 Syntax and Notation

Our language is based on a polyadic $\pi$-calculus with output prefixing [12]. We omit the original $\pi$-calculus choice construct $P+Q$, partly in order to keep the language close to the core of Pict [16]. However, we have the constructs introduced in Section 1 for choosing between a collection of labelled processes, as proposed by Honda et al. [5, 20]. We also omit the matching construct, which allows channel names to be tested for equality, again because it is not present in core Pict. The inclusion of output prefixing is different from many recent presentations of the $\pi$-calculus, but it is essential because our type system must

be able to impose an order on separate outputs on the same channel. It is convenient to add a conditional process expression, written $\mathsf{if}\, b \,\mathsf{then}\, P \,\mathsf{else}\, Q$ where $b$ is a boolean value, and therefore we also have a ground type of booleans; other ground types, such as $\mathsf{int}$ as used in the examples in Section 1, could be added along with appropriate primitive operations. As is standard, we use the replication operator ! instead of recursive process definitions.

The type system has separate constructors for input-only, output-only and dual-capability channels, as suggested by Pierce and Sangiorgi [15]. It also has constructors for session types, as proposed by Honda et al. [5, 20]. The need for linear control of session channels leads to the usage annotations on session types, which play a similar role to the polarities of Kobayashi et al. [9]. Subtyping will be defined in Section 3.

In general we use lower case letters for channel names, $l_1, \ldots, l_n$ for labels of choices, upper case $P$, $Q$, $R$ for processes, and upper case $T$, $U$ etc. for types. We write $\tilde{x}$ for a finite sequence $x_1, \ldots, x_n$ of names, and $\tilde{x} : \tilde{T}$ for a finite sequence $x_1 : T_1, \ldots, x_n : T_n$ of typed names.

### 2.1 Processes

The syntax of processes is defined by the following grammar. Note that $T$ and $\tilde{T}$ stand for types and lists of types, which have not yet been defined.

$$
\begin{array}{lll}
P ::= \mathbf{0} & & \\
\quad \mid\ P \mid Q & & \mid\ x \triangleright \{l_1 : P_1, \ldots, l_n : P_n\} \\
\quad \mid\ x\,?\,[\tilde{y} : \tilde{T}]\,.\,P & & \mid\ x \triangleleft l\ .\ P \\
\quad \mid\ x\,!\,[\tilde{y}]\,.\,P & & \mid\ !P \\
\quad \mid\ (\nu x : T)P & & \mid\ \mathsf{if}\, x \,\mathsf{then}\, P \,\mathsf{else}\, Q
\end{array}
$$

Most of this syntax is fairly standard. $\mathbf{0}$ is the inactive process, $\mid$ is parallel composition, $(\nu x : T)P$ declares a local name $x$ of type $T$ for use in $P$, and $!P$ represents a potentially infinite supply of copies of $P$. $x\,?\,[\tilde{y} : \tilde{T}]\,.\,P$ receives the names $\tilde{y}$, which have types $\tilde{T}$, along the channel $x$, and then executes $P$. $x\,!\,[\tilde{y}]\,.\,P$ outputs the names $\tilde{y}$ along the channel $x$ and then executes $P$. There should be no confusion between the use of ! for output and its use for replication, as the surrounding syntax is quite different in each case. $x \triangleright \{l_1 : P_1, \ldots, l_n : P_n\}$ offers a choice of subsequent behaviours—one of the $P_i$ can be selected as the continuation process by sending the appropriate label $l_i$ along the channel $x$, as explained in Section 1. $x \triangleleft l\ .\ P$ sends the label $l$ along $x$ in order to make a selection from an offered choice, and then executes $P$. The conditional expression has already been mentioned.

We define free and bound names as usual: $x$ is bound in $(\nu x : T)P$, the names in $\tilde{y}$ are bound in $x\,?\,[\tilde{y} : \tilde{T}]\,.\,P$, and all other occurrences are free. We then define $\alpha$-equivalence as usual, and identify processes which are $\alpha$-equivalent. We also define an operation of substitution of names for names: $P\{\tilde{x}/\tilde{y}\}$ denotes $P$ with the names $x_1, \ldots, x_n$ simultaneously substituted for $y_1, \ldots, y_n$, assuming that bound names are renamed if necessary to avoid capture of substituting names.

As usual we define a *structural congruence* relation, written $\equiv$, which helps to define the operational semantics. It is the smallest congruence (on $\alpha$-equivalence classes of processes) closed under the following rules.

$$
\begin{array}{rll}
P \mid \mathbf{0} \equiv P & & \text{S-Unit} \\
P \mid Q \equiv Q \mid P & & \text{S-Comm} \\
P \mid (Q \mid R) \equiv (P \mid Q) \mid R & & \text{S-Assoc} \\
(\nu x : T)P \mid Q \equiv (\nu x : T)(P \mid Q) & \text{if } x \text{ is not free in } Q & \text{S-Extr} \\
(\nu x : T)\mathbf{0} \equiv \mathbf{0} & & \text{S-Nil} \\
(\nu x : T)(\nu y : U)P \equiv (\nu y : U)(\nu x : T)P & & \text{S-Perm} \\
!P \equiv P|!P & & \text{S-Rep} \\
x \triangleright \{l_1 : P_1, \ldots, l_n : P_n\} \equiv x \triangleright \{l_{\sigma(1)} : P_{\sigma(1)}, \ldots, l_{\sigma(n)} : P_{\sigma(n)}\} & & \text{S-Offer}
\end{array}
$$

In rule S-Offer, $\sigma$ is a permutation on $\{1, \ldots, n\}$.

### 2.2 Types

The syntax of types is defined by the following grammar.

$$
\begin{array}{lll}
\text{Ground types} & G & ::= \mathsf{bool} \\
\text{Channel types} & C & ::= {?}[T_1, \ldots, T_n] \\
 & & \mid\ ![T_1, \ldots, T_n] \\
 & & \mid\ \hat{}[T_1, \ldots, T_n] \\
\text{Session types} & S & ::= \mathsf{end} \\
 & & \mid\ {?}[T_1, \ldots, T_n] \,.\, S \\
 & & \mid\ ![T_1, \ldots, T_n] \,.\, S \\
 & & \mid\ \&\langle l_1 : S_1, \ldots, l_n : S_n \rangle \\
 & & \mid\ \oplus\langle l_1 : S_1, \ldots, l_n : S_n \rangle \\
\text{Types} & T & ::= G | C | A \\
 & & \mid\ X (\text{type variable}) \\
 & & \mid\ \mu X.T (\text{recursive type}) \\
\text{Annotated session types} & A & ::= S^1 | S^2
\end{array}
$$

The *usage annotation*, or just *usage*, of a session type indicates how a channel of that type can be used: if $x : S^1$ then $x$ can only be used as specified by $S$, but if $x : S^2$ then both sides of $x$ can be used, including the side described by $\overline{S}$. We omit usage annotations from $\mathsf{end}$, and often omit usage annotations of 1.

We define the *unwinding* of a recursive type: $\text{unwind}(\mu X.T) = T\{\mu X.T/X\}$.

If $T$ is a type then $\overline{T}$ is the dual (or complementary) type of $T$, defined inductively as follows.

$$
\begin{array}{rlrl}
\overline{\&\langle l_1 : S_1, \ldots, l_n : S_n \rangle} & = \oplus\langle l_1 : \overline{S_1}, \ldots, l_n : \overline{S_n} \rangle & & \\
\overline{\oplus\langle l_1 : S_1, \ldots, l_n : S_n \rangle} & = \&\langle l_1 : \overline{S_1}, \ldots, l_n : \overline{S_n} \rangle & & \\
\overline{\mathsf{bool}} & = \mathsf{bool} & \overline{?[\tilde{T}]} & = ![\tilde{T}] \\
\overline{\mathsf{end}} & = \mathsf{end} & \overline{![\tilde{T}]} & = ?[\tilde{T}] \\
\overline{X} & = X & \overline{\mu X.T} & = \mu X.\overline{T}
\end{array}
$$

$S^2$ and $\overline{S}^2$ both describe a session channel of which both ends are being used. We adopt the convention that when $S^2$ is written, $S$ is either a branching type or begins with an input. We say that a session type is *complete* if it has usage 2; it is *incomplete* if it is $\mathsf{end}$ or has usage 1.

### 2.3 Environments

An environment is a set of typed names, written $x_1 : T_1, \ldots, x_n : T_n$. We use $\Gamma$, $\Delta$ etc. to stand for environments. We assume that all the names in an environment are distinct. We write $x \in \Gamma$ to indicate that $x$ is one of the names appearing in $\Gamma$, and then write $\Gamma(x)$ for the type of $x$ in $\Gamma$. When $x \notin \Gamma$ we write $\Gamma, x : T$ for the environment formed by adding $x : T$ to the set of typed names in $\Gamma$. When $\Gamma$ and $\Delta$ have disjoint sets of names, we write $\Gamma, \Delta$ for their union. Implicitly, *true* : $\mathsf{bool}$ and *false* : $\mathsf{bool}$ appear in every environment.

The partial operation + on types is defined by

$$\begin{aligned} T + T &= T \text{ if } T \text{ is a ground type, a channel type, or } \mathsf{end} \\ S^1 + \overline{S}^1 &= S^2 \text{ if } S \text{ is a session type} \end{aligned}$$

and is undefined in all other cases.

The partial operation +, combining a typed name with an environment, is defined as follows:

$$\begin{aligned} \Gamma + x : T &= \Gamma, x : T && \text{if } x \notin \Gamma \\ (\Gamma, x : T) + x : U &= \Gamma, x : (T + U) && \text{if } T + U \text{ is defined} \end{aligned}$$

and is undefined in all other cases.

We extend + to a partial operation on environments by defining

$$\Gamma + (x_1 : T_1, \ldots, x_n : T_n) = (\cdots(\Gamma + x_1 : T_1) + \cdots +) + x_n : T_n$$

We say that an environment is *unlimited* if it contains no session types except for $\mathsf{end}$.

## 3 The Type System

### 3.1 Subtyping

The principles behind the definition of subtyping have been described in Section 1. Figure 1 defines the subtype relation formally by means of a collection of inference rules for judgements of the form $\Sigma \vdash T \leqslant U$, where $\Sigma$ ranges over finite sets of instances of $\leqslant$. When $\emptyset \vdash T \leqslant U$ is derivable we simply write $T \leqslant U$. The inference rules can be interpreted as an algorithm for checking whether $T \leqslant U$ for given $T$ and $U$, as follows. Beginning with the goal $\emptyset \vdash T \leqslant U$, apply the rules upwards to generate subgoals; pattern matching on the structure of $T$ and $U$ determines which rule to use, except that the rule AS-Assump should always be used, causing the current subgoal to succeed, if it is applicable. If

both AS-Rec-L and AS-Rec-R are applicable then they should be used in either order. If a subgoal is generated which does not match any of the rules, the algorithm returns false.

Pierce and Sangiorgi [15] give two definitions of their subtype relation: one by means of inference rules (as in Figure 1) and one as a form of type simulation, defined coinductively. The subtyping algorithm derived from the inference rules can then be proved sound and complete with respect to the coinductive definition, while the coinductive definition permits straightforward proofs of transitivity and reflexivity of the subtype relation. In the same way, we can characterize our subtype relation coinductively, prove soundness and completeness of the subtyping algorithm, and prove transitivity and reflexivity; due to space constraints, we have omitted the details from the present paper.

The subtype relation is defined on non-annotated types, but annotations preserve subtyping: if $i \in \{1, 2\}$ then $S^i \leqslant T^i$ if and only if $S \leqslant T$.

If $\tilde{T}$ and $\tilde{U}$ have the same length, $n$, and $\forall i \in \{1, \ldots, n\}.T_i \leqslant U_i$, we write $\tilde{T} \leqslant \tilde{U}$.

### 3.2 Typing rules

The typing rules are defined in Figure 2. Note that a judgement of the form $\Gamma \vdash x : T \leqslant U$ means $x : T \in \Gamma$ and $T \leqslant U$. Subtyping appears in the hypotheses of rules T-Out, T-OutSeq, T-In and T-InSeq, where it must be possible to promote the type of the channel to the desired input or output type. It appears less explicitly in rules T-Offer and T-Choose, where the type of $x$ must include enough labels for the choice being constructed.

Each typing rule is only applicable when any instances of $+$ which it contains are actually defined. This ensures that the environment correctly records the use being made of session channels. Consider again the two applications of T-Par from Section 1.

$$\frac{x : S^1 \vdash \mathsf{server} \quad x : \overline{S}^1 \vdash \mathsf{client}}{x : S^2 \vdash \mathsf{server} \mid \mathsf{client}} \qquad \frac{x : \overline{S}^1 \vdash \mathsf{client} \quad x : \overline{S}^1 \vdash \mathsf{client}}{x : \overline{S}^1 \vdash \mathsf{client} \mid \mathsf{client}}$$

The first is correct because $S^1 + \overline{S}^1 = S^2$. The second is incorrect because $\overline{S}^1 + \overline{S}^1$ is not defined; this prevents the session channel $x$ from being used simultaneously by both copies of client.

Notice also that in rules T-Out and T-OutSeq, the names being output are added to the environment; this means that if a session channel is output then the part of its usage which is given away cannot be used again by the remainder of the process. This allows a process to begin a communication on a session channel, then delegate the rest of the session to another process by sending it the channel; of course, the first process must not use the channel again. Such behaviour arises when a recursive process, which uses a session channel (of a recursive type), is represented in terms of replication: when the next instance of the recursive process is invoked, the session channel must be passed on.

$$\frac{T \leqslant U \in \Sigma}{\Sigma \vdash T \leqslant U}\ \text{AS-Assump}$$

$$\frac{}{\Sigma \vdash \mathsf{bool} \leqslant \mathsf{bool}}\ \text{AS-Bool} \qquad \frac{}{\Sigma \vdash \mathsf{end} \leqslant \mathsf{end}}\ \text{AS-End}$$

$$\frac{\Sigma \vdash \tilde{T} \leqslant \tilde{U}}{\Sigma \vdash ?[\tilde{T}] \leqslant ?[\tilde{U}] \quad \Sigma \vdash \hat{}[\tilde{T}] \leqslant ?[\tilde{U}]}\ \text{AS-In}$$

$$\frac{\Sigma \vdash \tilde{U} \leqslant \tilde{T}}{\Sigma \vdash ![\tilde{T}] \leqslant ![\tilde{U}] \quad \Sigma \vdash \hat{}[\tilde{T}] \leqslant ![\tilde{U}]}\ \text{AS-Out}$$

$$\frac{\Sigma \vdash \tilde{T} \leqslant \tilde{U} \text{ and } \Sigma \vdash \tilde{U} \leqslant \tilde{T}}{\Sigma \vdash \hat{}[\tilde{T}] \leqslant \hat{}[\tilde{U}]}\ \text{AS-InOut}$$

$$\frac{\Sigma \vdash V \leqslant W \quad \Sigma \vdash \tilde{T} \leqslant \tilde{U}}{\Sigma \vdash ?[\tilde{T}] \,.\, V \leqslant ?[\tilde{U}] \,.\, W}\ \text{AS-InSeq}$$

$$\frac{\Sigma \vdash V \leqslant W \quad \Sigma \vdash \tilde{U} \leqslant \tilde{T}}{\Sigma \vdash ![\tilde{T}] \,.\, V \leqslant ![\tilde{U}] \,.\, W}\ \text{AS-OutSeq}$$

$$\frac{m \leqslant n \quad \forall i \in \{1, \ldots, m\}.\Sigma \vdash S_i \leqslant T_i}{\Sigma \vdash \&\langle l_1 : S_1, \ldots, l_m : S_m \rangle \leqslant \&\langle l_1 : T_1, \ldots, l_n : T_n \rangle}\ \text{AS-Branch}$$

$$\frac{m \leqslant n \quad \forall i \in \{1, \ldots, m\}.\Sigma \vdash S_i \leqslant T_i}{\Sigma \vdash \oplus\langle l_1 : S_1, \ldots, l_n : S_n \rangle \leqslant \oplus\langle l_1 : T_1, \ldots, l_m : T_m \rangle}\ \text{AS-Choice}$$

$$\frac{\Sigma, \mu X.S \leqslant T \vdash \text{unwind}(\mu X.S) \leqslant T}{\Sigma \vdash \mu X.S \leqslant T}\ \text{AS-Rec-L}$$

$$\frac{\Sigma, T \leqslant \mu X.S \vdash T \leqslant \text{unwind}(\mu X.S)}{\Sigma \vdash T \leqslant \mu X.S}\ \text{AS-Rec-R}$$

**Fig. 1.** Inference rules for subtyping

$$\frac{\Gamma \text{ unlimited}}{\Gamma \vdash \mathbf{0}}\ \text{T-Nil} \qquad \frac{\Gamma \vdash P \quad \Delta \vdash Q}{\Gamma + \Delta \vdash P \mid Q}\ \text{T-Par}$$

$$\frac{\Gamma, x : T \vdash P \quad \text{if } T \text{ is a session type it must be complete}}{\Gamma \vdash (\nu x : T)P}\ \text{T-New}$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash x \leqslant\, ![\tilde{T}]}{\Gamma + \tilde{y} : \tilde{T} \vdash x\,![\tilde{y}]\,.\,P}\ \text{T-Out} \qquad \frac{\Gamma, \tilde{y} : \tilde{T} \vdash P \quad \Gamma \vdash x \leqslant\, ?[\tilde{T}]}{\Gamma \vdash x\,?\,[\tilde{y} : \tilde{T}]\,.\,P}\ \text{T-In}$$

$$\frac{\Gamma, x : S \vdash P \quad S \text{ is incomplete} \quad \tilde{U} \leqslant \tilde{T}}{(\Gamma, x : ![\tilde{T}]\,.\,S) + \tilde{y} : \tilde{U} \vdash x\,![\tilde{y}]\,.\,P}\ \text{T-OutSeq}$$

$$\frac{\Gamma, x : S, \tilde{y} : \tilde{U} \vdash P \quad S \text{ is incomplete} \quad \tilde{T} \leqslant \tilde{U}}{\Gamma, x : ?[\tilde{T}]\,.\,S \vdash x\,?\,[\tilde{y} : \tilde{U}]\,.\,P}\ \text{T-InSeq}$$

$$\frac{\Gamma, x : S_1 \vdash P_1 \ldots \Gamma, x : S_n \vdash P_n \quad \text{each } S_i \text{ is incomplete} \quad m \leqslant n}{\Gamma, x : \&\langle l_1 : S_1, \ldots, l_m : S_m\rangle^1 \vdash x \triangleright \{l_1 : P_1, \ldots, l_n : P_n\}}\ \text{T-Offer}$$

$$\frac{\Gamma, x : {S_i}^{t_i} \vdash P \quad S_i = \mathsf{end} \text{ or } t_i = 1}{\Gamma, x : \oplus\langle l_1 : S_1, \ldots, l_n : S_n\rangle^1 \vdash x \triangleleft l_i\ .\ P}\ \text{T-Choose}$$

$$\frac{\Gamma \vdash x : \mathsf{bool} \quad \Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash\ \mathsf{if}\ x\ \mathsf{then}\ P\ \mathsf{else}\ Q}\ \text{T-Cond} \qquad \frac{\Gamma \vdash P \quad \Gamma \text{ unlimited}}{\Gamma \vdash !P}\ \text{T-Rep}$$

**Fig. 2.** Typing rules

$$\frac{}{x\,?\,[\tilde{y} : \tilde{T}]\,.\,P \mid x\,![\tilde{z}]\,.\,Q \longrightarrow P\{\tilde{z}/\tilde{y}\} \mid Q}\ \text{R-Comm}$$

$$\frac{i \in \{1, \ldots, n\}}{x \triangleright \{l_1 : P_1, \ldots, l_n : P_n\} \mid x \triangleleft l_i\ .\ Q \longrightarrow P_i \mid Q}\ \text{R-Select}$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}\ \text{R-Par} \qquad \frac{P' \equiv P \quad P \longrightarrow Q \quad Q \equiv Q'}{P' \longrightarrow Q'}\ \text{R-Cong}$$

$$\frac{P \longrightarrow P'}{(\nu x : T)P \longrightarrow (\nu x : T)P'}\ \text{R-New}$$

$$\frac{}{\mathsf{if}\ \mathit{true}\ \mathsf{then}\ P\ \mathsf{else}\ Q \longrightarrow P}\ \text{R-True} \qquad \frac{}{\mathsf{if}\ \mathit{false}\ \mathsf{then}\ P\ \mathsf{else}\ Q \longrightarrow Q}\ \text{R-False}$$

**Fig. 3.** The reduction relation

Finally, note that a session type $T.\mathsf{end}$ effectively specifies a linear non-session channel of type $T$ as used in [9].

## 4 Operational Semantics

As usual for $\pi$-calculus-like languages, the operational semantics is defined by means of a reduction relation [11]. $P \longrightarrow Q$ means that the process $P$ reduces to the process $Q$ by executing a single communication step (or evaluating a conditional expression). The reduction relation is the smallest relation closed under the rules in Figure 3, most of which are standard. The rules R-Comm and R-Select introduce communication steps; R-Comm is standard and R-Select introduces communications which select labelled options. Note that R-Comm applies to both session and non-session channels, as there is no indication of the type of $x$ in either process.

The usual way of proving type soundness is first to prove a subject reduction theorem: if $\Gamma \vdash P$ and $P \longrightarrow Q$ then there is an environment $\Delta$ such that $\Delta \vdash Q$. Then, one proves that if $\Gamma \vdash P$ then the immediately available communications in $P$ do not cause type errors. Together these results imply that a well-typed process can be executed safely through any sequence of reduction steps. However, the presence of subtyping means that examining $\Gamma$ is not sufficient to determine what constitutes correct use of names in $P$; different occurrences of a single name in $P$ might be constrained to have different types. For example, the typed process

$$a : \widehat{\ }[![\mathsf{bool}]], b : \widehat{\ }[?[\mathsf{bool}]], x : \widehat{\ }[\mathsf{bool}] \vdash \\ a\,![x]\,.\,b\,![x]\,.\,\mathbf{0} \mid a\,?\,[y : ![\mathsf{bool}]]\,.\,P \mid b\,?\,[z : ?[\mathsf{bool}]]\,.\,Q$$

reduces in two steps to, essentially, $x : \widehat{\ }[\mathsf{bool}] \vdash P\{x/y\} \mid Q\{x/z\}$, and occurrences of $x$ in $P$ have type $![\mathsf{bool}]$ but those in $Q$ have type $?[\mathsf{bool}]$.

To address this difficulty, we adopt Pierce and Sangiorgi's technique of introducing *tagged* processes [15], written $E$, $F$, etc. instead of $P$, $Q$, etc. The syntax of tagged processes is identical to that of ordinary processes except that *all* occurrences of names are typed, for example $(x : T)\,![\tilde{x} : \tilde{U}]\,.\,E$. Structural congruence is defined on tagged processes by the same rules, with tags added, as for untagged processes. We also introduce tagged typing rules, defining judgements of the form $\Gamma \vdash E$. The tagged typing rules are essentially the same as the untagged typing rules; a typical example is the rule TT-Out.

$$\frac{\Gamma \vdash E \quad \Gamma \vdash x \leqslant T \leqslant ![\tilde{U}] \quad \tilde{V} \leqslant \tilde{U}}{\Gamma + \tilde{y} : \tilde{V} \vdash (x : T)\,![\tilde{y} : \tilde{U}]\,.\,E}\ \text{TT-Out}$$

Note that the type declared for a name in the environment must be a subtype of the type with which that name is tagged in the process.

The tagged reduction relation is written $\Gamma \vdash E \longrightarrow \Delta \vdash F$ and is defined by the rules in Figure 4, together with tagged versions of the rules R-Par, R-Cong, R-New, R-True and R-False.

$$\frac{T \leqslant ?[\tilde{U}] \quad V \leqslant ![\tilde{U}] \quad \tilde{W} \leqslant \tilde{U}}{\begin{array}{c}\Gamma, x : \hat{}[\tilde{S}] \vdash (x : T)\,?\,[\tilde{y} : \tilde{U}]\,.\,P \mid (x : V)\,!\,[\tilde{z} : \tilde{W}]\,.\,Q \\ \longrightarrow \Gamma, x : \hat{}[\tilde{S}] \vdash P\{\tilde{z}/\tilde{y}\} \mid Q\end{array}}\ \text{TR-Comm}$$

$$\frac{S \leqslant ?[\tilde{U}]\,.\,R \quad V \leqslant ![\tilde{Q}]\,.\,\overline{R} \quad \tilde{W} \leqslant \tilde{U}}{\begin{array}{c}\Gamma, x : ?[\tilde{T}]\,.\,R^2 \vdash (x : S)\,?\,[\tilde{y} : \tilde{U}]\,.\,P \mid (x : V)\,!\,[\tilde{z} : \tilde{W}]\,.\,Q \\ \longrightarrow \Gamma, x : R^2 \vdash P\{\tilde{z}/\tilde{y}\} \mid Q\end{array}}\ \text{TR-CommSeq}$$

$$\frac{m \leqslant n \quad l \in \{l_1, \ldots, l_n\}}{\begin{array}{c}\Gamma, x : \&\langle l_1 : T_1, \ldots, l_n : T_n\rangle^2 \vdash (x : S) \triangleright \{l_1 : P_1, \ldots, l_n : P_n\} \mid (x : V) \triangleleft l\,.\,Q \\ \longrightarrow \Gamma, x : T_i \vdash P_i \mid Q\end{array}}\ \text{TR-Select}$$

**Fig. 4.** The tagged reduction relation (selected rules)

The function Erase from tagged processes to untagged processes simply removes the extra type information. The definition is straightforward, for example

$$\mathrm{Erase}((x : T)\,!\,[\tilde{y} : \tilde{U}]\,.\,E) = x\,!\,[\tilde{y}]\,.\,\mathrm{Erase}(E).$$

**Theorem 1 (Tagged Subject Reduction).** *If $\Gamma \vdash E \longrightarrow \Delta \vdash F$ and $\Gamma \vdash E$ is derivable then $\Delta \vdash F$ is derivable.*

*Proof.* By induction on the derivation of $\Gamma \vdash E \longrightarrow \Delta \vdash F$. The assumption that $\Gamma \vdash E$ is derivable provides the information about the components of $F$ which is needed to build a derivation of $\Delta \vdash F$.

Observe that the Tagged Subject Reduction Theorem guarantees that the tagged reduction relation is well-defined as a relation on *derivable* tagged typing judgements.

**Lemma 1.** *If $\Gamma \vdash P$ is a derivable untagged typing judgement, then there is a tagged process $E$ such that $P = \mathrm{Erase}(E)$ and $\Gamma \vdash E$ is a derivable tagged typing judgement.*

*Proof.* We can define a function $\mathrm{Tag}_\Gamma(P)$, by induction on the structure of $P$, which essentially tags every name in $P$ with its exact type as declared in $\Gamma$ or by a binding $\nu$ or input. The presence of session types causes a slight complication: if $x : S^2 \in \Gamma$ and $x$ is used in both $P$ and $Q$, then $\mathrm{Tag}_\Gamma(P \mid Q)$ must ensure that $x$ is tagged with $S^1$ in $P$ and with $\overline{S}^1$ in $Q$, or vice versa. Essentially the same problem is encountered in linear type inference [10], and we use the same solution: $\mathrm{Tag}_\Gamma(P)$ returns a pair $(P', \Gamma')$ where $P'$ is a tagged process and $\Gamma'$ differs from $\Gamma$ only by the possible removal of some usages of session types. Then

$\mathrm{Tag}_\Gamma(P \mid Q) = (P' \mid Q', \Gamma'')$ where $\mathrm{Tag}_\Gamma(P) = (P'.\Gamma')$ and $\mathrm{Tag}_{\Gamma'}(Q) = (Q', \Gamma'')$. When $\Gamma \vdash P$ and $\mathrm{Tag}_\Gamma(P) = (P', \Gamma')$ we have that $\Gamma'$ is unlimited (so all session types have been removed), $\mathrm{ok}_\Gamma(P')$ and $P = \mathrm{Erase}(P')$.

**Theorem 2.** *If $\Gamma \vdash P$ is derivable and $\Gamma \vdash E$ is derivable and $P = \mathrm{Erase}(E)$ and $P \longrightarrow^* Q$ then there exists $\Delta \vdash F$ such that $\Gamma \vdash E \longrightarrow^* \Delta \vdash F$ and $Q = \mathrm{Erase}(F)$.*

*Proof.* By breaking the sequence of reductions into individual steps, and showing that the result holds for each step; the latter fact can be proved by induction on the derivation of the reduction step.

The Tagged Subject Reduction Theorem, Lemma 1 and Theorem 2 imply that any sequence of reductions from a well-typed untagged process can be mirrored by a sequence of reductions from a well-typed tagged process. The final theorem establishes that well-typed tagged processes do not contain any immediate possibilities for incorrect communication. It follows easily from the taggd typing rules; most of the work in proving type soundness is concentrated into the proof of the Tagged Subject Reduction Theorem. Each case of the conclusion shows that whenever a tagged process appears to contain a potential reduction, the preconditions for the relevant tagged reduction rule are satisfied and the reduction can safely be carried out.

**Theorem 3.** *If $P$ is a tagged process, $\Gamma \vdash \mathrm{Erase}(P)$ and $\mathrm{ok}_\Gamma(P)$, then*

1. *if $P \equiv (\nu\tilde{x} : \tilde{X})((a : T)\,?\,[\tilde{y} : \tilde{U}]\,.\,P_1 \mid (a : V)\,!\,[\tilde{z} : \tilde{W}]\,.\,P_2 \mid Q)$ and $T$ is not a session type then the declaration $a : S$ occurs in either $\Gamma$ or $\tilde{x} : \tilde{X}$, with $S \leqslant T \leqslant ?[\tilde{U}]$ and $S \leqslant V \leqslant ![\tilde{W}]$ and $\tilde{W} \leqslant \tilde{U}$.*
2. *if $P \equiv (\nu\tilde{x} : \tilde{X})((a : T\,.\,T')\,?\,[\tilde{y} : \tilde{U}]\,.\,P_1 \mid (a : V\,.\,V')\,!\,[\tilde{z} : \tilde{W}]\,.\,P_2 \mid Q)$ then the declaration $a : S^2$ occurs in either $\Gamma$ or $\tilde{x} : \tilde{X}$, with $\overline{S} \leqslant T\,.\,T'$ and $S \leqslant V\,.\,V'$ and $T \leqslant ![\tilde{U}]$ and $V \leqslant ?[\tilde{W}]$ and $\tilde{U} \leqslant \tilde{W}$.*
3. *if $P \equiv (\nu\tilde{x} : \tilde{X})((a : T) \triangleright \{l_1 : P_1, \ldots, l_m : P_m\} \mid (a : V) \triangleleft l\ .\ P_0 \mid Q)$ then the declaration $a : S$ occurs in either $\Gamma$ or $\tilde{x} : \tilde{X}$, with $S \leqslant T$ and $T = \&\langle l_1 : T_1, \ldots, l_n : T_n\rangle$ and $n \leqslant m$ and $\overline{S} \leqslant V$ and $V = \oplus\langle l_1 : V_1, \ldots, l_r : V_r\rangle$ and $r \leqslant n$ and $l \in \{l_1, \ldots, l_r\}$.*
4. *if $P \equiv (\nu\tilde{x} : \tilde{X})(\,\mathsf{if}\ a\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \mid Q)$ then the declaration $a : \mathsf{bool}$ occurs in either $\Gamma$ or $\tilde{x} : \tilde{X}$.*

If we take a well-typed untagged process and convert it into a tagged process, no reduction sequence can lead to a type error. Because every reduction sequence from a well-typed untagged process can be matched by a reduction sequence from a well-typed tagged process, we conclude that no type errors can result from executing a well-typed untagged process.

## 5 The POP3 Protocol

As a more substantial example, we will now use our type system to specify the POP3 protocol [13]. This protocol is typically used by electronic mail software

$$
\begin{array}{l}
A = \mu X.\&\langle\, \mathsf{quit} : \oplus\langle \mathsf{ok} : ![\mathsf{str}] \,.\, \mathsf{end}\rangle, \\
\qquad\qquad \mathsf{user} : ?[\mathsf{str}] \,.\, \oplus\langle\, \mathsf{error} : ![\mathsf{str}] \,.\, X, \\
\qquad\qquad\qquad\qquad \mathsf{ok} : ![\mathsf{str}] \,.\, \&\langle\, \mathsf{quit} : \oplus\langle \mathsf{ok} : ![\mathsf{str}] \,.\, \mathsf{end}\rangle, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{pass} : ?[\mathsf{str}] \,.\, \oplus\langle\, \mathsf{error} : ![\mathsf{str}] \,.\, X, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{ok} : ![\mathsf{str}] \,.\, T\rangle\rangle\rangle\rangle \\
T = \mu X.\&\langle\, \mathsf{stat} : \oplus\langle \mathsf{ok} : ![\mathsf{int}, \mathsf{int}] \,.\, X\rangle, \\
\qquad\qquad \mathsf{retr} : ?[\mathsf{int}] \,.\, \oplus\langle\, \mathsf{ok} : ![\mathsf{str}] \,.\, ![\mathsf{str}] \,.\, X, \\
\qquad\qquad\qquad\qquad \mathsf{error} : ![\mathsf{str}] \,.\, X\rangle, \\
\qquad\qquad \mathsf{quit} : \oplus\langle \mathsf{ok} : ![\mathsf{str}] \,.\, \mathsf{end}\rangle\rangle
\end{array}
$$

**Fig. 5.** Types for the POP3 protocol

to download new messages from a remote mailbox, so that they can be read and processed locally; it does not deal with sending messages or routing messages through a network. A POP3 server requires a client to authenticate itself by means of the user and pass commands. A client may then use commands such as stat to obtain the status of the mailbox, retr to retrieve a particular message, and quit to terminate the session. Some of these commands require additional information to be sent, for example the number of a message. We have omitted several POP3 commands from our description, but it is straightforward to fill in the missing ones.

To specify the behaviour of a channel which can be used for a POP3 session, we use the type definitions in Figure 5: $A$ describes interactions with the authentication state, and $T$ describes interactions with the transaction state. These definitions are for the server side of the channel, and we assume that there is a ground type str of strings. These definitions illustrate the complex structure possible for session types, and show the use of recursive types to describe repetitive session behaviour. The server both offers and makes choices, in contrast to the example in Section 1. After receiving a command (a label) from the client, the server can respond with either ok or error (except for the quit command, which always succeeds and does not allow an error response). The client implements an interaction of type $\overline{A}$, and therefore must offer a choice between ok and error when waiting for a response to a command. In the published description of the protocol, ok and error responses are simply strings prefixed with +OK or -ERR. This does not allow us to replace the corresponding $\oplus$ by $![\mathsf{str}]$ in the above definitions because the different continuation types after ok and error are essential for an accurate description of the protocol's behaviour. We specify a string message as well, because a POP3 server is allowed to provide extra information such as an error code.

As in Section 1 we could implement a process POP3body such that $x : A \vdash \mathsf{POP3body}$. Defining $\mathsf{POP3} = \mathsf{port}?[x : A^1].\mathsf{POP3body}$ gives $\mathsf{port} : \hat{}[A^1] \vdash \mathsf{POP3}$, which can be published as the specification of the server and its protocol.

The POP3 protocol permits an alternative authentication scheme, accessed by the apop command, in which the client sends a mailbox name and an au-

thenticating string simultaneously. This option does not have to be provided, but a server which does implement it requires a channel of type $B$, where $B$ is obtained from $A$ by adding a third option to the first choice:

$$\mathsf{apop}:?[\mathsf{str},\mathsf{str}]\,.\,\oplus\langle \mathsf{error}:![\mathsf{str}]\,.\,X, \mathsf{ok}:![\mathsf{str}]\,.\,T\rangle$$

A server which implements the apop command can be typed as follows: $\mathsf{port} : \widehat{\ }[B^1] \vdash \mathsf{newPOP3}$. Now suppose that client is a POP3 client which does not know about the apop command. As before, we have $\mathsf{client} = (\nu x : A^2)\mathsf{port}![x].\mathsf{clientbody}$ where $x : A^1 \vdash \mathsf{clientbody}$. This gives $\mathsf{port} : \widehat{\ }[A^1] \vdash \mathsf{client}$. The following derivation shows that client can also be typed in the environment $\mathsf{port} : \widehat{\ }[B^1]$, and can therefore be put in parallel with newPOP3. The key fact is that $A \leqslant B$, because the top-level options provided by $A$ are a subset of those provided by $B$.

$$\dfrac{\dfrac{x : \overline{A}^1 \vdash \mathsf{clientbody} \quad \widehat{\ }[B] \leqslant ![A]}{\mathsf{port} : \widehat{\ }[B^1], x : A^2 \vdash \mathsf{port}\,![x]\,.\,\mathsf{clientbody}}\ \text{T-Out}}{\mathsf{port} : \widehat{\ }[B^1] \vdash (\nu x : A^2)\mathsf{port}\,![x]\,.\,\mathsf{clientbody}}\ \text{T-New}$$

Space does not permit us to present the definition of POP3body, but we claim that it is simpler and more readable than an equivalent definition in conventional $\pi$-calculus or Pict. The key factor is that the session type of $x$ allows it to be used for all the messages exchanged in a complete POP3 session. Without session types, the client has to create a fresh channel every time it needs to send a message of a different type to the server; these channels also have to be sent to the server before use, which adds an overhead to every communication and therefore also to the channel types. Also in this case, the subtype relation on non-session types does not describe the relationship between interactions with POP3 and with newPOP3.

## 6 Conclusions and Future Work

We have defined a language whose type system incorporates session types, as suggested by Honda et al. [5, 20], and subtyping, based on Pierce and Sangiorgi's work [15] and extended to session types. Session channels must be controlled linearly in order to guarantee that messages go to the correct destinations, and we have adapted the work of Kobayashi et al. [9] for this purpose. Our language differs minimally from the $\pi$-calculus, the only additions being primitives for offering and making labelled choices. Unlike Honda et al. we do not introduce special syntax for establishing and manipulating session channels; everything is taken care of by the typing rules. We have advocated using a session type as part of the published specification of a server's protocol, so that static type-checking can be used to verify that client implementations behave correctly. Using the POP3 protocol as an example, we have shown that subtyping increases the utility of this idea: if upgrading a server causes its protocol to have a session type which is a supertype of its original session type, then existing client implementations are still type-correct with respect to the new server.

Session types have some similarities to the types for active objects studied by Nierstrasz [14] and Puntigam [18, 19]. Both incorporate the idea of a type which specifies a sequence of interactions. The main difference seems to be that in the case of active objects, types can specify interdependencies between interactions on different channels. However, the underlying framework (concurrent objects with method invocation, instead of channel-based communication between processes) is rather different, and we have not yet made a detailed comparison of the two systems.

The present paper is the first report of our work on a longer term project to investigate advanced type systems in the context of the Pict [16] programming language. Our next goal is to extend the Pict compiler to support the type system presented here. Because Pict is based on the *asynchronous* $\pi$-calculus [4, 6, 1] the output prefixing of our language will have to be encoded by explicitly attaching a continuation channel to each message. Initially we will work with a non-polymorphic version of Pict; later, after more theoretical study of the interplay between session types and polymorphism, we will integrate session types with the full Pict type system including polymorphism. The Pict compiler uses a powerful partial type inference technique [17], and it will be interesting to see how it can be extended to handle session types. Because of the value of explicit session types as specifications, we might not want to allow the programmer to omit them completely; however, automatic inference of, for example, some usage annotations will probably be very useful.

The implementation will allow us to gain more experience of programming with sessions, which in turn should suggest other typing features which can usefully be added to the system—for example, it would be interesting to consider Kobayashi's type system [7, 8] for partial deadlock-freedom.

**Acknowledgements**

Malcolm Hole is funded by the EPSRC project "Novel Type Systems for Concurrent Programming Languages" (GR/L75177). Simon Gay is partially funded by the same EPSRC project, and also by a grant from the Nuffield Foundation. We thank the anonymous referees for their comments and suggestions. Paul Taylor's proof tree macros were used in the production of this paper.

## References

[1] G. Boudol. Asynchrony and the $\pi$-calculus (note). Rapporte de Recherche 1702, INRIA Sofia-Antipolis, May 1992.

[2] S. J. Gay. A sort inference algorithm for the polyadic $\pi$-calculus. In *Proceedings, 20th ACM Symposium on Principles of Programming Languages*. ACM Press, 1993.

[3] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[4] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proceedings of the European Conference on Object-Oriented Programming*, LNCS. Springer-Verlag, 1994.

[5] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of the European Symposium on Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[6] K. Honda and N. Yoshida. Combinatory representation of mobile processes. In *Proceedings, 21st ACM Symposium on Principles of Programming Languages*, 1994.

[7] N. Kobayashi. A partially deadlock-free typed process calculus. In *Proceedings, Twelfth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1997.

[8] N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems*, 20:436–482, 1998.

[9] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. In *Proceedings, 23rd ACM Symposium on Principles of Programming Languages*, 1996.

[10] I. Mackie. Lilac : A functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):1–39, October 1994.

[11] R. Milner. The polyadic $\pi$-calculus: A tutorial. Technical Report 91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1991.

[12] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–77, September 1992.

[13] J. Myers and M. Rose. Post office protocol version 3, May 1996. Internet Standards RFC1939.

[14] O. Nierstrasz. Regular types for active objects. *ACM Sigplan Notices*, 28(10):1–15, October 1993.

[15] B. Pierce and D. Sangiorgi. Types and subtypes for mobile processes. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1993.

[16] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press, 1998.

[17] B. C. Pierce and D. N. Turner. Local type inference. In *Proceedings, 25th ACM Symposium on Principles of Programming Languages*, 1998.

[18] F. Puntigam. Synchronization expressed in types of communication channels. In *Proceedings of the European Conference on Parallel Processing (Euro-Par'96)*, volume 1123 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[19] F. Puntigam. Coordination requirements expressed in types for active objects. In M. Aksit and S. Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[20] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *Proceedings of the 6th European Conference on Parallel Languages and Architectures*, number 817 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[21] D. N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1996.