Subtyping Supports Safe Session Substitution

Simon J. Gay[⋆]

School of Computing Science, University of Glasgow, UK
Simon.Gay@glasgow.ac.uk

Abstract. Session types describe the structure of bi-directional point-to-point communication channels by specifying the sequence and format of messages on each channel. A session type defines a communication protocol. Type systems that include session types are able to statically verify that communication-based code generates, and responds to, messages according to a specified protocol. It is natural to consider subtyping for session types, but the literature contains conflicting definitions. It is now folklore that one definition is based on safe substitutability of channels, while the other is based on safe substitutability of processes. We explain this point in more detail, and show how to unify the two views.

0 Prologue

My first encounter with Phil Wadler was at the Marktoberdorf Summer School in 1992, where he lectured on monads in functional programming. His course had the characteristically punning title "Church and State: taming effects in functional languages", and during his first lecture he removed his shirt to reveal a T-shirt printed with the category-theoretic axioms for a monad. The following year I met him again, at POPL in Charleston, South Carolina. He presented the paper "Imperative Functional Programming", also covering the monadic approach and co-authored with Simon Peyton Jones, which in 2003 won the award for the most influential paper from POPL 1993.

At that time, in the early 1990s, Phil was a professor at the University of Glasgow, and I was a PhD student at Imperial College London, attending POPL to present my own very first paper, which was about types for pi calculus. We PhD students in the Imperial theory group had a view of the world in which a small number of other departments were admired for their theoretical research, and the rest were ignored. The Glasgow functional programming group was admired, as was the LFCS in Edinburgh, where Phil now works. More than twenty years later, I myself am a professor at the University of Glasgow, but when I arrived here as a lecturer in 2000, Phil had already departed for the USA. When he returned to Scotland to take a chair in theoretical computer science

[⋆] Supported by EPSRC (EP/K034413/1 "From Data Types to Session Types: A Basis for Concurrency and Distribution" and EP/L00058X/1 "Exploiting Parallelism through Type Transformations for Hybrid Manycore Systems") and COST Action IC1201 "Behavioural Types for Reliable Large-Scale Software Systems".

at Edinburgh, we started the Scottish Programming Languages Seminar, which is still active. Eventually the opportunity for a technical collaboration arose, when Phil developed an interest in session types because of the Curry-Howard correspondence between session types and linear logic, discovered by Luís Caires and Frank Pfenning. This led to a successful EPSRC grant application by Phil, myself, and Nobuko Yoshida, which is still running.

I am delighted to contribute this paper to Phil's 60th birthday Festschrift.

1 Introduction

Session types were introduced by Honda et al. (1994; 1998), based on earlier work by Honda (1993), as a way of expressing communication protocols type-theoretically, so that protocol implementations can be verified by static type-checking. The original formulation, now known as binary session types, covers pairwise protocols on bidirectional point-to-point communication channels. Later, the theory was extended to multiparty session types (Honda et al., 2008), in order to account for collective protocols among multiple participants.

Gay and Hole (1999, 2005) extended the theory of session types by adding subtyping, to support more flexible interaction between participants in a protocol. Many papers now include subtyping. However, not all of them use the same definition as Gay and Hole; some papers give the opposite variance to the session type constructors. The alternative definition first appears in the work of Carbone et al. (2007) and has been used in many papers by Honda et al., including one by Demangeon and Honda (2011) whose main focus is subtyping. For example, consider two simple session types with branching (external choice), one which offers a choice between labels a and b, and one which offers label a alone. According to Gay and Hole's definition, the session type constructor for branch is covariant in the set of labels, so that $\&\langle \mathsf{a} : \mathsf{end} \rangle \leqslant \&\langle \mathsf{a} : \mathsf{end}, \mathsf{b} : \mathsf{end} \rangle$. According to the Honda et al. definition, branch is contravariant, so that $\&\langle \mathsf{a} : \mathsf{end}, \mathsf{b} : \mathsf{end} \rangle \sqsubseteq \&\langle \mathsf{a} : \mathsf{end} \rangle$. Each paper defines a type system and an operational semantics and proves that typability guarantees runtime safety. How can both definitions of subtyping be correct?

The fundamental way of justifying the definition of subtyping for any language is by Liskov and Wing's (1994) concept of safe substitutability. Type T is a subtype of type U if a value of type T can be safely used whenever a value of type U is expected, where "safely" means without violating the runtime safety property that the type system guarantees. In this paper we will analyse safe substitutability in the setting of session types, to see how each definition of subtyping can be justified. The answer lies in careful consideration of what is being safely substituted. Gay and Hole's definition is based on safe substitution of communication channels, whereas Honda et al.'s definition is based on safe substitution of processes. This situation contrasts with that of λ-calculus, where the syntax consists only of expressions and there is no choice about which kind of entity is being substituted.

The remainder of the paper is structured as follows. Section 2 reviews the concepts of session types and motivates the need for subtyping. Section 3 considers safe substitution of channels, and gives several justifications of the Gay and Hole definition. Section 4 considers safe substitution of processes, and justifies the Honda et al. definition. Section 5 describes a session type system for higher-order communication, introduced by Mostrous and Yoshida (2007, 2015), and shows how that setting allows the two definitions of subtyping to be reconciled. Finally, Section 6 concludes. In order to reduce the volume of definitions, we focus on intuition and explanation, and do not give full details of the languages and type systems that we discuss.

2 Session Types and Subtyping

To illustrate the need for a theory of subtyping for session types, consider the example of a server for mathematical operations (Gay and Hole, 2005). There are two services: addition of integers, which produces an integer result, and testing of integers for equality, which produces a boolean result. A client and a server run independently and communicate on a bidirectional point-to-point channel, which has two endpoints, one for the client and one for the server. The protocol is specified by defining a pair of dual session types, one for each endpoint. The type of the server's endpoint is S, defined as follows.

$$\begin{aligned} S = \&\langle\ & \mathsf{add} : ?[\mathsf{int}].?[\mathsf{int}].![\mathsf{int}].\mathsf{end}, \\ & \mathsf{eq} : ?[\mathsf{int}].?[\mathsf{int}].![\mathsf{bool}].\mathsf{end}\ \rangle \end{aligned}$$

The $\&\langle \dots \rangle$ constructor specifies that a choice is offered between, in this case, two options, labelled add and eq. It is an external choice, often called "branch": the server must be prepared to receive either label. Each label leads to a type describing the continuation of the protocol, different in each case. The constructors $?[\dots]$ and $![\dots]$ indicate receiving and sending, respectively. Sequencing is indicated by $.$ and end marks the end of the interaction. For simplicity this protocol allows only a single service invocation.

In a run of the protocol, the first message is one of the labels plus and eq, sent from the client to the server. Subsequently, the client sends messages according to the continuation of the label that was chosen, and then receives a message from the server. The type of the client's endpoint is $\overline{S}$, the dual of S.

$$\begin{aligned} \overline{S} = \oplus\langle\ & \mathsf{add} : ![\mathsf{int}].![\mathsf{int}].?[\mathsf{int}].\mathsf{end}, \\ & \mathsf{eq} : ![\mathsf{int}].![\mathsf{int}].?[\mathsf{bool}].\mathsf{end}\ \rangle \end{aligned}$$

The structure is the same, but the directions of the communications are reversed. The $\oplus\langle \dots \rangle$ constructor is an internal choice, often called "select" or "choice": the client can decide which label to send.

Suppose that the server is upgraded by the addition of new services and an extension of an existing service. The equality test can now handle floating point numbers, and there is a multiplication service. The type of the new server is S'.

$$\begin{aligned} S' = \&\langle\ & \mathsf{add}:?[\mathsf{int}].?[\mathsf{int}].![\mathsf{int}].\mathsf{end}, \\ & \mathsf{mul}:?[\mathsf{int}].?[\mathsf{int}].![\mathsf{int}].\mathsf{end}, \\ & \mathsf{eq}:?[\mathsf{float}].?[\mathsf{float}].![\mathsf{bool}].\mathsf{end}\ \rangle \end{aligned}$$

The requirement for a theory of subtyping is to allow a client that has been typechecked with type $\overline{S}$ to interact with the new server. In Sections 3 and 4 we will discuss the two approaches to subtyping that have appeared in the literature, and explain how they account for this example.

Another possibility is to upgrade the type of the addition service to

$$\mathsf{add}:?[\mathsf{float}].?[\mathsf{float}].![\mathsf{float}].\mathsf{end}.$$

This motivates the development of a theory of bounded polymorphism (Gay, 2008) that allows the addition service to have type

$$\mathsf{add}(X \leqslant \mathsf{float}):?[X].?[X].![X].\mathsf{end}$$

so that the client chooses an instantiation of the type variable X as either int or float and then proceeds appropriately. We do not consider bounded polymorphism further in the present paper.

3 Channel-Oriented Subtyping

In Section 2 we defined session types for the mathematical server and client, without choosing a language in which to implement them. We will now give pi calculus definitions, following the example of Gay and Hole (2005). The core of the server process is $\mathsf{serverbody}$, parameterised by its channel endpoint x.

$$\begin{aligned} \mathsf{serverbody}(x) = x \triangleright \{\ & \mathsf{add}:x?[u{:}\mathsf{int}].x?[v{:}\mathsf{int}].x![u+v].\mathbf{0}, \\ & \mathsf{eq}:x?[u{:}\mathsf{int}].x?[v{:}\mathsf{int}].x![u=v].\mathbf{0}\ \}. \end{aligned}$$

The branching construct $x \triangleright \{\ldots\}$ corresponds to the type constructor $\&\langle\ldots\rangle$; it receives a label and executes the code corresponding to that label. The rest of the syntax is pi calculus extended with arithmetic operations. The process $x?[u{:}\mathsf{int}].P$ receives an integer value on channel x, binds it to the name u, and then executes P. The process $x![u+v].Q$ sends the value of the expression $u+v$ on channel x and then executes Q. Finally, $\mathbf{0}$ is the terminated process.

The type system derives judgements of the form $\Gamma \vdash P$, where Γ is an environment of typed channels and P is a process. A process does not have a type; it is either correctly typed or not. The typing of $\mathsf{serverbody}$ is

$$x:S \vdash \mathsf{serverbody}(x)$$

where S is the session type defined in Section 2. Although we are not showing the typing rules here, it should be clear that the process and the type have the same communication structure.

Similarly, the core of one possible client process, which uses the addition service, is clientbody, again parameterised by its channel endpoint.

$$\mathsf{clientbody}(x) = x \triangleleft \mathsf{add}.x![2].x![3].x?[u{:}\mathsf{int}].\mathbf{0}$$

The typing is

$$x : \overline{S} \vdash \mathsf{clientbody}(x).$$

The client and server need to establish a connection so that they use opposite endpoints c^- and c^+ of channel c. The original formulation of session types (Takeuchi et al., 1994) used matching request and accept constructs, but here we follow Gay and Hole's approach in which the client uses standard pi calculus name restriction to create a fresh channel and then sends one endpoint to the server. This requires a globally known channel a of type $\widehat{\ }[S]$ (the standard pi calculus channel type constructor). The complete definitions and typings are

$$\begin{aligned}
\mathsf{server}(a) &= a?[y{:}S].\mathsf{serverbody}(y)\\
\mathsf{client}(a) &= (\nu x : S)(a![x^+].\mathsf{clientbody}(x^-))\\
a{:}\widehat{\ }[S] &\vdash \mathsf{server}(a)\\
a{:}\widehat{\ }[S] &\vdash \mathsf{client}(a)
\end{aligned}$$

where $(\nu x : S)$ binds x^+ with type S and x^- with type $\overline{S}$.

In Section 2 we defined the session type S' for an upgraded server. The corresponding definition of newserverbody is

$$\begin{aligned}
\mathsf{newserverbody}(x) = x \triangleright \{\ &\mathsf{add} : x?[u{:}\mathsf{int}].x?[v{:}\mathsf{int}].x![u+v].\mathbf{0},\\
&\mathsf{mul} : x?[u{:}\mathsf{int}].x?[v{:}\mathsf{int}].x![u*v].\mathbf{0},\\
&\mathsf{eq} : x?[u{:}\mathsf{float}].x?[v{:}\mathsf{float}].x![u=v].\mathbf{0}\ \}.
\end{aligned}$$

and we have $x : S' \vdash \mathsf{newserverbody}(x)$. Now newserver is defined in terms of newserverbody in the same way as before.

Clearly client can interact safely with newserver; the question is how to extend the type system so that

$$a{:}\widehat{\ }[S'] \vdash \mathsf{client}(a) \mid \mathsf{newserver}(a).$$

In Gay and Hole's theory, this is accounted for by considering safe substitution of channels. It is safe for newserverbody to interact on a channel of type S instead of the channel of type S' that it expects; the substitution means that newserverbody never receives label mul and always receives integers in the eq service, because the process at the other endpoint of the channel is using it with type $\overline{S}$. Subtyping is defined according to this concept of safe substitution, giving $S \leqslant S'$. We see that branch (external choice) is covariant in the set of labels, and input is covariant in the message type. If we imagine changing the definitions so that the server creates the channel and sends one endpoint to the client, we find that select (internal choice) is contravariant in the set of labels and output is contravariant in the message type. These are the variances established by Pierce and Sangiorgi (1996) in the first work on subtyping for (non-session-typed) pi

$$\frac{}{\mathsf{end} \leqslant \mathsf{end}}\ \text{S-END} \qquad \frac{T \leqslant U \qquad U \leqslant T}{\ulcorner[T] \leqslant \ulcorner[U]}\ \text{S-CHAN}$$

$$\frac{T \leqslant U \qquad V \leqslant W}{?[T].V \leqslant ?[U].W}\ \text{S-INS} \qquad \frac{m \leqslant n \qquad \forall i \in \{1,\ldots,m\}.S_i \leqslant T_i}{\&\langle\, l_i : S_i \,\rangle_{1\leqslant i\leqslant m} \leqslant \&\langle\, l_i : T_i \,\rangle_{1\leqslant i\leqslant n}}\ \text{S-BRANCH}$$

$$\frac{U \leqslant T \qquad V \leqslant W}{![T].V \leqslant ![U].W}\ \text{S-OUTS} \qquad \frac{m \leqslant n \qquad \forall i \in \{1,\ldots,m\}.S_i \leqslant T_i}{\oplus\langle\, l_i : S_i \,\rangle_{1\leqslant i\leqslant n} \leqslant \oplus\langle\, l_i : T_i \,\rangle_{1\leqslant i\leqslant m}}\ \text{S-CHOICE}$$

Fig. 1. Subtyping for non-recursive types.

calculus. Furthermore, it is clear that all of the session type constructors are covariant in the continuation type, simply because after the first communication we can again consider safe substitutability with the continuation types in the substituting and substituted positions. The definition of subtyping for non-recursive session types is summarised in Figure 1, which includes the invariant rule for standard channel types. For recursive types, the same principles are used as the basis for a coinductive definition (Gay and Hole, 2005).

This definition of subtyping is justified by the proof of type safety in Gay and Hole's system. In the rest of this section, we give alternative technical justifications for the definition.

3.1 Safe substitutability of channels, formally

Gay and Hole express safe substitutability of channels for channels by (a more general form of) the following result, in which z^p is either z^+ or z^-.

Lemma 1 (Substitution (Gay and Hole, 2005, Lemma 8)). *If* $\Gamma, w\!:\!W \vdash P$ *and* $Z \leqslant W$ *and* $z^p \notin dom(\Gamma)$ *then* $\Gamma, z^p\!:\!Z \vdash P\{z^p/w\}$.

They define subtyping first, taking into account recursive types by giving a coinductive definition based on the principles of Figure 1, and then prove type safety, for which Lemma 1 is a necessary step. Alternatively, we can take Lemma 1 as a statement of safe substitutability, i.e. a specification of the property that we want subtyping to have, and then derive the definition of subtyping by considering how to prove the lemma by induction on typing derivations. For example, in the inductive case for an input process, the relevant typing rule is T-INS. The occurrence of subtyping in the rule is based on safe substitutability of channels: the received value of type T must be substitutable for the bound variable y of type U.

$$\frac{\Gamma, x^p{:}S, y{:}U \vdash P \qquad T \leqslant U}{\Gamma, x^p{:}?[T].S \vdash x^p?[y{:}U].P}\ \text{T-INS}$$

The specific case in the proof of the lemma is that we substitute z^p for x in the process $x?[y{:}U].Q$. The typing derivation for $x?[y{:}U].Q$ concludes with an application of rule T-InS.

$$\frac{\Gamma, x{:}S, y{:}U \vdash Q \qquad T \leqslant U}{\Gamma, x{:}?[T].S \vdash x?[y{:}U].Q}\ \text{T-InS}$$

To prove this case of Lemma 1 we need to establish $\Gamma, z^p : Z \vdash z^p?[y{:}U].Q\{z^p/x\}$, which has to have a derivation ending with an application of rule T-InS:

$$\frac{\Gamma, z^p{:}S', y{:}U \vdash Q\{z^p/x\} \qquad A \leqslant U}{\Gamma, z^p{:}?[A].S' \vdash z^p?[y{:}U].Q\{z^p/x\}}\ \text{T-InS}$$

From the assumptions $Z \leqslant ?[T].S$ and $T \leqslant U$, we need to be able to conclude $Z = ?[A].S'$ and $S' \leqslant S$ (to use the induction hypothesis) and $A \leqslant U$. To go from $T \leqslant U$ to $A \leqslant U$, for arbitrary U, requires $A \leqslant T$. Overall, in order to make the proof of Lemma 1 work, we need:

if $Z \leqslant ?[T].S$ then $Z = ?[A].S'$ with $A \leqslant T$ and $S' \leqslant S$.

This is essentially one of the clauses in the coinductive definition of subtyping (Gay and Hole, 2005, Definition 3); the only difference is that the coinductive definition unfolds recursive types.

As another example we can consider the case of branch, with the typing rule T-Offer. It contains the raw material of subtyping in the form of the condition $m \leqslant n$, which guarantees that the received label is within the range of possibilities.

$$\frac{m \leqslant n \quad \forall i \in \{1, \ldots, m\}.(\Gamma, x^p{:}S_i \vdash P_i)}{\Gamma, x^p{:}\&\langle\, l_i : S_i \,\rangle_{1\leqslant i\leqslant m} \vdash x^p \triangleright \{\, l_i : P_i \,\}_{1\leqslant i\leqslant n}}\ \text{T-Offer}$$

The relevant case in the proof of Lemma 1 is that we substitute z^p for x in the process $x \triangleright \{\, l_i : P_i \,\}_{1\leqslant i\leqslant n}$. The typing derivation for $x \triangleright \{\, l_i : P_i \,\}_{1\leqslant i\leqslant n}$ concludes with an application of rule T-Offer.

$$\frac{m \leqslant n \quad \forall i \in \{1, \ldots, m\}.(\Gamma, x{:}S_i \vdash P_i)}{\Gamma, x{:}\&\langle\, l_i : S_i \,\rangle_{1\leqslant i\leqslant m} \vdash x \triangleright \{\, l_i : P_i \,\}_{1\leqslant i\leqslant n}}\ \text{T-Offer}$$

We need to establish $\Gamma, z^p{:}Z \vdash z^p \triangleright \{\, l_i : P_i\{z^p/x\} \,\}_{1\leqslant i\leqslant n}$, whose derivation must end with an application of T-Offer:

$$\frac{r \leqslant n \quad \forall i \in \{1, \ldots, r\}.(\Gamma, z^p{:}Z_i \vdash P_i)}{\Gamma, z^p{:}\&\langle\, l_i : Z_i \,\rangle_{1\leqslant i\leqslant r} \vdash z^p \triangleright \{\, l_i : P_i \,\}_{1\leqslant i\leqslant n}}\ \text{T-Offer}$$

From the assumptions $Z \leqslant \&\langle\, l_i : S_i \,\rangle_{1\leqslant i\leqslant m}$ and $m \leqslant n$ we need to be able to conclude $Z = \&\langle\, l_i : Z_i \,\rangle_{1\leqslant i\leqslant r}$ and $\forall i \in \{1 \ldots r\}.Z_i \leqslant S_i$ and $r \leqslant n$. Therefore we need:

if $Z \leqslant \&\langle\, l_i : S_i \,\rangle_{1\leqslant i\leqslant m}$ then $Z = \&\langle\, l_i : Z_i \,\rangle_{1\leqslant i\leqslant r}$ with $r \leqslant m$ and $\forall i \in \{1 \ldots r\}.Z_i \leqslant S_i$.

$$\begin{aligned}
[\![\mathsf{end}]\!] &= \mathsf{cap}_\emptyset[\,] \\
[\![?[T].S]\!] &= \mathsf{cap}_\mathsf{i}[[\![T]\!], [\![S]\!]] \\
[\![![T].S]\!] &= \mathsf{cap}_\mathsf{o}[[\![T]\!], [\![\overline{S}]\!]] \\
[\![\&\langle l_1 : S_1, \ldots, l_n : S_n\rangle]\!] &= \mathsf{cap}_\mathsf{i}[\langle l_1 : [\![S_1]\!], \ldots, l_n : [\![S_n]\!]\rangle] \\
[\![\oplus\langle l_1 : S_1, \ldots, l_n : S_n\rangle]\!] &= \mathsf{cap}_\mathsf{o}[\langle l_1 : [\![\overline{S_1}]\!], \ldots, l_n : [\![\overline{S_n}]\!]\rangle]
\end{aligned}$$

Fig. 2. Translation of session types into linear and variant types (Dardha et al., 2012). Data types such as int and bool are translated into themselves.

Again this is essentially a clause in the coinductive definition of subtyping. The other clauses are obtained similarly by considering other possibilities for the structure of P, with the exception of the clause for end which requires proving (straightforwardly) that $\Gamma, x{:}\mathsf{end} \vdash P$ if and only if $\Gamma \vdash P$ and $x \notin \mathit{fn}(P)$. In order to type as many processes as possible we take the largest relation satisfying this clause, which leads to the coinductive definition.

3.2 Channel-oriented subtyping by translation

Kobayashi (2002) observed informally that the branch and choice constructors of session types can be represented by variant types, which have been considered in pi calculus independently of session types (Sangiorgi and Walker, 2001). Specifically, making a choice corresponds to sending one label from a variant type, and offering a choice corresponds to a case-analysis on a received label. With this representation, Gay and Hole's definition of subtyping follows by combining the standard definition of subtyping for variants in λ-calculus (Pierce, 2002) with Pierce and Sangiorgi's (1996) definition of subtyping for input and output types in pi calculus. Dardha et al. (2012) developed this idea in detail[1]. They showed that not only subtyping, but also polymorphism, can be derived from a translation of session types into linear pi calculus (Kobayashi et al., 1999) with variants.

The translation combines three ideas. First, in the linear pi calculus, a linear channel can be used for only one communication. To allow a session channel to be used for a series of messages, a continuation-passing style is used, so that each message is accompanied by a fresh channel which is used for the next message. Second, the linear pi calculus separates the capabilities for input (cap_i) and output (cap_o) on a channel. A session type corresponding to an initial input or output is translated into an input-only or output-only capability. Polyadic channel types of the form $\mathsf{cap}_\mathsf{i}[T_1, \ldots, T_n]$ are used. Third, branch and select session types are both translated into variant types $\langle l_1 : T_1, \ldots, l_n : T_n\rangle$, with the input or output capability as appropriate. The translation of types is defined in Figure 2; note the use of the empty capability $\mathsf{cap}_\emptyset$ in the translation of end.

[1] See also Dardha's (2014) PhD thesis.

To illustrate the translation, consider the session types S and $\overline{S}$ from Section 2.

$$\begin{aligned}
[\![S]\!] = \mathsf{cap}_{\mathsf{i}}[\langle\ & \mathsf{add} : \mathsf{cap}_{\mathsf{i}}[\mathsf{int}, \mathsf{cap}_{\mathsf{i}}[\mathsf{int}, \mathsf{cap}_{\mathsf{o}}[\mathsf{int}, \mathsf{cap}_{\emptyset}[\,]\ ]\ ]\ ]\\
& \mathsf{eq} : \mathsf{cap}_{\mathsf{i}}[\mathsf{int}, \mathsf{cap}_{\mathsf{i}}[\mathsf{int}, \mathsf{cap}_{\mathsf{o}}[\mathsf{bool}, \mathsf{cap}_{\emptyset}[\,]\ ]\ ]\ ]\ \rangle\ ]
\end{aligned}$$

$$\begin{aligned}
[\![\overline{S}]\!] = \mathsf{cap}_{\mathsf{o}}[\langle\ & \mathsf{add} : \mathsf{cap}_{\mathsf{i}}[\mathsf{int}, \mathsf{cap}_{\mathsf{i}}[\mathsf{int}, \mathsf{cap}_{\mathsf{o}}[\mathsf{int}, \mathsf{cap}_{\emptyset}[\,]\ ]\ ]\ ]\\
& \mathsf{eq} : \mathsf{cap}_{\mathsf{i}}[\mathsf{int}, \mathsf{cap}_{\mathsf{i}}[\mathsf{int}, \mathsf{cap}_{\mathsf{o}}[\mathsf{bool}, \mathsf{cap}_{\emptyset}[\,]\ ]\ ]\ ]\ \rangle\ ]
\end{aligned}$$

Duality appears as a reversal of input/output capability at the top level only, so that the message types of $[\![S]\!]$ and $[\![\overline{S}]\!]$ are the same.

The standard definition of subtyping for variant types is covariant in the set of labels. With the definitions in Figure 2, considering that $\mathsf{cap}_{\mathsf{i}}$ is covariant and $\mathsf{cap}_{\mathsf{o}}$ is contravariant, this gives the same variances for branch and select that Gay and Hole define.

3.3 Channel-oriented subtyping by semantic subtyping

Castagna et al. (2009) use the idea of semantic subtyping (Frisch et al., 2002) to develop a set-theoretic model of session types in which subtyping is simply set inclusion. The foundation for their model is a duality relation $\bowtie$ on session types, which characterises safe communication. Gay and Hole generalised the duality function $\overline{(\cdot)}$ to a coinductively-defined relation $\perp$ (in order to handle recursive types) which requires exact matching of label sets and message types. For example, with the definitions in Section 2, we have $S \perp \overline{S}$ but not $S' \perp \overline{S}$. In contrast, $S' \bowtie \overline{S}$ because every message sent on an endpoint of type $\overline{S}$ will be understood by the process at the opposite endpoint of type S', and vice versa. Deriving the definition of subtyping from this definition of duality is technically more complicated because of the circularity that the definition of duality itself involves subtyping of message types, which in general can be session types. When the theory is worked out, the session type constructors have the same variances as in Figure 1. The paper does not discuss the distinction between channel-oriented and process-oriented subtyping, but the fact that the definition of duality includes subtyping on message types, which can be channel types, implies that substitutability of channels is being considered.

4 Process-Oriented Subtyping

We consider the paper by Demangeon and Honda (2011) as a typical example that defines subtyping in the Honda et al. style. Like Dardha et al., they study subtyping for session types by translating into a simpler language. Apart from the definition of subtyping, the main difference is that their linear pi calculus combines variant types and input/output types so that a label is accompanied by a message. The intuition behind the Honda et al. definition of subtyping is easiest to understand by considering the type environment to be the type of a process, which we emphasize here by writing judgements in the form $P \vdash \mathit{\Gamma}$. Their subsumption result has a clear analogy with subtyping for λ-calculus.

Here, the relation $\sqsubseteq$ is the subtyping relation on session types, lifted pointwise to environments.

Proposition 1 (Subsumption (Demangeon and Honda, 2011, Proposition 9)). *If* $P \vdash \Gamma$ *and* $\Gamma \sqsubseteq \Gamma'$ *then* $P \vdash \Gamma'$.

Considering the type environment as a specification of the interactions that process must allow, we can return to the maths server to see that all of the following typings make sense.

$$\begin{aligned} \mathsf{serverbody}(x) &\vdash x\!:\!S \\ \mathsf{newserverbody}(x) &\vdash x\!:\!S' \\ \mathsf{newserverbody}(x) &\vdash x\!:\!S \end{aligned}$$

In the first two judgements, the type and the process match exactly. The third judgement states that the process with more behaviour, newserverbody, satisfies the requirement expressed by the original type S, i.e. it provides the add and eq services. In relation to subtyping, and Proposition 1, this means $S' \sqsubseteq S$, i.e.

$$\left.\begin{array}{l} \&\langle\ \mathsf{add}\!:\!?[\mathsf{int}].?[\mathsf{int}].![\mathsf{int}].\mathsf{end}, \\ \quad \mathsf{mul}\!:\!?[\mathsf{int}].?[\mathsf{int}].![\mathsf{int}].\mathsf{end}, \\ \quad \mathsf{eq}\!:\!?[\mathsf{float}].?[\mathsf{float}].![\mathsf{bool}].\mathsf{end}\ \rangle \end{array}\right\} \sqsubseteq \left\{\begin{array}{l} \&\langle\ \mathsf{add}\!:\!?[\mathsf{int}].?[\mathsf{int}].![\mathsf{int}].\mathsf{end}, \\ \quad \mathsf{eq}\!:\!?[\mathsf{int}].?[\mathsf{int}].![\mathsf{bool}].\mathsf{end}\ \rangle \end{array}\right.$$

This is the opposite of the subtyping relationship for these types in Gay and Hole's theory, which we described in Section 3. Indeed, Demangeon and Honda define subtyping for branch types to be contravariant in the set of labels, and subtyping for select types is covariant. The example above also shows that input needs to be contravariant in the message type and output needs to be covariant, so that the subtyping relation is exactly the converse of Gay and Hole's definition. We emphasise that branch and input both involve receiving a value, so it makes sense that they have the same variance; similarly select and output. Demangeon and Honda actually define subtyping for input, which in their presentation is unified with branch, to be *covariant* in the message type, and conversely subtyping for output, which they unify with select, is *contravariant.* It seems that this is just a typographical error, as other papers in the Honda et al. style, for example by Chen et al. (2014), use the expected variance in the message type. In some definitions of subtyping, for example by Mostrous and Yoshida (2015), the variance depends on whether data or channels are being communicated; the technical details need further investigation to clarify this point.

Demangeon and Honda give further justification for their definition of subtyping by relating it to duality with the following result, stated here in a form that combines their Definition 10 and Proposition 12.

Proposition 2. $S_1 \sqsubseteq S_2$ *if and only if for all* S, $S_1 \bowtie S \Rightarrow S_2 \bowtie S$.

In words: if a process satisfies type S_1, then it also satisfies type S_2 if and only if all safe interactions with S_1 are also safe interactions with S_2.

The view of the session type environment as the specification of a process, and consideration of when one process satisfies the specification originally given for another process, correspond to working with safe substitution of processes, instead of safe substitution of channels as in Section 3. Explicitly, consider the following typing derivation for a server and client in parallel; we ignore the step of establishing the connection, because in Demangeon and Honda's system this is achieved by a symmetrical request/accept construct rather than by sending a channel endpoint.

$$\frac{\mathsf{serverbody}(x^+) \vdash x^+ : S \qquad \mathsf{clientbody}(x^-) \vdash x^- : \overline{S}}{\mathsf{serverbody}(x^+) \mid \mathsf{clientbody}(x^-) \vdash x^+ : S, x^- : \overline{S}}$$

Because we also have $\mathsf{newserverbody}(x^+) \vdash x^+ : S$, the process $\mathsf{newserverbody}(x^+)$ can be safely substituted for $\mathsf{serverbody}(x^+)$ in the derivation, i.e. in the typed system.

It should be possible to prove a formal result about safe substitutability of processes, along the following lines, where $C[\cdot]$ is a process context.

Conjecture 1. If $P \vdash \Gamma_P$ and $Q \vdash \Gamma_Q$ and $\Gamma_Q \sqsubseteq \Gamma_P$ and $C[P] \vdash \Gamma$ then $C[Q] \vdash \Gamma$.

5 Unifying Channel- and Process-Oriented Subtyping

Sections 3 and 4 described two separate type systems, with two definitions of subtyping. To better understand the relationship between these definitions, we use a language and type system in which both channel substitution and process substitution can be expressed internally, by message-passing. This requires higher-order communication, i.e. the ability to send a process as a message.

A natural way to develop a higher-order process calculus is to require a process, before being sent as a message, to be made self-contained by being abstracted on all of its channels. When a process is received, it can be applied to the channels that it needs, and then will start executing. This approach is realistic for distributed systems, as it does not assume that channels are globally available. Therefore we need a system that combines pi calculus and λ-abstraction, with session types. This combination has been studied by Mostrous and Yoshida (2015). We will now informally discuss typings in such a system.

In our presentation of Honda et al. style subtyping (Section 4) we used typing judgements of the form $P \vdash \Gamma$. We will now write this as

$$P \vdash \mathsf{proc}(\Gamma)$$

to emphasise the view of Γ as the type of P. In our presentation of Gay and Hole subtyping (Section 3) we used typing judgements of the form $\Gamma \vdash P$. We will now write this as

$$\Gamma \vdash P : \mathsf{proc}$$

to emphasise that P is being given the type which says that it is a correct process.

In Mostrous and Yoshida's language of higher-order processes we can abstract a process on its channels to obtain an expression with a function type, for example

$$\vdash \lambda x.\mathsf{server}(x) : S \to \mathsf{proc}$$

and

$$\vdash \lambda x.\mathsf{newserver}(x) : S' \to \mathsf{proc}.$$

Now we can identify $\mathsf{proc}(x : S)$ with $S \to \mathsf{proc}$. If we use Gay and Hole's subtyping relation, so that $S \leqslant S'$, then the standard definition of subtyping for function types gives $\mathsf{proc}(x{:}S') \leqslant \mathsf{proc}(x : S)$, which is Honda et al.'s definition. In other words, by moving to a setting in which safe substitution of both channels and processes can be expressed in terms of message-passing, we can explain the difference between Honda et al. subtyping and Gay/Hole subtyping by the fact that subtyping on function types is contravariant in the parameter.

6 Conclusion

We have discussed two definitions of subtyping for session types, both justified in terms of safe substitutability. The first definition, due to Gay and Hole, is based on safe substitutability of channels and has been derived in several ways in the literature. The second definition, due to Honda et al., is based on safe substitutability of processes. We have shown that the two definitions can be reconciled in a session type system for higher-order processes, of the kind defined by Mostrous and Yoshida. This is achieved by identifying the type of a process (its session environment, à la Demangeon and Honda) with a function type that abstracts all of its channels. The Honda et al. definition, in which branch types are contravariant, therefore arises from the combination of the Gay and Hole definition, in which branch types are covariant, and the standard definition of subtyping for function types, which is contravariant in the parameter.

We have used an informal presentation in order to focus on intuition and reduce the number of definitions being quoted from the literature. A more technically detailed account, including a proof of Conjecture 1, will be left for a future longer paper.

Acknowledgements

I am grateful to Nobuko Yoshida for discussion of the origin of the Honda et al. definition of subtyping, and to the anonymous referees for their comments.

References

Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: Proceedings of the 16th European Symposium on Programming Languages and Systems (ESOP). pp. 2–17. Lecture

Notes in Computer Science, Springer (2007), http://dx.doi.org/10.1007/978-3-540-71316-6_2

Castagna, G., Dezani-Ciancaglini, M., Giachino, E., Padovani, L.: Foundations of session types. In: Proceedings of the 11th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP). pp. 219–230. ACM (2009), http://doi.acm.org/10.1145/1599410.1599437

Chen, T., Dezani-Ciancaglini, M., Yoshida, N.: On the preciseness of subtyping in session types. In: Proceedings of the 16th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP). pp. 135–146. ACM (2014), http://doi.acm.org/10.1145/2643135.2643138

Dardha, O.: Type Systems for Distributed Programs: Components and Sessions. PhD thesis, University of Bologna (2014), https://tel.archives-ouvertes.fr/tel-01020998

Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. In: Proceedings of the 14th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP). pp. 139–150. ACM (2012)

Demangeon, R., Honda, K.: Full abstraction in a subtyped pi-calculus with linear types. In: Proceedings of the 22nd International Conference on Concurrency Theory (CONCUR). Lecture Notes in Computer Science, vol. 6901, pp. 280–296. Springer (2011), http://dx.doi.org/10.1007/978-3-642-23217-6_19

Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping. In: Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS). pp. 137–146. IEEE (2002), http://dx.doi.org/10.1109/LICS.2002.1029823

Gay, S.J.: Bounded polymorphism in session types. Mathematical Structures in Computer Science 18(5), 895–930 (2008), http://dx.doi.org/10.1017/S0960129508006944

Gay, S.J., Hole, M.J.: Types and subtypes for client-server interactions. In: Proceedings of the European Symposium on Programming Languages and Systems (ESOP). Lecture Notes in Computer Science, vol. 1576, pp. 74–90. Springer (1999)

Gay, S.J., Hole, M.J.: Subtyping for session types in the pi calculus. Acta Informatica 42(2/3), 191–225 (2005)

Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 273–284. ACM (2008)

Honda, K.: Types for dyadic interaction. In: Proceedings of the International Conference on Concurrency Theory (CONCUR). Lecture Notes in Computer Science, vol. 715, pp. 509–523. Springer (1993)

Honda, K., Vasconcelos, V., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Proceedings of the European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 1381, pp. 122–138. Springer (1998)

Kobayashi, N.: Type systems for concurrent programs. In: Formal Methods at the Crossroads: From Panacea to Foundational Support (Proceedings of UNU/IIST 10th Anniversary Colloquium). Lecture Notes in Computer Science, vol. 2757, pp. 439–453. Springer (2002), extended version at www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf

Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the Pi-Calculus. ACM Transactions on Programming Languages and Systems 21(5), 914–947 (1999)

Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems 16(6), 1811–1841 (1994)

Mostrous, D., Yoshida, N.: Two session typing systems for higher-order mobile processes. In: Proceedings of the 8th International Conference on Typed Lambda Calculi and Applications (TLCA). Lecture Notes in Computer Science, vol. 4583, pp. 321–335. Springer (2007), http://dx.doi.org/10.1007/978-3-540-73228-0_23

Mostrous, D., Yoshida, N.: Session typing and asynchronous subtyping for the higher-order π-calculus. Information and Computation 241, 227–263 (2015), http://dx.doi.org/10.1016/j.ic.2015.02.002

Pierce, B.C., Sangiorgi, D.: Typing and subtyping for mobile processes. Mathematical Structures in Computer Science 6(5), 409–454 (1996)

Pierce, B.C.: Types and Programming Languages. MIT Press (2002)

Sangiorgi, D., Walker, D.: The π-calculus: a Theory of Mobile Processes. Cambridge University Press (2001)

Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: Proceedings of Parallel Architectures and Languages Europe (PARLE). Lecture Notes in Computer Science, vol. 817, pp. 398–413. Springer (1994)