Session Types as Generic Process Types

Simon J. Gay[1] Nils Gesbert[1] António Ravara[2]

13th May 2008

1 Introduction

Session types [7, 4] are an increasingly popular technique for specifying and verifying protocols in concurrent and distributed systems. In a setting of point-to-point private-channel-based communication, the session type of a channel describes the sequence and type of messages that can be sent on it. For example

$$\& \langle \mathsf{service} : !\,[\mathsf{int}]\,.\,?\,[\mathsf{bool}]\,.\,\mathsf{end}, \mathsf{quit} : \mathsf{end}\rangle$$

describes the server's view of a channel on which a client can select either service or quit. In the former case, the client then sends an integer and receives a boolean; in the latter case, the protocol ends. From the client's viewpoint, the channel has a dual type in which the direction of messages is reversed:

$$\oplus \langle \mathsf{service} : ?\,[\mathsf{int}]\,.\,!\,[\mathsf{bool}]\,.\,\mathsf{end}, \mathsf{quit} : \mathsf{end}\rangle$$

Session types provide concise specifications of protocols and allow certain properties of protocol implementations to be verified by static typechecking. Originally formulated for languages similar to pi-calculus, type systems incorporating session types have now been defined for other language paradigms including object-oriented languages [2] and service-oriented systems [1].

The theory of session types was developed in order to analyse a particular correctness criterion for concurrent systems: that every message is of the type expected by the receiver, and that whenever a client selects a service, the server offers a matching service. Igarashi and Kobayashi [5] have taken a different approach to type-theoretic specifications of concurrent systems, by developing a single generic type system for the pi-calculus from which numerous specific type systems can be obtained by varying certain parameters. Their motivation is to express the common aspects of a range of type systems, enabling much of the work of designing typing rules and proving type soundness to be packaged into a general theory instead of being worked out for each case. In the generic type system, types are abstractions of processes, so that the typing rules display a very direct correspondence between the structure of processes and the structure of types. There is also a subtyping relation, which can be modified in order to obtain specific type systems; this allows, for example, a choice of retaining

[1]Department of Computing Science, University of Glasgow
[2]SQIG at Instituto de Telecomunicações and Departamento de Matemática, IST, Technical University of Lisbon

or discarding information about the order of communications. A logic is provided in which to define an *ok* predicate that is interpreted both as a desired runtime property of processes and as a correctness condition for typings. This double interpretation allows a generic type soundness theorem to be proved, but means that type checking becomes more like model checking unless the specific subtyping relation can be exploited to yield an efficient type checking algorithm.

Kobayashi [6, Section 10] has stated that the generic type system (GTS) subsumes session types, although without presenting a specific construction. The purpose of the present paper is to clarify the relationship. This is relevant to the design of programming languages for distributed systems. For example: if we want an object-oriented language with static typing of protocols, is it better to work directly with session types or to develop an object-oriented formulation of GTS? However, the present paper considers pi-calculus so that we can study a precise question about two type systems for essentially the same language.

Kobayashi did not explain what it means for GTS to "subsume" session types. We interpret it as defining a translation $[\![\cdot]\!]$ from processes and type environments in the source language into GTS, satisfying as many of the following conditions as possible. (1) $[\![P]\!]$ should have a similar structure to P. (2) There should be a correspondence in both directions between the operational semantics, ideally $P \longrightarrow Q$ if and only if $[\![P]\!] \longrightarrow [\![Q]\!]$. (3) There should be a correspondence in both directions between typing derivations, ideally $\Gamma \vdash P$ if and only if $[\![\Gamma]\!] \triangleright [\![P]\!]$. (4) Type soundness for session types should follow from the generic type soundness theorem.

We take the source language to be the version of session types defined by Gay and Hole [3]. This version does not include the accept/request primitives [7, 4] and does not consider progress properties [2]. We remove recursive types, for simplicity, and make some changes to the structural congruence relation, to remove inessential differences compared with GTS. Three key issues remain. First, translating the polarities in the source language: x^+ and x^- refer to the two endpoints of channel x. Second, translating the labels used in branching and selection (external and internal choice). Third, obtaining a correspondence between subtyping in the source language and the subtyping relation which is always present in GTS. The present paper focuses on the first two points and does not discuss subtyping. We satisfy conditions (1–4) above in some form, although the details are more complicated.

From now on we refer to the source language as *session processes* and the target language as *generic processes*.

2 Translation

Processes and types. The languages share common process constructors (inaction, parallel composition, scope restriction[1], and replication), differing basically in two ways. In session processes, (1) channels are decorated with *polarities* (absent in generic processes), and processes only synchronise if the subjects have complementary polarities; (2) there are constructors for *branch*, an input labelled external choice, and *select*, to choose a branch of the choice. Generic processes instead have mixed guarded sums (but no labels), and input and output actions are decorated with *events* (taken from a countable set).

[1] Following a suggestion by Kobayashi, we have added a type annotation to ν in GTS.

$$
\begin{array}{rcl}
\text{Common Syntax } C & ::= & \mathbf{0} \mid (P_1 \mid P_2) \mid *P \\
\text{Source Language } P & ::= & C \mid (\nu x : S)\, P \mid x^p?[y] \,.\, P \mid x^p![y^q] \,.\, P \quad (x^p \neq y^q) \\
& & \mid x^p \triangleright \{l_i : P_i\}_{i=1}^n \mid x^p \triangleleft l \,.\, P \\
\text{polarities } p & ::= & + \mid - \mid \varepsilon \\
\text{Session Types } S & ::= & \mathtt{end} \mid ?[S_1] \,.\, S_2 \mid ![S_1] \,.\, S_2 \mid \& \langle l_i : S_i \rangle_{i=1}^n \mid \oplus \langle l_i : S_i \rangle_{i=1}^n \\
\text{Target Language } P & ::= & C \mid (\nu \tilde{x} : \tau)\, P \mid \textstyle\sum_{i=1}^n G_i \\
\text{guarded processes } G & ::= & x![\tilde{y}] \,.\, P \mid x?[\tilde{y}] \,.\, P \\
\text{Generic Types } \Gamma & ::= & \mathbf{0} \mid \textstyle\sum_{i=1}^n \gamma_i \mid (\Gamma_1 \mid \Gamma_2) \mid (\Gamma_1 \,\&\, \Gamma_2) \\
\text{guarded types } \gamma & ::= & x![\tau] \,.\, \Gamma \mid x?[\tau] \,.\, \Gamma \\
\text{tuple types } \tau & ::= & (\tilde{x})\Gamma
\end{array}
$$

Figure 1: Syntax

Since these tags are only relevant for liveness properties like deadlock-freedom, which we do not address in this work, we omit them.

Consider x, y from a countable set of *channels*, disjoint from a **finite** set of N *labels*, ranged over by l, l_i. The grammars in Figure 1 define the languages of both sessions and generic processes and types. Session processes are monadic (for simplicity), while generic processes are polyadic (as required by the encoding). A session type environment is a finite mapping from polarised channels to session types, $\Delta = x_1^{p_1} : S_1; \ldots; x_n^{p_n} : S_n$. A generic type environment is a process type Γ. We write $\prod_{i=1}^n P_i$ for $P_1 \mid \cdots \mid P_n$ and $\bigwedge_{i=1}^n \Gamma_i$ for $\Gamma_1 \& \cdots \& \Gamma_n$.

We consider the operational semantics of both languages based on a reduction relation. However, for session processes, instead of the original structural congruence relation, we take the structural preorder of the generic processes.

Encoding processes and type environments. For each input-guarded labelled sum the translation creates a new name for every possible label, sends them all to the channel subject of the input, and waits in an input-guarded sum where the subjects correspond to the labels in the original process and a fresh channel is received for the continuation of the protocol. Output selection is encoded dually.

Polarities distinguish between the two endpoints of a channel; communication only occurs between x^+ and x^-. Erasing polarities would translate processes that do not reduce into processes that reduce. However, it is possible to solve this problem by inserting some type information into the translated process. But when translating scope restriction, one cannot prefix the body of the process, otherwise there would be no meaningful operational correspondence. Therefore, for each source channel we introduce a pair of target channels, one for each polarity, and a forwarder between them, which is in parallel with the translated process.

Let σ be a numbering of the labels from 1 to N. For any channels p, m, the rules in Figure 2 inductively define the forwarder from p to m following the structure of a session type S. Consider the translation homomorphic for common processes. The rules in Figure 3 inductively define the translation $[\![P]\!]_\varphi^\Gamma$ of the session process P into a generic process, where φ is an injective

$$
\begin{aligned}
\textsc{fw}\,(p, m, \mathsf{end}) &= \mathbf{0} \\
\textsc{fw}\,(p, m, ?\,[S_1] \,.\, S_2) &= m?\,[z] \,.\, p\,!\,[z] \,.\, \textsc{fw}\,(p, m, S_2) \\
\textsc{fw}\,(p, m, !\,[S_1] \,.\, S_2) &= p?\,[z] \,.\, m\,!\,[z] \,.\, \textsc{fw}\,(p, m, S_2) \\
\textsc{fw}\,(p, m, \& \,\langle l_i : S_i \rangle_{i=1}^n) &= p?\,[\kappa_1 \ldots \kappa_N] \,.\, (\nu \lambda_1 \ldots \lambda_N)\, m\,!\,[\tilde{\lambda}] \,. \\
&\qquad \textstyle\sum_{i=1}^n \lambda_{\sigma(l_i)}?\,[m'] \,.\, (\nu p' : (\!|S_i \mid \overline{S_i}|\!))\, \kappa_{\sigma(l_i)}\,!\,[p'] \,.\, \textsc{fw}\,(p', m', S_i) \\
\textsc{fw}\,(p, m, \oplus \,\langle l_i : S_i \rangle_{i=1}^n) &= m?\,[\kappa_1 \ldots \kappa_N] \,.\, (\nu \lambda_1 \ldots \lambda_N)\, p\,!\,[\tilde{\lambda}] \,. \\
&\qquad \textstyle\sum_{i=1}^n \lambda_{\sigma(l_i)}?\,[p'] \,.\, (\nu m' : (\!|S_i \mid \overline{S_i}|\!))\, \kappa_{\sigma(l_i)}\,!\,[m'] \,.\, \textsc{fw}\,(p', m', S_i)
\end{aligned}
$$

Figure 2: Definition of the forwarder

$$
\begin{aligned}
[\![x^p\,!\,[y^q] \,.\, P]\!]_\varphi^{\Gamma, x^p : ![S_1].S_2} &= \varphi(x^p)\,!\,[\varphi(y^q)] \,.\, [\![P]\!]_\varphi^{\Gamma, x^p : S_2} \\
[\![x^p?\,[y] \,.\, P]\!]_\varphi^{\Gamma, x^p : ?[S_1].S_2} &= \varphi(x^p)?\,[z] \,.\, [\![P]\!]_{\varphi + \{y^\varepsilon \mapsto z\}}^{\Gamma, x^p : S_2} \\
[\![x^p \triangleleft l \,.\, P]\!]_\varphi^{\Gamma, x^p : \oplus\langle l : S, \ldots\rangle} &= \varphi(x^p)?\,[\lambda_{1\ldots N}] \,.\, (\nu z : (\!|S \mid \overline{S}|\!))\, \lambda_{\sigma(l)}\,!\,[z] \,.\, [\![P]\!]_{\varphi + \{x^p \mapsto z\}}^{\Gamma, x^p : S} \\
[\![x^p \triangleright \{l_i : P_i\}]\!]_\varphi^{\Gamma, x^p : \&\langle l_i : P_i\rangle} &= (\nu \lambda_{1\ldots N})\, \varphi(x^p)\,!\,[\tilde{\lambda}] \,.\, \textstyle\sum_{i=1}^n \lambda_{\sigma(l_i)}?\,[z] \,.\, [\![P_i]\!]_{\varphi + \{x^p \mapsto z\}}^{\Gamma, x^p : S_i} \\
[\![(\nu x : S)\, P]\!]_\varphi^\Gamma &= (\nu p, m : (p, m))([\![p : S]\!] \mid [\![m : \overline{S}]\!] \mid [\![p : \overline{S}]\!] \mid [\![m : S]\!])) \\
&\qquad (\textsc{fw}\,(p, m, S) \;\mid\; [\![P]\!]_{\varphi + \{x^+ \mapsto p; x^- \mapsto m\}}^{\Gamma, x^+ : S, x^- : \overline{S}})
\end{aligned}
$$

Figure 3: Process translation

mapping from the free polarised channels in P to channels of generic processes and Γ is a session type environment such that $\Gamma \vdash P$ (we omit φ and Γ when not relevant). The rules in Figure 4 inductively define the translation of the session type environment Γ to a generic type environment, where φ is an injective mapping from $\mathrm{dom}(\Gamma)$ to the set of channels. Let $(\!|S|\!)$ stand for $(z)[\![y : S]\!]_{y \mapsto z}$. The last rule uses the notion of *dual session type* (denoted $\overline{S}$), which exchanges inputs and outputs, and branch and selection [3].

3 Results

We state operational and typing correspondences.

Theorem 1. *For any well-typed closed session process P, whenever $P \longrightarrow Q$, then $[\![P]\!] \longrightarrow^n [\![Q]\!]$ with $n = 2$ or 4, depending on whether the reduction step is a communication or a selection.*

The extra steps are due to the forwarders and the encoding of labels. We require the source process to be well-typed to ensure that the forwarders behave correctly. There is also a reverse correspondence. It is complicated to state correctly, because a forwarder adds a one-place buffer to the synchronous communication of the pi-calculus; some deadlocked processes in the session calculus can, when translated, take one reduction step. In future, we would like to obtain a full abstraction result with respect to some behavioural equivalence.

$$
\begin{aligned}
[\![x_1^{p_1} : S_1, \ldots, x_n^{p_n} : S_n]\!]_\varphi &= [\![x_1^{p_1} : S_1]\!]_\varphi \mid \cdots \mid [\![x_n^{p_n} : S_n]\!]_\varphi \\
[\![x^p : \mathtt{end}]\!]_\varphi &= \mathbf{0} \\
[\![x^p : ?[S_1] \,.\, S_2]\!]_\varphi &= \varphi(x^p)?[(\!|S_1|\!)] \,.\, [\![x^p : S_2]\!]_\varphi \\
[\![x^p : ![S_1] \,.\, S_2]\!]_\varphi &= \varphi(x^p)![(\!|S_1|\!)] \,.\, [\![x^p : S_2]\!]_\varphi \\
[\![x^p : \& \langle l_i : S_i \rangle_{i=1}^n]\!]_\varphi &= \varphi(x^p)!\left[(\lambda_1, \ldots, \lambda_N) \bigwedge_{i=1}^n \lambda_{\sigma(l_i)}![(\!|S_i|\!)]\right] \\
[\![x^p : \oplus \langle l_i : S_i \rangle_{i=1}^n]\!]_\varphi &= \varphi(x^p)?\left[(\lambda_1, \ldots, \lambda_N) \bigwedge_{i=1}^n \lambda_{\sigma(l_i)}!\left[(\!|\overline{S_i}|\!)\right]\right]
\end{aligned}
$$

Figure 4: Type environment translation

Typing correspondence. Let the subtyping relation of the generic type system be such that: (1) Sequential information about the communications on different channels is removed (Igarashi and Kobayashi's SUB-DIVIDE rule); (2) subtyping can occur beneath a prefix (input or output); (3) type $\mathbf{0}$ is a subtype of $*\mathbf{0}$; (4) the sum operator is idempotent. Moreover, let $ok(\Gamma)$ hold if and only if Γ is well-formed (meaning that whenever a communication is possible, the type sent is a subtype of the one expected by the receiver) and either: Γ reduces in one step to $\mathbf{0}$; or for any free variable x of Γ, there exists a session type S such that $\Gamma \downarrow \{x\} \geq [\![x : S]\!] \mid [\![x : \overline{S}]\!]$, where $\geq$ is the subtyping relation and $\Gamma \downarrow \{x\}$ is the restriction of Γ to x (that is Γ with all actions whose subject is not x removed). The first condition ensures a correct use of labels in a branching/selection, and the second one deals with regular session channels. Note that $ok(\Gamma)$ is stable by reductions, meaning that if $\Gamma \longrightarrow \Gamma'$ and $ok(\Gamma)$ hold, then $ok(\Gamma')$ holds. This makes ok a proper consistency predicate.

Lemma 1. $[\![p : \overline{S}]\!] \mid [\![m : S]\!] \triangleright \mathrm{FW}\,(p, m, S)$

Theorem 2 (Completeness)**.** *For any session process P and any corresponding φ, if $\Delta \vdash P$ then $[\![\Delta]\!]_\varphi \triangleright [\![P]\!]_\varphi^\Delta$.*

The *ok* predicate is checked in the typing rule for ν, so this theorem implies that bound channels in $[\![P]\!]$ are used consistently.

The reverse direction is more difficult to state. GTS types more processes, for the following reason. Sending a message of type `end` in the session calculus removes the channel from the sender's environment, but in GTS, sending a message of type $\mathbf{0}$ does not remove any capabilities.

Theorem 3 (Soundness)**.** *Let P be a closed session process. If $[\![P]\!]$ is well-typed in the generic type system and no type annotation of P is* `end`*, then P is well-typed as a session process.*

Since the conditions of the generic type soundness theorem hold in this setting, we also conclude the desired runtime safety property of session types.

4 Conclusion

We have defined a translation from a system of session types for the π-calculus into Igarashi and Kobayashi's generic type system (GTS). We have proved correspondence results between process reductions in the two systems, and between

typing derivations; we can also apply the generic type soundness theorem. The translation clarifies the relationship between session types and GTS, and provides an interesting application of GTS. Because GTS can also represent more complex behavioural properties including deadlock-freedom, embedding session types into it may suggest ways of extending session types with such properties.

In our opinion, despite the translation into GTS, session types themselves remain of great interest for programming language design, for several reasons. First, session types are a high-level abstraction for structuring inter-process communication [7]; preservation of this abstraction and the corresponding programming primitives is very important for high-level programming. Second, there is now a great deal of interest in session types for languages other than the π-calculus. Applying GTS would require either translation into π-calculus, obscuring distinctive programming abstractions, or extension of GTS to other language, which might not be easy. Third, proofs of type soundness for session types are conceptually fairly straightforward, even when these are liveness properties, as is frequently the case. The amount of work saved by using the generic type soundness theorem is relatively small. Fourth, for practical languages we are very interested in typechecking algorithms for session types; GTS does not yield an algorithm automatically, so specific algorithms for session types need to be developed in any case.

Acknowledgements. António Ravara is partially supported by the Portuguese Fundação para a Ciência e a Tecnologia (via SFRH/BSAB/757/2007 and project Space-Time-Types, POSC/EIA/55582/2004), by FEDER, by the EU IST proactive initiative FET-Global Computing (project Sensoria, IST–2005–16004), and by the EPSRC grant EP/F037368/1 "Behavioural types for object-oriented languages". Nils Gesbert is supported by the EPSRC grant EP/F065708/1 "Engineering Foundations of Web Services: Theories and Tool Support". Simon Gay is partially supported by both EPSRC grants. We are grateful to Luís Caires, Kohei Honda, and Naoki Kobayashi for useful comments and discussions.

References

[1] M. Carbone, K. Honda, and N. Yoshida. Structured global programming for communication behaviour. *ESOP*, LNCS 4421:2–17, 2007.

[2] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. *ECOOP*, LNCS 4067:328–352, 2006.

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

[4] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. *ESOP*, LNCS 1381:122–138, 1998.

[5] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1–3):121–163, 2004.

[6] N. Kobayashi. Type systems for concurrent programs. *Formal Methods at the Crossroads*, LNCS 2757:439–453, 2002. Extended version at `www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf`.

[7] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. *PARLE*, LNCS 817:398–413, 1994.