HAL
open science

Session Types as Generic Process Types

Simon J. Gay, Nils Gesbert, António Ravara

To cite this version:

Simon J. Gay, Nils Gesbert, António Ravara. Session Types as Generic Process Types. Proceedings of the Combined 21st International Workshop on Expressiveness in Concurrency and 11th Workshop on Structural Operational Semantics (EXPRESS/SOS)., Sep 2014, Rome, Italy. pp.94 - 110, 10.4204/EPTCS.160.9 . hal-01102349

HAL Id: hal-01102349

https://inria.hal.science/hal-01102349v1

Submitted on 12 Jan 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Session Types as Generic Process Types

Simon J. Gay
School of Computing Science, University of Glasgow, UK

Nils Gesbert
Grenoble INP – Ensimag, France

António Ravara
CITI and Dep. de Informática, FCT, Universidade Nova de Lisboa, Portugal

Behavioural type systems ensure more than the usual safety guarantees of static analysis. They are based on the idea of "types-as-processes", providing dedicated type algebras for particular properties, ranging from protocol compatibility to race-freedom, lock-freedom, or even responsiveness. Two successful, although rather different, approaches, are session types and process types. The former allows to specify and verify (distributed) communication protocols using specific type (proof) systems; the latter allows to infer from a system specification a process abstraction on which it is simpler to verify properties, using a generic type (proof) system. What is the relationship between these approaches? Can the generic one subsume the specific one? At what price? And can the former be used as a compiler for the latter? The work presented herein is a step towards answers to such questions. Concretely, we define a stepwise encoding of a π-calculus with sessions and session types (the system of Gay and Hole [4]) into a π-calculus with process types (the Generic Type System of Igarashi and Kobayashi [6]). We encode session type environments, polarities (which distinguish session channels end-points), and labelled sums. We show forward and reverse operational correspondences for the encodings, as well as typing correspondences. To faithfully encode session subtyping in process types subtyping, one needs to add to the target language record constructors and new subtyping rules. In conclusion, the programming convenience of session types as protocol abstractions can be combined with the simplicity and power of the π-calculus, taking advantage in particular of the framework provided by the Generic Type System.

1 Introduction

This work is a contribution to the understanding of the relationship between two different type disciplines for concurrent processes, aiming as well at compiling one into the other (considering thus the specific session constructors as π macros).

Session types [5, 12] 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

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

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:

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

Session types provide concise specifications of protocols and allow certain properties of protocol implementations to be verified by static type-checking. The theory of session types was developed in order to

J. Borgström, S. Crafa (Eds.): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2014)
EPTCS 160, 2014, pp. 94–110, doi:10.4204/EPTCS.160.9

© S. J. Gay, N. Gesbert, and A. Ravara
This work is licensed under the Creative Commons Attribution License.

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.

The generic type system (GTS, from now on) of Igarashi and Kobayashi [6] is a different approach to type-theoretic specifications of concurrent systems: from a single generic type system for the π-calculus [10, 11], inferring a generic type abstracting the behaviour of the process, it is possible to enforce specific properties 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 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.

GTS vs. session types. There is no doubt today of the usefulness of session types: expressing protocols as type abstractions and verifying statically their implementations is very relevant in a society of ubiquitous computing. Process types, instead, are somehow more "low-level" and are supported by powerful type systems able of ensuring a wider range of behavioural properties. Therefore, it would be beneficial to use process types as an executable intermediate language for sessions-based ones.

Kobayashi [7, Section 10] has stated that GTS subsumes session types, although without presenting a specific construction or giving a precise technical meaning to the term "subsumes". It is clearly possible, within GTS, to define a type system similar to session types in the sense that types specify certain allowed sequences of messages. A somewhat different question, which we aim to answer in the present paper, is whether a specific existing system of session types can be reproduced within GTS.

Related work. Kobayashi's paper also defines an encoding of π-calculus with session types into π-calculus with a linear type system [8] and record types, and observes that with this encoding, subtyping for session types [4] arises from the standard subtyping rules for records. That encoding is interesting: the target language is the π-calculus with additional process constructs that are a good match for the branching and selection operators of session constructors and types.

Dardha *et al.* [1] show an operational and a typing correspondence, proving the correctness of Kobayashi's encoding, and illustrate its robustness by testing it with respect to session subtyping, polymorphism and higher-order communication. Concretely, the target language is the π-calculus with boolean and variant values and with a case constructor; the types contain linear, variant, and product types. Session type duality is captured by passing opposite capabilities in linear channel types, and the linearised (or linearly sequential) behaviour of a session is enforced by passing in each communication a fresh channel where the subsequent communications of the session should take place. Not surprisingly, branching is encoded using the case constructor and branch and select types are encoded using variant types. Naturally, session subtyping coincides with variant subtyping.

Demangeon and Honda [2] study a subtyping theory for a π-calculus augmented with branch and select constructors (obtaining thus a fully abstract encoding of a session calculus). In the case of this

work, the source and target languages are quite close.

How to encode session types in process types, using GTS, is less obvious; the types are more general and powerful, and the process language is simpler — just synchronous π, *i.e.*, has no extra features. Our aim is two relate the two different behavioural type disciplines as they were defined, without adding extra features to the target language. This is relevant to the design of programming languages for distributed systems. For example: if one wants to design a distributed object-oriented language with static typing of protocols, can one work directly with session types instead of developing an object-oriented formulation of process types, and "compile" (features of) the language into GTS?

Results. We assume that "GTS subsumes session types" means defining a translation $[\![\cdot]\!]$ from processes and type environments in the source language into GTS, satisfying as many of the (usual) following conditions as possible: (1) the encoding function should be compositional; (2) $[\![P]\!]$ should have a similar structure to P (the encoding should be uniform); (3) there should be a correspondence in both directions between the operational semantics, ideally $P \longrightarrow Q$ if and only if $[\![P]\!] \longrightarrow [\![Q]\!]$; (4) there should be a correspondence in both directions between typing derivations, ideally $\Gamma \vdash P$ if and only if $[\![\Gamma]\!] \triangleright [\![P]\!]$; (5) type soundness for session types follows from the GTS type soundness theorem.

Defining a translation between the languages, enjoying the above properties, is the aim of this paper. We start by introducing the calculi under study: the next section presents the source and the target languages (syntax and operational semantics), and Section 3 presents the types and type systems of both languages. Then, we address three key issues: (1) translating the polarities in the source language: x^+ and x^- refer to the two endpoints of channel x (Section 4); (2) translating the labels used in branching and selection— external and internal choice (Section 5); (3) comparing the subtyping disciplines of the source and the target languages (Section 6). Finally, Section 7 concludes the paper, summarising the achievements, contributions, and future work.

For all encodings hold forward and reverse operational correspondences, as well as typing correspondences. The encodings respect the properties identified above, do not restrict neither the source nor the target language, and the operational correspondences are as much as possible independent of the type system. Due to space limitations, we do not present herein the failed attempts to define the translations with the envisaged properties, nor proofs (which, however, are all straightforward inductions over reduction rules and typing rules; the difficulty was in formulating the definitions).

2 Languages

We take the source language to be the version of session types defined by Gay and Hole [4]. This language uses π-calculus ν to establish sessions, instead of special $\mathsf{accept}/\mathsf{request}$ primitives [5, 12], and does not consider progress properties [3]. Also, we remove recursive types, for simplicity, and we make some changes to the structural congruence relation, to avoid inessential differences compared with GTS. From now on we refer to the source language — the polarized monadic π-calculus — as *session processes* and to the target language — the polyadic π-calculus — as *generic processes*. The source language is monadic, for simplicity, but the target language is polyadic by demand of the encoding. [1]

The languages share several common process constructors: inaction, parallel composition, scope restriction, and replication. There are two differences. First, in session processes, channels are decorated with *polarities*, which are absent from generic processes, and processes only synchronise when channel

[1] The syntax and the (static and dynamic) semantics of the source and target languages are taken from [4] and [6].

Choice-free source language (synchronous monadic π-calculus with polarities):

$$\begin{array}{rl} \textit{Choice-Free Session Processes} & H,J ::= \mathbf{0} \mid (H \mid J) \mid (\nu x)H \mid *H \mid x^p?[y].H \mid x^p![y^q].H \\ \textit{Polarities} & p,q ::= + \mid - \mid \varepsilon \end{array}$$

Full source language: adds *branching* and *selection*.

$$\textit{Session Processes} \quad P,Q ::= \mathbf{0} \mid (P \mid Q) \mid (\nu x)P \mid *P \mid x^p?[y].P \mid x^p![y^q].P \mid x^p \triangleright \{l_i : P_i\}_{i\in I} \mid x^p \triangleleft l.P$$

Full target language (synchronous polyadic π-calculus with mixed guarded sums):

$$\begin{array}{rl} \textit{Generic Processes} & P ::= \mathbf{0} \mid (P \mid Q) \mid (\nu x)P \mid *P \mid \sum_{i\in I} G_i \\ \textit{Guarded Processes} & G ::= x![\tilde{y}].P \mid x?[\tilde{y}].P \end{array}$$

Figure 1: Syntax of the source and the target languages

names have complementary polarities. Second, session processes have 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. Since these tags are only relevant for properties like deadlock-freedom, which we do not address herein, we omit them.

Syntax. Consider x,y,z from a countable set $\mathscr{V}$ of *channels*. Assume that I is a non-empty finite indexing set. The source and target languages are inductively defined by the grammars in Figure 1. As usual, $\tilde{x}$ abbreviates a sequence $x_1 \cdots x_n$ and $\nu\tilde{x}$ abbreviates $\nu x_1 \cdots \nu x_n$, for some $n \geq 0$.

To illustrate the session language, we present a simple example that we incrementally develop, using it throughout the paper to also clarify aspects of the encodings. To have a more "realistic" example, we assume available in the languages basic numerical values, boolean expressions and boolean types. These are not in the syntax (which is minimal, following the principle of Occam's razor), but can be straightforwardly added without affecting the encodings.

The following code, parameterised on channel x, implements the service branch of the server referred to in the introduction:

$$\mathsf{serviceBody}(x) \stackrel{\text{def}}{=} x?[i].x![i=3].\mathbf{0}$$

The complete server, containing code to implement both service and quit, is below (the code for quit is trivial).

$$\mathsf{server}(x) \stackrel{\text{def}}{=} x \triangleright \{\mathsf{service} : \mathsf{serviceBody}(x),\ \ \mathsf{quit} : \mathbf{0}\} \tag{1}$$

A client first selects one of the two options — in this case, service — and then follows the corresponding protocol. The definition is parameterised on channel x.

$$\mathsf{client}(x) \stackrel{\text{def}}{=} x \triangleleft \mathsf{service}.x![3].x?[b].\mathbf{0}$$

In a complete system, the client and server will be instantiated with opposite endpoints of a common channel:

$$\mathsf{server}(c^+) \mid \mathsf{client}(c^-)$$

Common rules (for session and generic processes):

$$\frac{P \to Q}{(\nu x)\,P \to (\nu x)\,Q}\ \text{(R-RES)} \qquad \frac{P \to Q}{(P \mid R) \to (Q \mid R)}\ \text{(R-PAR)}$$

$$\frac{P \preceq P' \qquad P' \to Q' \qquad Q' \preceq Q}{P \to Q}\ \text{(R-SP)}$$

Consider a *duality* function on polarities, defined as follows: $\overline{+} = -$, $\overline{-} = +$, and $\overline{\varepsilon} = \varepsilon$.
Channel communication in session processes:

$$(x^p?[z]\,.\,P \mid x^{\overline{p}}![y^q]\,.\,Q) \;\to\; (Q\,[y^q/z] \mid Q) \quad \text{with } p,q \neq \varepsilon \quad \text{(R-S-COM)}$$

Labelled communication in session processes:

$$(x^p \triangleright \{l_i : P_i\}_{i\in I} \mid x^{\overline{p}} \triangleleft l_k\,.\,P) \;\to\; (P_k \mid P) \quad \text{if } k \in I \text{ and with } p \neq \varepsilon \quad \text{(R-S-COMLAB)}$$

Communication in generic processes:

$$((\cdots + x?[\tilde{z}]\,.\,P + \cdots) \mid (\cdots + x![\tilde{y}]\,.\,Q + \cdots)) \;\to\; (P\,[\tilde{y}/\tilde{z}] \mid Q) \quad \text{(R-G-COMM)}$$

Figure 2: Reduction rules for processes

This configuration is reached by one participant — in this example, the client — creating the common channel and sending one endpoint to the other participant. The channel z on which this initial communication takes place is defined by (νz) at the top level.

$$\mathsf{system} \stackrel{\text{def}}{=} (\nu z)((\nu c)z^+![c^+]\,.\,\mathsf{client}(c^-) \mid z^-?[x]\,.\,\mathsf{server}(x)) \tag{2}$$

Operational semantics. Let both languages be equipped with a structural pre-order (denoted $\preceq$), along the lines of that of the generic processes, inductively defined by the rules in Figure 10 (Page 109), where $P \equiv Q$ stands for $P \preceq Q$ and $Q \preceq P$.

The computational mechanism of the languages is a reduction relation on processes, inductively defined by the rules in Figure 2. The axioms use *substitution* of polarized names for unpolarized names in processes. The definition is standard, renaming bound variables if necessary in order to avoid capture: $P\,[y^p/x]$ denotes the substitution of y^p for the free occurrences of x in P, and $P\,[\widetilde{y^p}/\tilde{x}]$ denotes the simultaneous substitution of the polarized channels in $\widetilde{y^p}$ for the respective free occurrences of the channels in $\tilde{x}$ in P (assuming $|\tilde{x}| = |\widetilde{y^p}|$, where $|\tilde{x}|$ denotes the length of the sequence $\tilde{x}$ of channels). Recall that the operators of the languages which introduce bindings are restriction and the input actions. Let $\mathsf{fn}(P)$ denote the set of channels occurring free in P.

3 Types and Type Systems

We define now the type systems of both languages, and summarise the results for each system.

$$\textit{Session Types} \quad S ::= \mathtt{end} \mid ?[S_1].S_2 \mid ![S_1].S_2 \mid \&\langle l_i : S_i\rangle_{i\in I} \mid \oplus\langle l_i : S_i\rangle_{i\in I}$$
$$\textit{Typing Environments} \quad \Delta ::= x_1^{p_1} : S_1; \ldots ; x_n^{p_n} : S_n,\ n \geq 0$$

Assume that I is a non-empty finite indexing set. Consider a countable set of labels l, m, l_1, etc, disjoint from the set of channels. In a branch type $\&\langle l_i : S_i\rangle_{i\in I}$ and in a select type $\oplus\langle l_i : S_i\rangle_{i\in I}$, consider all labels pairwise disjoint. Let $\mathsf{tlbs}(S)$ denote the set of all labels occuring at top level in a session type: $\mathsf{tlbs}(\&\langle l_i : S_i\rangle_{i\in I}) = \mathsf{tlbs}(\oplus\langle l_i : S_i\rangle_{i\in I}) = \{l_i \mid i \in I\}$ and $\mathsf{tlbs}(S) = \varnothing$ in the remaining cases.

Consider a *duality* function on session types, defined as follows:

$$\overline{\mathtt{end}} = \mathtt{end},\ \ \overline{?[S_2].S_1} = ![S_2].\overline{S}_1,\ \ \overline{![S_2].S_1} = ?[S_2].\overline{S}_1,$$

$$\overline{\&\langle l_i : S_i\rangle_{i\in I}} = \oplus\left\langle l_i : \overline{S}_i\right\rangle_{i\in I},\ \text{ and }\ \overline{\oplus\langle l_i : S_i\rangle_{i\in I}} = \&\left\langle l_i : \overline{S}_i\right\rangle_{i\in I}$$

A typing environment is a mapping from polarised channels into session types. A balanced typing environment requires $\Delta(x^+) = \overline{\Delta(x^-)}$ whenever $\{x^+, x^-\} \subseteq \mathsf{dom}(\Delta)$. The following rule defines a transition relation on balanced typing environments.

$$\Delta, x^p : \&\langle l_i : S_i\rangle_{i\in I}, x^{\overline{p}} : \oplus\left\langle l_i : S'_i\right\rangle_{i\in I} \xrightarrow{x,l_k} \Delta, x^p : S_k, x^{\overline{p}} : S'_k \quad \text{if } k \in I \quad (\text{RST-COMM})$$

Finally, the rules below inductively define the type system of the source language.

$$\frac{\forall x^p \in \mathsf{dom}(\Delta), \Delta(x^p) = \mathtt{end}}{\Delta \vdash \mathbf{0}}\ (\text{T-NIL}) \qquad \frac{\vdash P}{\vdash *P}\ (\text{T-REP}) \qquad \frac{\Delta_1 \vdash P_1 \qquad \Delta_2 \vdash P_2}{(\Delta_1 + \Delta_2) \vdash (P_1 \mid P_2)}\ (\text{T-PAR})$$

$$\frac{\Delta, x^+ : S, x^- : \overline{S} \vdash P}{\Delta \vdash (\nu x)P}\ (\text{T-NEW}) \qquad \frac{\Delta, x^p : S_1, y : S_2 \vdash P}{\Delta, x^p : ?[S_2].S_1 \vdash x^p?[y].P}\ (\text{T-IN})$$

$$\frac{\Delta, x^p : S_1 \vdash P}{((\Delta, x^p : ![S_2].S_1) + y^q : S_2) \vdash x^p![y^q].P}\ (\text{T-OUT})$$

$$\frac{\forall i \in I, (\Delta, x^p : S_i \vdash P_i)}{\Delta, x^p : \&\langle l_i : S_i\rangle_{i\in I} \vdash x^p \triangleright \{l_i : P_i\}_{i\in I}}\ (\text{T-OFFER}) \quad \frac{k \in I \qquad l = l_k \qquad \Delta, x^p : S_k \vdash P}{\Delta, x^p : \oplus\langle l_i : S_i\rangle_{i\in I} \vdash x^p \triangleleft l.P}\ (\text{T-CHOOSE})$$

Figure 3: Types and typing rules for session processes

Session types. Figure 3 defines the syntax of session types and type environments, and the corresponding typing rules. Several rules use the $+$ operation on type environments (Definition 3.1). This operation is defined for a type environment and a typed identifier, and then extended inductively. It is a partial operation, and if it occurs in a rule then definedness of the operation is an implicit hypothesis of the rule.

Definition 3.1 (from [4]) 1. *Let $\Delta, x^p : S = \Delta$, if $x^p : S \in \Delta$, and $\Delta, x^p : S = \Delta \cup \{x^p : S\}$, if $x^p \notin \mathsf{dom}(\Delta)$. Otherwise the operation is undefined.*

2. *Consider $p \neq \varepsilon$. Let $\Delta + x^p : S = \Delta, x^p : S$, if $\{x^p, x^\varepsilon\} \cap \mathsf{dom}(\Delta) = \varnothing$; let $\Delta + x^\varepsilon : S = \Delta, x^\varepsilon : S$, if $\{x^+, x^-, x^\varepsilon\} \cap \mathsf{dom}(\Delta) = \varnothing$. Otherwise the operation is undefined.*

One easily concludes that the system presented in Section 2 as Process (2) is well-typed: $\vdash \mathsf{system}$. More interesting is the typing of the process below the restrictions:

$$z^+ : ![S].\mathtt{end}, z^- : ?[S].\mathtt{end}, c^+ : S, c^- : \overline{S} \vdash z^+![c^+].\mathsf{server}(c^+) \mid z^-?[x].\mathsf{client}(x)$$

$$\begin{array}{rl} \textit{Process Types} & \Gamma ::= \mathbf{0} \mid *\Gamma \mid (\Gamma_1 \,|\, \Gamma_2) \mid (\Gamma_1 \,\&\, \Gamma_2) \mid \sum_{i\in I} \gamma_i \\ \textit{Guarded Types} & \gamma ::= x![\tau]\,.\,\Gamma \mid x?[\tau]\,.\,\Gamma \mid \mathbf{t}\,.\,\Gamma \\ \textit{Tuple Types} & \tau ::= (\tilde{x})\Gamma \end{array}$$

Assume that I is a non-empty finite indexing set. The tuple type binds the channels (variables) $\tilde{x}$ in Γ. The definition of the set of free channels in Γ (denoted $\mathsf{fv}(\Gamma)$) is then straightforward. A tuple or process type is *closed* if it contains no free channels. A typing environment is a process type.

Let $S \subseteq \mathscr{V}$. Consider that $x{\downarrow_S}$ is x if it occurs in S and is $\mathbf{t}$ otherwise. Then the operation $\downarrow_S$ is defined homomorphically on closed tuple and process types, except in the following case: $((\tilde{x})\Gamma){\downarrow_S} = (\tilde{x})\Gamma{\downarrow_{S\cup\tilde{x}}}$. Moreover, $\Gamma{\uparrow_S} = \Gamma{\downarrow_{\mathscr{V}\setminus S}}$.

Generic subtyping relation: base rules.

$$(\Gamma \,|\, \mathbf{0}) \simeq \Gamma \ (\textsc{Sub-Nil}) \qquad (\Gamma_1 \,|\, \Gamma_2) \simeq (\Gamma_2 \,|\, \Gamma_1) \ (\textsc{Sub-Comm}) \qquad *\Gamma \simeq (\Gamma \,|\, *\Gamma) \ (\textsc{Sub-Unfold})$$

$$(\Gamma_1 \,|\, (\Gamma_2 \,|\, \Gamma_3)) \simeq ((\Gamma_1 \,|\, \Gamma_2) \,|\, \Gamma_3) \ (\textsc{Sub-Assoc}) \qquad (\Gamma_1 \& \Gamma_2) \leq \Gamma_i (i \in \{1,2\}) \ (\textsc{Sub-IChoice})$$

$$\frac{\Gamma \leq \Gamma'}{*\Gamma \leq *\Gamma'} \ (\textsc{Sub-Rep}) \qquad \frac{\Gamma \leq \Gamma'}{(\tilde{x})\Gamma \leq (\tilde{x})\Gamma'} \ (\textsc{Sub-Abs}) \qquad \frac{\Gamma \leq \Gamma'}{\Gamma{\downarrow_S} \leq \Gamma'{\downarrow_S}} \ (\textsc{Sub-Restrict})$$

$$\frac{\Gamma \leq \Gamma'}{\Gamma[\tilde{y}/\tilde{x}] \leq \Gamma'[\tilde{y}/\tilde{x}]} \ (\textsc{Sub-Subs}) \qquad \frac{\Gamma_1 \leq \Gamma_1' \qquad \Gamma_2 \leq \Gamma_2'}{(\Gamma_1 \,|\, \Gamma_2) \leq (\Gamma_1' \,|\, \Gamma_2')} \ (\textsc{Sub-Par})$$

Generic subtyping relation: additional rules.

$$*\mathbf{0} \simeq \mathbf{0} \ (\textsc{Sub-Inact}) \qquad \Gamma \simeq \mathbf{t}\,.\,\Gamma \ (\textsc{Sub-TPref}) \qquad \Gamma{\downarrow_V} \,|\, \Gamma{\uparrow_V} \leq \Gamma \ (\textsc{Sub-Divide})$$

$$\frac{\forall i \in I, \gamma_i \leq \gamma_i'}{\sum_{i\in I} \gamma_i \leq \sum_{i\in I} \gamma_i'} \ (\textsc{Sub-Choice})$$

Generic typing rules:

$$\frac{}{\mathbf{0} \triangleright \mathbf{0}} \ (\textsc{T-Nil}) \quad \frac{\Gamma \triangleright P}{*\Gamma \triangleright *P} \ (\textsc{T-Rep}) \quad \frac{\Gamma_1 \triangleright P_1 \qquad \Gamma_2 \triangleright P_2}{(\Gamma_1 \,|\, \Gamma_2) \triangleright (P_1 \,|\, P_2)} \ (\textsc{T-Par}) \quad \frac{\Gamma' \triangleright P \qquad \Gamma \leq \Gamma'}{\Gamma \triangleright P} \ (\textsc{T-Sub})$$

$$\frac{\Gamma_1 \,|\, \Gamma_2 \triangleright P \qquad \{\tilde{y}\} \cap \mathsf{fv}(\Gamma_1) = \varnothing}{x?[(\tilde{y})\Gamma_2]\,.\,\Gamma_1 \triangleright x?[\tilde{y}]\,.\,P} \ (\textsc{T-In}) \qquad \frac{\Gamma_1 \triangleright P}{x![(\tilde{y})\Gamma_2]\,.\,(\Gamma_1 \,|\, \Gamma_2[\tilde{z}/\tilde{y}]) \triangleright x![\tilde{z}]\,.\,P} \ (\textsc{T-Out})$$

$$\frac{\forall i \in I, \gamma_i \triangleright G_i}{\sum_{i\in I} \gamma_i \triangleright \sum_{i\in I} G_i} \ (\textsc{T-Choice}) \qquad \frac{\Gamma \triangleright P \qquad \mathsf{ok}(\Gamma{\downarrow_{\tilde{x}}}) \qquad \mathsf{fv}(\Gamma{\uparrow_{\tilde{x}}}) \cap \{\tilde{x}\} = \varnothing}{\Gamma{\uparrow_{\tilde{x}}} \triangleright (\nu\tilde{x})P} \ (\textsc{T-New})$$

Figure 4: Types, subtyping and typing for generic processes

$$\frac{\tau_1 \leq \tau_2}{(x![\tau_1].\Gamma_1 \,|\, x?[\tau_2].\Gamma_2) \;\rightarrow\; (\Gamma_1 \,|\, \Gamma_2)}\ (\text{RPT-COMM}) \qquad \mathbf{t}.\Gamma \rightarrow \Gamma\ (\text{RPT-EVT})$$

$$\frac{\Gamma \rightarrow \Gamma'}{(\Gamma \,|\, \Gamma_1) \;\rightarrow\; (\Gamma' \,|\, \Gamma_1)}\ (\text{RPT-PAR}) \qquad \frac{\Gamma_1 \leq \Gamma_1' \qquad \Gamma_1' \rightarrow \Gamma_2' \qquad \Gamma_2' \leq \Gamma_2}{(\Gamma_1 \,|\, \Gamma_2) \;\leq\; (\Gamma_1' \,|\, \Gamma_2')}\ (\text{RPT-SUB})$$

Figure 5: Reduction on process types

where the session types S and $\overline{S}$ are defined in Section 1.

Gay and Hole [4] proved Type Preservation and Type Safety (no errors in well-typed processes). Appendix A presents the results adapted to the setting we use herein (Theorems A.2 and A.3).

Generic types. GTS is parameterized by a subtyping relation and a consistency condition on types: instantiating them yields a particular type system, ensuring a given (safety) property on processes. These "generic" conditions occur in the typing rules. Type soundness results depend on the particular semantic property on processes one is interested in. We present thus the particular subtyping relation and consistency predicate used in this work: Figure 4 defines the syntax of generic process types, and presents the rules defining the subtyping and the typing relations. Most constructors are fairly standard in process algebras: '*' stands for replication, '&' for internal choice, '+' for external choice, and '**t**' for a synchonisation event. The subtyping relation $\leq$ is the preorder satisfying the rules presented in the figure, where $\Gamma_1 \simeq \Gamma_2$ if $\Gamma_1 \leq \Gamma_2$ and $\Gamma_2 \leq \Gamma_1$.

Since types are themselves processes, one needs to define an operational semantics to describe their evolution. Let $\Gamma[\tilde{y}/\tilde{x}]$ denote the simultaneous capture-avoiding substitution of the channels in $\tilde{y}$ for the respective free occurrences of the channels in $\tilde{x}$ in Γ. Reduction on process types is inductively defined by the rules in Figure 5. Write $\rightarrow^*$ for the reflexive and transitive closure of $\rightarrow$.

The process types that "behave correctly" are those that are *well-formed*.

Definition 3.2 (from [6]) *A process type* Γ *is* well-formed, *written* $\mathsf{WF}(\Gamma)$, *if whenever* $\Gamma \rightarrow^* (x![\tau_1].\Gamma_1 \,|\, x?[\tau_2].\Gamma_2 \,|\, \Gamma_3)$ *then* $\tau_1 \leq \tau_2$ *holds.*

To ensure particular conditions on the behaviour of process types, GTS introduces the notion of *consistency predicate*. It uses the auxiliary notion of non-active process type, in the form of a NULL predicate.

Definition 3.3 (from [6]) *Let* $\mathsf{NULL}(\Gamma)$ *hold if* Γ *has no subprocess input or output guarded. A predicate* ok *on process types is a* proper consistency predicate *if it is preserved by reduction, if* $\mathsf{ok}(\Gamma)$ *then* $\mathsf{WF}(\Gamma)$, *and if* $\mathsf{ok}(\Gamma)$ *and* $\mathsf{NULL}(\Gamma')$ *then* $\mathsf{ok}(\Gamma \,|\, \Gamma')$.

The particular consistency condition we are interested in is *linearity*: a process type has no parallel sends or receives on a given channel.

Definition 3.4 *A process type* Γ *is* linear, *written* $\mathsf{LIN}(\Gamma)$, *if* $\mathsf{WF}(\Gamma)$, *if* $\Gamma \rightarrow^* (x![\tau_1].\Gamma_1 \,|\, \Gamma_2)$ *implies* $\Gamma_2 \not\rightarrow^* (x![\tau_2].\Gamma_3 \,|\, \Gamma_4)$, *and if* $\Gamma \rightarrow^* (x?[\tau_1].\Gamma_1 \,|\, \Gamma_2)$ *implies* $\Gamma_2 \not\rightarrow^* (x?[\tau_2].\Gamma_3 \,|\, \Gamma_4)$.

Lemma 3.5 $\mathsf{LIN}(\Gamma)$ *is a proper consistency predicate.*

Igarashi and Kobayashi proved Subject Reduction — as processes evolve "their" (well-formed) process types evolve accordingly — and, as a corollary, a property preservation result relating properties on processes and on process types: let p be an invariant predicate on processes and consider an ok predicate

Let φ be a name translation function. The translation of processes is defined by the following axioms (here p, q may be $+$, $-$ or ε), assuming the names u and v fresh, being homomorphic in the cases omitted.

$$[\![x^p\,![y^q]\,.\,P]\!]_\varphi = [\![x^p\,![y^q]]\!]_\varphi \,.\, [\![P]\!]_\varphi \qquad [\![x^p?[y]\,.\,P]\!]_\varphi = [\![x^p?[y]]\!]_\varphi^{(u,v)} \,.\, [\![P]\!]_{\varphi+\{y\mapsto(u,v)\}}$$

$$[\![(\nu x)\,P]\!]_\varphi = (\nu u,v)\,[\![P]\!]_{\varphi+\{x\mapsto(u,v)\}}$$

The translation of prefixes is defined as follows, where $p,q \neq \varepsilon$.

$$[\![x^p\,![y^q]]\!]_\varphi = \varphi^{\overline{p}}(x)\,!\left[\varphi^q(y),\varphi^{\overline{q}}(y)\right] \qquad [\![x^p\,![y]]\!]_\varphi = \varphi^{\overline{p}}(x)\,!\left[\varphi^+(y),\varphi^-(y)\right]$$

$$[\![x\,![y^q]]\!]_\varphi = \varphi^-(x)\,!\left[\varphi^q(y),\varphi^{\overline{q}}(y)\right] \qquad [\![x\,![y]]\!]_\varphi = \varphi^-(x)\,!\left[\varphi^+(y),\varphi^-(y)\right]$$

$$[\![x^p?[y]]\!]_\varphi^{(u,v)} = \varphi^p(x)?[u,v] \qquad [\![x?[y]]\!]_\varphi^{(u,v)} = \varphi^+(x)?[u,v]$$

Figure 6: Process translation

that is its correspondent on process types; a consequence of Subject Reduction is that, if a process type typing a given process satisfies an ok predicate, the process satisfies the invariant p (Theorem A.5 in Appendix A). Hereafter, $\mathsf{LIN}(\Gamma)$ is our ok predicate.

The counterpart of an error session process is a generic process with an arity mismatch or with races (parallel sends or receives on the same channel).

Definition 3.6 (Error process) *A generic process P is* an error*, if one of the three conditions below hold.*

1. *Whenever $P \equiv (\nu\tilde{x})\,(x?[\tilde{z}]\,.\,P_1 \mid x![\tilde{y}]\,.\,P_2 \mid Q)$ it is the case that $|\tilde{z}| \neq |\tilde{y}|$;*
2. *$P \equiv (\nu\tilde{x})\,(x![\tilde{z}]\,.\,P_1 \mid x![\tilde{y}]\,.\,P_2 \mid Q)$;*
3. *$P \equiv (\nu\tilde{x})\,(x?[\tilde{z}]\,.\,P_1 \mid x?[\tilde{y}]\,.\,P_2 \mid Q)$.*

The counterpart of session type safety is simply linearity and the absence of arity mismatches. Linearity implies the absence of races, as ensured usually by session types; the definition of well-formedness (Definition 3.2 above) implies no arity mismatches, allowing however a channel to change its arity after a reduction step. The predicate LIN states these conditions on process types: in the original process and after each reduction step, there are no parallel sends or receives on the same channel nor arity mismatches (this last condition resulting from LIN being a consistency predicate — Lemma 3.5).

To ensure that well-typed generic processes are type safe, it suffices to show that absence of errors in generic processes (an invariant property) corresponds to LIN. The result is a corollary of a theorem proved by Igarashi and Kobayashi.

Proposition 3.7 (Resulting from Theorem 5.1 of [6]) *If $\Gamma \triangleright P$ and $\mathsf{LIN}(\Gamma)$ then P is not an error.*

4 Encoding Polarities

The first question in defining a translation from session processes to generic processes is how to represent polarities. It is easy to see that a non-trivial encoding is necessary. Simply erasing polarities would lead to reductions in the target language that are not possible in the source language; this is clear from R-S-COM and R-G-COM. We do not restrict the encoding to well-typed processes, as we want to show that it

Let φ be the mapping from names to pairs of names, introduced in Figure 6.
Consider $(\!|S|\!) = (u,v)[\![y : S]\!]_{\{y \mapsto (u,v)\} \cup \varphi}$, where u, v, and y are fresh.

$$[\![\{x_1^{p_1} : S_1, \ldots, x_n^{p_n} : S_n\}]\!]_\varphi = [\![x_1^{p_1} : S_1]\!]_\varphi \mid \cdots \mid [\![x_n^{p_n} : S_n]\!]_\varphi$$

$$[\![x^p : \mathtt{end}]\!]_\varphi = \mathbf{0}$$

$$[\![x : ![S_1].S_2]\!]_\varphi = \varphi^-(x)![(\!|S_1|\!)].[\![x : S_2]\!]_\varphi \qquad [\![x : ?[S_1].S_2]\!]_\varphi = \varphi^+(x)?[(\!|S_1|\!)].[\![x : S_2]\!]_\varphi$$

$$[\![x^+ : ?[S_1].S_2]\!]_\varphi = \varphi^+(x)?[(\!|S_1|\!)].[\![x^+ : S_2]\!]_\varphi \qquad [\![x^- : ?[S_1].S_2]\!]_\varphi = \varphi^-(x)?[(\!|S_1|\!)].[\![x^- : S_2]\!]_\varphi$$

$$[\![x^+ : ![S_1].S_2]\!]_\varphi = \varphi^-(x)![(\!|S_1|\!)].[\![x^+ : S_2]\!]_\varphi \qquad [\![x^- : ![S_1].S_2]\!]_\varphi = \varphi^+(x)![(\!|S_1|\!)].[\![x^- : S_2]\!]_\varphi$$

Figure 7: Type environment translation

is possible to define a translation where the session types system and an instance of GTS yield the same classification of typable processes.

The translation guarantees an operational and a typing correspondence. To state the former in the reverse direction, we need to restrict the result to well-typed processes.

To simplify the presentation in this section we consider only choice-free (*i.e.*, non-branching, using only sequence and parallel) processes and types.

The process translation function maps each free name of the process into two new target names, according to the rules in Figure 6. The translation uses a *name translation* (partial) *function* φ from names to pairs of names. Write φ^+ and φ^- for the compositions of φ with the first and second projections, respectively. We require φ^+ and φ^- to be injective and have disjoint images. When $\mathsf{dom}(\varphi)$ and $\mathsf{dom}(\psi)$ are disjoint, we write $\varphi + \psi$ to denote the union of the name translation functions φ and ψ (otherwise, this operation is undefined).

To illustrate this encoding, consider the process system (Process 2 in Page 98). The encoding of the process is below, considering $\varphi \stackrel{\text{def}}{=} \{x \mapsto (t,u)\}, \chi \stackrel{\text{def}}{=} \{z \mapsto (v,w)\}, \psi \stackrel{\text{def}}{=} \{c \mapsto (d,e)\}$.

$$\begin{aligned}
[\![\mathsf{system}]\!]_\varnothing &= (\nu v,w)\,[\![(\nu c)\,z^+![c^+].\mathsf{client}(c^-) \mid z^-?[x].\mathsf{server}(x)]\!]_\chi \\
&= (\nu v,w)\,((\nu d,e)\,w![d,e].[\![\mathsf{client}(c^-)]\!]_{\chi+\psi}) \mid w?[t,u].[\![\mathsf{server}(x)]\!]_{\varphi+\chi}
\end{aligned}$$

Thus, the communication between z^+ and z^- becomes a communication on w.

The forward operational correspondence between source and target reduction steps is one-to-one. Let P be a choice-free session process and φ be a name translation function.

Lemma 4.1 *1. If $p \neq \varepsilon$ then $[\![P[x^p/y]]\!]_\varphi = [\![P]\!]_{\varphi + \{y \mapsto (\varphi^p(x), \varphi^{\overline{p}}(x))\}}$*

2. $[\![P[x/y]]\!]_\varphi = [\![P]\!]_{\varphi + \{y \mapsto (\varphi^+(x), \varphi^-(x))\}}$

Theorem 4.2 *If $P \longrightarrow Q$ then $[\![P]\!]_\varphi \longrightarrow [\![Q]\!]_\varphi$.*

The type translation (in Figure 7) maps sequential session types to generic (choice-free) process types. Let us first present an example. Considering

$$\Delta \stackrel{\text{def}}{=} \{c^+ : ?[\mathsf{int}] . ![\mathsf{bool}] . \mathsf{end}, c^- : ![\mathsf{int}] . ?[\mathsf{bool}] . \mathsf{end}\}$$

we have $\Delta \vdash c?[i] . c![i=3] . \mathbf{0} \mid c![3] . c?[b] . \mathbf{0}$. Then,

$$\begin{aligned} \Gamma \stackrel{\text{def}}{=} [\![\Delta]\!]_\psi &= [\![\{c^+ : ?[\mathsf{int}] . ![\mathsf{bool}] . \mathsf{end}\}]\!]_\psi \mid [\![\{c^- : ![\mathsf{int}] . ?[\mathsf{bool}] . \mathsf{end}\}]\!]_\psi \\ &= d?[\mathsf{int}] . e![\mathsf{bool}] . \mathbf{0} \mid d![\mathsf{int}] . e?[\mathsf{bool}] . \mathbf{0} \end{aligned}$$

It is easy to check that the encoded system is typable with the encoding of the type environment (*i.e.*, $\Gamma \triangleright [\![\mathsf{system}]\!]_\psi$). However, the generic process type no longer captures the flow of the protocol, as the two steps (exchanging first an integer and then a boolean) happen now on different channels. If one thinks of a process with (possibly long and complex) sessions, one understands that the encoding produces a large number of new channels, requiring a partial order on them to exibit the flow of the protocol that is clear in each session type.

The reverse operational correspondence requires typing: a communication between, for example, send on x^- and receive on x, does not reduce in the source language (and is ill-typed) but translates into a reduction in the target language.

Theorem 4.3 *Let P be a well-typed choice-free session process and let φ be a name translation function. If $[\![P]\!]_\varphi \longrightarrow Q$ then there exists P' such that $P \longrightarrow P'$ and $[\![P']\!]_\varphi = Q$.*

Typing correspondence. We show a correspondence in both directions between typing derivations. Let P be a choice-free session process and let φ be a name translation function. We state first completeness and then soundness.

Theorem 4.4 *If $\Delta \vdash P$ for some balanced Δ, then $[\![\Delta]\!]_\varphi \triangleright [\![P]\!]_\varphi$ and $\mathsf{LIN}([\![\Delta]\!]_\varphi)$.*

The converse of completeness does not hold. For example, take $P = *\mathbf{0}$. Then $[\![x^+ : \mathsf{end}]\!] \triangleright [\![*\mathbf{0}]\!]$, but it is not the case that $x^+ : \mathsf{end} \vdash *\mathbf{0}$.

Theorem 4.5 *If $\Gamma \triangleright [\![P]\!]_\varphi$ and $\mathsf{LIN}(\Gamma)$ then $\Delta \vdash P$, for some balanced Δ.*

Note that Γ must be linear, otherwise, considering $\varphi(x) = (u,v)$, we have $(u![] \mid u![]) \triangleright (u![] \mid u![])$, whereas $(x^+![] \mid x^+![])$ is not typable as a session process.

5 Encoding labels

Figure 8 extends the translation of processes to include labelled sums (branch) and selectors. We translate the labels occuring in a branch process as fresh names that are sent to the translation of the corresponding select process; the latter, in turn, selects its desired branch outputing on the name corresponding to the label. We use a function σ to map the labels to the fresh channels.

To translate the select process, we need typing information (an environment typing the process) to know how many labels the corresponding branch has. That is the number of parameters of the channel which is waiting for the fresh names created by the other end-point to represent the labels.

Let Δ be a typing environment.

Extend the translation of prefixes with the following rules, where $p \neq \varepsilon$.

$$[\![x^p \triangleleft l]\!]^{\Delta}_{\varphi} = \varphi^p(x)?[\lambda_1 \ldots \lambda_n], \text{ where } \{\lambda_1 \ldots \lambda_n\} = \mathsf{tlbs}(\Delta(x^{\overline{p}})), \text{ with } n \geq 0$$
$$[\![x \triangleleft l]\!]^{\Delta}_{\varphi} = \varphi^+(x)?[\lambda_1 \ldots \lambda_n], \text{ where } \{\lambda_1 \ldots \lambda_n\} = \mathsf{tlbs}(\Delta(x^{\overline{p}})), \text{ with } n \geq 0$$

Extend the translation of processes with the following rules, where $p \in \{+,-,\varepsilon\}$. For all $i \in \{1,\ldots,n\}$ let $\{\lambda_1 \ldots \lambda_n\} \cap \mathsf{fn}(P_i) = \varnothing$, and let $\Delta \xrightarrow{x,l_i} \Delta_i$. Let $\sigma(l) = \lambda_j$ where $\Delta(x^p) = \oplus\langle l_i : S_i\rangle_{i=1}^n$ and $l = l_j$.

$$[\![x^p \triangleleft l\,.\,P]\!]^{\Delta}_{\varphi} = [\![x^p \triangleleft l]\!]^{\Delta}_{\varphi}\,.\,\sigma(l)\,!\,[]\,.\,[\![P]\!]^{\Delta_i}_{\varphi}$$
$$[\![x^+ \triangleright \{l_i : P_i\}_{i=1}^n]\!]^{\Delta}_{\varphi} = (\nu\lambda_1 \ldots \lambda_n)\,\varphi^-(x)\,!\,[\lambda_1 \ldots \lambda_n]\,.\,\textstyle\sum_{i=1}^n \lambda_i?[]\,.\,[\![P_i]\!]^{\Delta_i}_{\varphi}$$
$$[\![x^- \triangleright \{l_i : P_i\}_{i=1}^n]\!]^{\Delta}_{\varphi} = (\nu\lambda_1 \ldots \lambda_n)\,\varphi^+(x)\,!\,[\lambda_1 \ldots \lambda_n]\,.\,\textstyle\sum_{i=1}^n \lambda_i?[]\,.\,[\![P_i]\!]^{\Delta_i}_{\varphi}$$
$$[\![x \triangleright \{l_i : P_i\}_{i=1}^n]\!]^{\Delta}_{\varphi} = (\nu\lambda_1 \ldots \lambda_n)\,\varphi^-(x)\,!\,[\lambda_1 \ldots \lambda_n]\,.\,\textstyle\sum_{i=1}^n \lambda_i?[]\,.\,[\![P_i]\!]^{\Delta_i}_{\varphi}$$

Figure 8: Process translation for branch and select

The illustrate the idea, we encode now the server (process 1). Recall that $\varphi \stackrel{\text{def}}{=} \{x \mapsto (t,u)\}$ and $\chi \stackrel{\text{def}}{=} \{z \mapsto (v,w)\}$, $S = \&\langle \mathsf{service} : ?[\mathsf{int}]\,.\,![\mathsf{bool}]\,.\,\mathsf{end}, \mathsf{quit} : \mathsf{end}\rangle$, and $\Delta \stackrel{\text{def}}{=} \{x^+ : ?[\mathsf{int}]\,.\,![\mathsf{bool}]\,.\,\mathsf{end}, x^- : ![\mathsf{int}]\,.\,?[\mathsf{bool}]\,.\,\mathsf{end}\}$. Consider now $\Delta' \stackrel{\text{def}}{=} \{x^+ : S, x^- : \overline{S}\}$ and let the function σ associate the labels *service* and *quit* with the channels *service* and *quit*. Then,

$$\begin{aligned}[\![\mathsf{server}(x^+)]\!]^{\Delta'}_{\varphi+\chi} \quad = \quad & (\nu service, quit)\,u\,!\,[service, quit]\,. \\ & (service?[]\,.\,[\![\mathsf{serviceBody}(x^+)]\!]^{\Delta'}_{\varphi+\chi} + quit?[]\,.\,\mathbf{0})\end{aligned}$$

Forward operational correspondence. We no longer have a one-to-one correspondence between reduction steps, because communication on a label in the source language is translated into two communications.

Theorem 5.1 *Whenever $P \longrightarrow Q$, then $[\![P]\!] \longrightarrow^n [\![Q]\!]$ with $n = 1$ or $n = 2$.*

Reverse operational correspondence. This must also take into account the extra reductions involved in communication of labels. The second case in the theorem describes the intermediate configuration between the two reduction steps corresponding to communication of a label.

Theorem 5.2 *Let $\Delta \vdash P$ with Δ balanced and let φ be a name translation function. If $[\![P]\!]^{\Delta}_{\varphi} \longrightarrow Q$ then there exists Δ' and P' such that $P \longrightarrow P'$ and either $[\![P']\!]^{\Delta'}_{\varphi} = Q$ or $Q \equiv (\nu\tilde{\lambda})\,Q_1$ and $Q_1 \to Q_2 \equiv [\![P']\!]^{\Delta'}_{\varphi}$.*

Notice that, by Subject-Reduction, Δ' is balanced and $\Delta' \vdash P'$.

Let φ be the mapping from names to pairs of names, introduced in Figure 6.
Consider $(\!|S|\!) = (u,v)[\![y : S]\!]_{\{y \mapsto (u,v)\} \cup \varphi}$, where u, v, and y are fresh.
For all $i \in \{1,\ldots,n\}$ let $\{\lambda_1 \ldots \lambda_n\} \cap \mathsf{fn}(P_i) = \varnothing$.

$$
\begin{aligned}
[\![x^+ : \&\langle l_i : S_i\rangle_{i=1}^n]\!]_\varphi &= \varphi^-(x)\,![\lambda_1,\ldots,\lambda_n]\,\textstyle\sum_{i=1}^n \lambda_i?[(\!|S_i|\!)]\,.\,\mathbf{0}\\
[\![x^- : \&\langle l_i : S_i\rangle_{i=1}^n]\!]_\varphi &= \varphi^+(x)\,![\lambda_1,\ldots,\lambda_n]\,\textstyle\sum_{i=1}^n \lambda_i?[(\!|S_i|\!)]\,.\,\mathbf{0}\\
[\![x : \&\langle l_i : S_i\rangle_{i=1}^n]\!]_\varphi &= \varphi^-(x)\,![\lambda_1,\ldots,\lambda_n]\,\textstyle\sum_{i=1}^n \lambda_i?[(\!|S_i|\!)]\,.\,\mathbf{0}\\
[\![x^+ : \oplus\langle l_i : S_i\rangle_{i=1}^n]\!]_\varphi &= \varphi^+(x)?[\lambda_1,\ldots,\lambda_n]\,\&_{i=1}^n \lambda_i!\,\left[(\!|\overline{S_i}|\!)\right]\,.\,\mathbf{0}\\
[\![x^- : \oplus\langle l_i : S_i\rangle_{i=1}^n]\!]_\varphi &= \varphi^-(x)?[\lambda_1,\ldots,\lambda_n]\,\&_{i=1}^n \lambda_i!\,\left[(\!|\overline{S_i}|\!)\right]\,.\,\mathbf{0}\\
[\![x : \oplus\langle l_i : S_i\rangle_{i=1}^n]\!]_\varphi &= \varphi^+(x)?[\lambda_1,\ldots,\lambda_n]\,\&_{i=1}^n \lambda_i!\,\left[(\!|\overline{S_i}|\!)\right]\,.\,\mathbf{0}
\end{aligned}
$$

Figure 9: Type environment translation

Typing correspondence. Figure 9 extends the translation of typing environments to branching types. Error processes in the full session language must now take into account the possibility of a selector "asking for" a non-existing label in a branching offer:

if $P \equiv (\nu\tilde{x})\,(x^p \triangleright \{l_i : P_i\}_{i\in I} \mid x^{\overline{p}} \triangleleft l\,.\,Q \mid R)$ then either $l = l_k$ and $k \notin I$ or $\{x^+, x^-\} \cap \mathsf{fn}(R) \neq \varnothing$.

The ok predicate is still simply LIN. Theorems 4.4 and 4.5 have corresponding versions for the full calculi: a session process is well-typed if and only if its encoding is well-typed in an ok typing environment. In the following results, let P be a session process and let φ be a name translation function.

Theorem 5.3 (Completeness) *If $\Delta \vdash P$ then $[\![\Delta]\!]_\varphi \triangleright [\![P]\!]_\varphi^\Delta$ and $\mathsf{LIN}([\![\Delta]\!]_\varphi)$.*

Theorem 5.4 (Soundness) *Let Δ' be a balanced session type environment. If $\Gamma \triangleright [\![P]\!]_\varphi^{\Delta'}$ and $\mathsf{LIN}(\Gamma)$ then $\Delta \vdash P$, for some balanced Δ.*

6 Subtyping correspondence

Subtyping is an essential ingredient of the theory of session types. Originally proposed by Gay and Hole [4], it has been widely used in other session-based systems, with subject-reduction and type-safety holding. We now discuss how to represent session subtyping in GTS.

Safe substitutability. Notice that subtyping in session types means less branching ('&') and more choice ('$\oplus$'). This basic principle conforms to the "safe substitutability principle" of Liskov and Wing [9]. However, the principle has no counterpart in process types: although the internal choice axiom follows the principle, the rule for external choice ('+') does not allow changing the number of arguments of the operation (Appendix B presents the subtyping relations on session types and on generic process types).

It is thus not surprising that the encoding presented in the previous section does not preserve subtyping. Let $S_1 \stackrel{\text{def}}{=} \&\langle l_1 : \mathsf{end}\rangle$ and $S_2 \stackrel{\text{def}}{=} \&\langle l_1 : \mathsf{end}, l_2 : \mathsf{end}\rangle$; we have $S_1 \leqslant S_2$, but the encoded types are not related:

$$(\!|S_1|\!) = (u,v)v![l_1]\,l_1![(\!|\mathsf{end}|\!)]\,.\,\mathbf{0} \not\leq (u,v)v![l_1,l_2]\,(l_1![(\!|\mathsf{end}|\!)]\,\&l_2![(\!|\mathsf{end}|\!)])\,.\,\mathbf{0} = (\!|S_2|\!)$$

The problem is twofold: (1) the subtyping relation on process types does not allow changing the sequence of names sent on an output; (2) session subtyping means less branching, but process subtyping on choice does not allow to change the number of summands.

To achieve an encoding guaranteeing a typing correspondence one would have to "compensate" subtyping, always passing all labels occurring at top level in the branch process of a given channel. Consider

$$\Delta \stackrel{\text{def}}{=} \{x^+ : \& \langle l_1 : \mathsf{end}\rangle, x^- : \oplus \langle l_1 : \mathsf{end}\rangle\} \text{ and } P \stackrel{\text{def}}{=} (x^+ \triangleright \{\mathsf{l}_1 : \mathbf{0}, \mathsf{l}_2 : \mathbf{0}\} \mid x^- \triangleleft \mathsf{l}_1 . \mathbf{0})$$

Note that $\Delta \vdash P$. To avoid breaking the typing correspondence, instead of encoding Δ and P independently, if one encodes the typing judgement then the encoding of Δ needs to take into account all the labels of the branch offered by x^+. It is easy to get an encoding preserving and reflecting typability, but actually, the idea presented removes subtyping. The interesting question is whether an encoding in GTS, capturing the session type subtyping discipline, exists or not.

Variant subtyping. A straightforward way of faithfully representing session subtyping in GTS is to extend the language of processes with labelled values and a case constructor, and the language of types with variants (as done by Dardha *et al.* [1] and by Demangeon and Honda [2]). Using the usual rules of variant subtyping ([11]) one gets a sound and complete encoding.

7 Conclusions

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.

Therefore, the translations clarify the relationship between session types and GTS, and provide an interesting application of GTS, which can thus be used to support analysis techniques for sessions.

A preliminary version of this work was presented at PLACES'08 (but was not published). The translations presented in that work were more complicated (using forwarders) and did not consider subtyping.

Achievements. The encodings are a contribution: although encodings in process calculi have been thoroughly studied, the two aspects presented herein are novel: we are not aware of other investigations on how to encode polarities or labels (as constants) in calculi without such constructs. Our proposals may be used in other contexts.

The translation of branching and select types in the presence of subtyping reveals a difficulty: the protocol of sending the labels of the branch type to then offer a choice does not respect the generic process subtyping discipline. This is because the encoding of labelled choice requires the labels to be passed as fresh names; however, as subtyping does not allow changing the length of the sequence of names passed, session types related by subtyping are translated into unrelated process types. Even if the labels are passed one by one (as in the encoding of polyadic into monadic π-calculus), the encoded session types would not be related by subtyping; they may, nonetheless, be related by simulation.

We do not see how to achieve a correspondence between subtyping in session types and process types, but we leave the possibility of proving a negative result for future work.

An encoding using records (or a case constructor) and variant types in the target language is, however, simple to achieve, as done by Dardha *et al.* [1] and by Demangeon and Honda [2].

Assessment. In our opinion, the translation into GTS stresses that session types themselves remain of great interest for programming language design. Dardha *et al.* reached similar conclusions. There are several reasons for that.

First, session types are a high-level abstraction for structuring inter-process communication [12]; preservation of this abstraction and the corresponding programming primitives is very important for high-level programming. The translation of a session type into a process type produces a less informative type: the global specification of behaviour is lost.

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 the extension of GTS to other languages, which might not be easy (apart from adding the constructors required for subtyping).

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. It may, however, be possible to use treatments of deadlock-freedom in the generic type system as a basis for understanding how to combine session types and deadlock-freedom more directly.

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.

Fifth, the subtyping principles of session types, which provide flexibility both for programming and for typing, are not easily captured in a subtyping relation on "plain" process types.

In short, session constructors and session types are encodable in process types, and one may use the power of GTS to represent and reason about session specifications. As future work, we plan to investigate concrete analysis techniques for sessions based on GTS.

Acknowledgements. We are grateful to Luís Caires, Kohei Honda, and Naoki Kobayashi for useful discussions. We are also grateful for the careful work and important comments/suggestions of various anonymous reviewers.

António Ravara is partially supported by the Portuguese Fundação para a Ciência e a Tecnologia via project "Centro de Informática e Tecnologias da Informação (CITI/FCT/UNL)" — grant PEst-OE/EEI/UI0527/2014, and project "Liveness, statically" — grant PTDC/EIA-CCO/117513/2010. Simon Gay is partially supported by EPSRC grant EP/K034413/1 (From Data Types to Session Types: A Basis for Concurrency and Distribution).

References

[1] Ornela Dardha, Elena Giachino & Davide Sangiorgi (2012): *Session Types Revisited*. In: *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming (PPDP'12)*, ACM, pp. 139–150, doi:10.1145/2370776.2370794.

[2] Romain Demangeon & Kohei Honda (2011): *Full abstraction in a subtyped pi-calculus with linear types*. In: *Proceedings of the 22nd International Conference on Concurrency Theory*, *LNCS* 742, Springer, pp. 460–474, doi:10.1007/978-3-642-23217-6_19.

[3] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida & Sophia Drossopoulou (2006): *Session Types for Object-Oriented Languages*. In: *Proceedings of the 14th ECOOP*, *LNCS* 4067, Springer, pp. 328–352, doi:10.1007/11785477_20.

$$*\mathbf{0} \equiv \mathbf{0} \ \text{(SP-STAR)} \qquad (P \mid \mathbf{0}) \equiv P \ \text{(SP-NIL)} \qquad *P \preceq (*P \mid P) \ \text{(SP-REP)}$$

$$(P \mid Q) \equiv (Q \mid P) \ \text{(SP-COMMUT)} \qquad (P \mid (Q \mid R)) \equiv ((P \mid Q) \mid R) \ \text{(SP-ASSOC)}$$

$$((\nu\tilde{x})\, P \mid Q) \equiv (\nu\tilde{x})\, (P \mid Q) \text{ if } \tilde{x} \text{ are not free in } Q \ \text{(SP-NEW)}$$

$$\frac{P \preceq P' \qquad Q \preceq Q'}{(P \mid Q) \preceq (P' \mid Q')} \ \text{(SP-PAR)} \qquad \frac{P \preceq Q}{(\nu\tilde{x})\, P \preceq (\nu\tilde{x})\, Q} \ \text{(SP-CNEW)}$$

Figure 10: Structural pre-order — rules

[4] Simon J. Gay & Malcolm J. Hole (2005): *Subtyping for Session Types in the Pi-Calculus*. *Acta Informatica* 42(2-3), pp. 191–225, doi:10.1007/s00236-005-0177-z.

[5] Kohei Honda, Vasco Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In: *Proceedings of the 14th ESOP, LNCS* 1381, Springer, pp. 122–138, doi:10.1007/BFb0053567.

[6] Atsushi Igarashi & Naoki Kobayashi (2004): *A Generic Type System for the Pi-Calculus*. *Theoretical Computer Science* 311(1–3), pp. 121–163, doi:10.1016/S0304-3975(03)00325-6.

[7] Naoki Kobayashi (2002): *Type Systems for Concurrent Programs*. In: *Formal Methods at the Crossroads: From Panacea to Foundational Support (Proceedings of UNU/IIST 10th Anniversary Colloquium), LNCS* 2757, Springer, pp. 439–453, doi:10.1007/978-3-540-40007-3_26. Extended version at `http://www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf`.

[8] Naoki Kobayashi, Benjamin C. Pierce & David N. Turner (1999): *Linearity and the Pi-Calculus*. *ACM Transactions on Programming Languages and Systems* 21(5), pp. 914–947, doi:10.1145/330249.330251.

[9] Barbara Liskov & Jeannette M. Wing (1994): *A Behavioral Notion of Subtyping*. *ACM Transactions on Programming Languages and Systems* 16(6), pp. 1811–1841, doi:10.1145/197320.197383.

[10] Robin Milner, Joachim Parrow & David Walker (1992): *A Calculus of Mobile Processes, parts I and II*. *Information and Computation* 100(1), pp. 1–77, doi:10.1016/0890-5401(92)90009-5.

[11] Davide Sangiorgi & David Walker (2001): *The pi-calculus, a theory of mobile processes*. Cambridge University Press.

[12] Kaku Takeuchi, Kohei Honda & Makoto Kubo (1994): *An Interaction-Based Language and its Typing System*. In: *Proceedings of the 14th PARLE, LNCS* 817, Springer, pp. 398–413, doi:10.1007/3-540-58184-7_118.

A Properties of the source and target languages

On session processes.

Definition A.1 (from [4]) *A type environment is* balanced *if* $\{x^+, x^-\} \subseteq \mathsf{dom}(\Delta)$ *implies* $\Delta(x^+) = \overline{\Delta(x^-)}$.

Theorem A.2 (Type Preservation, from [4]) *If* $\Delta \vdash P$ *with* Δ *balanced and* $P \to P'$ *then there exists a balanced* Δ' *such that* $\Delta' \vdash P'$.

Type safety guarantees that a correctly-typed process contains no immediate possibilities for an error (in this context the property is called "session fidelity"). With Type Preservation, it ensures that "well-typed processes do not go wrong".

Theorem A.3 (Session Fidelity, from [4]) *Let* $\Delta \vdash P$ *for a balanced* Δ.

$$\frac{}{\mathtt{end} \leqslant \mathtt{end}}\ (\text{S-END}) \qquad \frac{S_1 \leqslant S_1' \qquad S_2 \leqslant S_2'}{?[S_1].S_2 \leqslant ?[S_1'].S_2'}\ (\text{S-IN}) \qquad \frac{S_1' \leqslant S_1 \qquad S_2 \leqslant S_2'}{![S_1].S_2 \leqslant ![S_1'].S_2'}\ (\text{S-OUT})$$

$$\frac{I \subseteq J \qquad \forall i \in I, S_i \leqslant S_i'}{\&\langle l_i : S_i\rangle_{i\in I} \leqslant \&\langle l_i : S_i'\rangle_{i\in J}}\ (\text{S-BRANCH}) \qquad \frac{J \subseteq I \qquad \forall i \in J, S_i \leqslant S_i'}{\oplus\langle l_i : S_i\rangle_{i\in I} \leqslant \oplus\langle l_i : S_i'\rangle_{i\in J}}\ (\text{S-CHOICE})$$

Figure 11: Subtyping for session types

$$\frac{\Delta, x^p : S_1, y : S_2' \vdash P \qquad S_2 \leqslant S_2'}{\Delta, x^p : ?[S_2].S_1 \vdash x^p?[y].P}\ (\text{T-IN}) \qquad \frac{\Delta, x^p : S_1 \vdash P \qquad S_2' \leqslant S_2}{(\Delta, x^p : ![S_2].S_1) + y^q : S_2' \vdash x^p![y]^q.P}\ (\text{T-OUT})$$

$$\frac{I \subseteq J \qquad \forall i \in I, (\Delta, x^p : S_i \vdash P_i)}{\Delta, x^p : \&\langle l_i : S_i\rangle_{i\in I} \vdash x^p \triangleright \{l_i : P_i\}_{i\in J}}\ (\text{T-OFFER})$$

Figure 12: Session typing rules with subtyping

$$(\text{SUB-OUT})\ \frac{\Gamma \leq \Gamma'}{x![\tau].\Gamma \leq x![\tau].\Gamma'} \qquad (\text{SUB-IN})\ \frac{\Gamma \leq \Gamma'}{x?[\tau].\Gamma \leq x?[\tau].\Gamma'}$$

Figure 13: Generic subtyping relation: additional rules for input and output

- *If* $P \equiv (\nu\tilde{x})\,(x^p?[z].P_1 \mid x^{\overline{p}}![y^q].P_2 \mid Q)$ *then* $\{x^p : ?[T].S, x^{\overline{p}}![T].\overline{S}, y^q : T\} \subseteq \Delta, \tilde{x} : \tilde{T}$*, and moreover, if* $p \in \{-,+\}$ *then* $\{x^+, x^-\} \cap \mathsf{fn}(Q) = \varnothing$.
- *If* $P \equiv (\nu\tilde{x})\,(x^p \triangleright \{l_i : P_i\}_{i\in I} \mid x^{\overline{p}} \triangleleft l.Q \mid R)$ *then* $p \in \{-,+\}$, $l \in I$ *and* $\{x^+, x^-\} \cap \mathsf{fn}(R) = \varnothing$.

On generic processes.

Definition A.4 (Correspondent properties) *An invariant predicate* p *on processes* correspond to *a consistency predicate* ok *on process types, if whenever* $\mathsf{ok}(\Gamma)$ *and* $\Gamma \triangleright P$ *then* $p(P)$ *holds.*

Theorem A.5 (Adapted from Theorem 4.1.2 of [6]) *Let some invariant* p *on processes correspond to an* ok *predicate. If* $\Gamma \triangleright P$ *and* $\mathsf{ok}(\Gamma)$*, then* $p(Q)$ *holds for every* Q *such that* $P \to^* Q$.

B Subtyping

Subtyping on session types. Recall the subtyping relation on session types (in Figure 11). Gay and Hole proved that subtyping is a preorder.

To incorporate subtyping into the source language, we modify the rules (T-IN), (T-OUT), and (T-OFFER). The new versions of these rules are in Figure 12.

Subtyping on process types. We extend the subtyping relation of the target language to take into account subtyping in input and output process types. The new rules are in Figure 13. It is simple to check that the relation is still a preorder.