

General Session Types

Giuseppe Castagna¹

Mariangiola Dezani-Ciancaglini²

Elena Giachino^{1,2}

Luca Padovani³

¹PPS (CNRS) - Université Denis Diderot - Paris, France

²Dipartimento di Informatica - Università degli Studi di Torino - Torino, Italy

³Istituto di Scienze e Tecnologie dell'Informazione - Università degli Studi di Urbino - Urbino, Italy

Abstract. We present a streamlined theory of session types based on a simple yet general and expressive formalism whose main features are semantically characterized and where each design choice is semantically justified. We formally define the semantics of session types and use it to define the subsessioning relation. We give a coinductive characterization of subsessioning and describe algorithms to decide all the key relations defined in the article. We then apply the theory to statically ensure progress for a simple π -based process calculus, give examples, and discuss related work.

1. Introduction

In distributed systems the communication between two points often consists of a conversation held on a channel and described by a protocol. Typical type systems for process algebras associate each channel with the type of messages exchanged through the channel. This practice either results in excessively strict requirements or it undermines the benefits of static typing by associating channels with less precise types. To obviate this problem Honda *et al.* introduced session types [15, 16].

Session types are used to type special channels through which several messages (of possibly different types) may be exchanged in sequence according to a given protocol. Such a session channel can be seen as a client-service connection, and the session type describes which actions the processes may perform through this channel and the order in which they are executed. We can thus assimilate the set of session types associated with the channels used by a process to a behavioural type of the process.

The current design of session types is somewhat *ad hoc* insofar as it heavily depends on programming primitives of the host language session types are used within. This hinders the adoption of session types in the existing general purpose languages unless these languages are extended by communication, synchronization and flow primitives specific to session types.

For these reasons we redesign session types to meet the following three criteria:

Language transparency. The introduction of session types must be transparently possible for every language that is able to emit signals: the introduction of session types must *only* imply the addition of new signals but not of new language primitives.

Type-checking independence. The (type-)checking of session composition must only rely on the existing language constructions: the description of how services are offered and served should spring from the analysis of the combination of the existing language primitives and not by the introduction of particular primitives whose goal is to handle the flow of the sessions.

Compositionality. Session types must be able to cope with possible ambiguities that may arise when creating new processes by composing existing ones. For instance this may happen if we create

a new process by composing two processes that offer some common signals, which raises the problem of how these signals are successively served (this is a problem akin to the disambiguation of multiple inheritance in object-oriented languages). Also, unrelated sessions/services may be given the same name (e.g. the name “search” is likely to be used for unrelated services) and the declared behaviour must allow the system to disambiguate their use.

Most importantly we do not want to pile up features in a bloated syntax but we want to define a clean formalism with a clear semantics: such a redesign is the ambitious goal of this work. How shall we proceed? We will start from scratch and design session types by taking into account what in our opinion is the ideal usage scenario of session types. In particular:

Three modalities of interaction. In our ideal scenario the only requirement for a programming language to use session types is that it must be able to emit a signal belonging to one of the following three classes corresponding to different modalities of interaction and then continue the flow of its computation. The classes are: (i) *session communication* where we find two signals $u!(e)$ and $u?(x : t)$ respectively to send the value of the expression e on the channel u and to receive on it a value of the type t and bind it to a variable x ; (ii) *session delegation*: if we want to have higher-order session, then we need a signal $u!(u')$ to delegate over a channel u the continuation of the session started on (channel) u' and a signal $u?(z : \eta)$ to receive on u the delegation of a session whose protocol is described by η and bind it to the variable x ; (iii) *session connection*, where we find signals of the form $c(z)$ by which a process manifests its will to initiate a session of name c —using the local name z —that will implement the behaviour described by the session type associated to c . Ultimately we require the host language to handle five new signals.¹

Three modalities of composition. Session types must describe *how* these signals are offered at a given point and how each signal is then served. For what concerns the way signals are offered by some service we can think of three different ways in which this may happen: (1) a packet of signals are offered in parallel which means that a client must handle all these signals to successfully interact with the service; (2) these packets are offered in sets, which means that the service gives the client the choice of on which packet to continue the session; (3) these sets are offered in a mutually exclusive way which means that the service will internally choose how to continue the session and the client must be ready to handle whatever choice the service has done. The way each signal emission is performed must be determined by

¹These signals are new with respect to the host language: they are instead pretty standard in the session type literature, though they are usually accompanied by other signals that explicitly handle the flow.

the type-checker according to the semantics of the host language composition primitives in which the emission occurs.

In this work we will drop the first modality, which corresponds to true concurrency, and handle parallel offers of signals via the interleaving of the signals. Therefore our session types will describe services that offer mutually exclusive sets of signals (each set corresponding to an internal choice of the service) among which the client is asked to choose (corresponding to an external choice proposed to the client). To comply with the criterion of “type-checking independence”, it is the task of the (session-)type-checker to analyse a process and infer the way (together or mutually exclusively) the process offers signals at each step.

Dynamic selection of the behaviour based on the received values.

We want to provide a fine-grained description of the flow of a session, so as to be able to type the composition of processes that use a same channel but in different ways: the exchanged value will determine which process the interaction is intended for and, thus, the continuation of the session. In other terms, we want to be able to describe sessions that have different continuations depending on the type of the received values. In order to implement such a fine-grained description of the computation flow, the types appearing in signals will be Boolean combinations of the types of the host language as well as singleton types (the latter, to handle value-level branching and to encode current definitions of session types present in the literature, where labels are the values used for branching). So we will consider signals such as, for instance, $u?(x : (\text{Bool} \vee \text{Int} \wedge \neg 0))$ corresponding to the reception on u of either a Boolean value or of a non-null integer.

Substitutability. New services must be obtained by assembling existing ones. Session types are designed to limit composition so that all assemblages are sound (no type mismatch in communications) and stuck free (no session deadlock). But of course they must not be rigid and must allow, within the constraint of soundness and stuck freedom, maximum reuse. Therefore it is necessary to define a type hierarchy that allows the programmer to use/replace services of a more precise type where services of a less precise type are expected. In particular we want to be able to upgrade a service to a new one that offers its clients more choices and/or has a more deterministic execution (by reducing the range of its internal choices). All of this must be obtained transparently for existing clients.

In this work we show how to fulfil all these requirements and design a theory of session types that can be applied to our ideal usage scenario allowing flexible composition and substitutability of services, yet ensuring soundness and stuck freedom for all interactions.

Outline of the presentation: After outlining our contribution and discussing related work we start the formal study in Section 2, devoted to session types. We formally define their syntax (§2.1) and semantics where we introduce the pivotal definition of *duality* and use it to give a semantic and quite intuitive characterization of subsessioning (§2.2). Subsessioning is required for defining the subtyping relation (§2.3), but its semantic characterization is hard to work with and gives little intuition about the properties it enjoys. Thus we devise a coinductive characterization of subsessioning and prove that it is equivalent to the semantic one (§2.4). We describe some interesting subsessioning relations (§2.5) which allow us to rewrite session types in an equivalent normal form. The normal form is at the heart of the algorithmic characterizations of all the given relations (§2.6). In Section 3 we apply our theory of general session types to a minimal process calculus describing interacting services. We devise the language syntax (§3.1), its operational semantics (§3.2), and a typing discipline which enforces the progress property for well-typed processes (§3.3). A conclusion closes our presentation. **Omitted proofs and inferences rules can be found in the version at the authors’ web pages.**

1.1 Contributions and related work

Our contribution is the definition of a clean and semantically grounded general theory for session types. We claim that we singled out a minimal set of features to define session types, insofar as the addition of any further constructor (such as a parallel composition or labelled synchronization) would restrict the programming language to which the framework could be applied, while the removal of any of them would jeopardize its generality. As we said the three kinds of signals we use are more or less present in all the works on session types, but the point is that they are always accompanied by other signals (e.g. labelled choices), constructors (e.g. parallel composition) or features (e.g. choices based on object classes) that tailor the type system to a restricted class of languages. Our aim and ambition is to synthesise the essential features of the session type literature and develop their minimal, and therefore general, theory. We already argued about the minimality of the actions (signals) described by our session types. Similarly we will argue why the two choices of composition of session types (internal and external choice) we included in our theory seem to be minimal ingredients of any general theory of session types. Our study focuses on the type theory that we want to be generally applicable to a large class of languages: the definition of a process calculus given at the end of the presentation plays a marginal role in this work and it is given as a instance of possible application of the theory. We hope and believe that the framework and techniques defined here may become a boilerplate of solutions for session types.

Subtyping for session types is studied in [13, 12] where the definition of the subtyping relation is driven by the observation that one can always safely replace a session by another that externally offers more choices and internally can make less choices. Since in the cited works choices are driven by labels, it turns out that external and internal choices have the same subtyping relation as record and variant types respectively. Here we work with more general choices which are based on the arguments of the communications rather than on labels:² when a session offers an output it will be able to synchronize only with the branches of a choice that accept that output. While the resulting subtyping relation is driven by the same principles as stated above, our new setting yields a richer and more general subtyping relation where a single or a set of choice branches may subsume another set of branches. Another difference is that in the cited works the subtyping relation is defined coinductively and axiomatically while here we characterize the relation semantically, and this better shows the underlying intuition.

With respect to concurrency theory we introduce an original treatment of output signals, by implementing a form of *partial asynchrony*. This treatment is similar to the one proposed by Castellani and Hennessy [5] for asynchronous CCS, where outputs cannot be blocked even if they guard external choices (we call this property “output irrevocability”). However, in our setting output signals are allowed to have a continuation. Thus the order of actions specified by a session type must be strictly followed (equivalence of session types modulo permutation of consecutive outputs is left for future work). Technically, this corresponds to be able to observe inputs even in the presence of (partial) asynchrony.

From a technical viewpoint in this work we introduce several novelties. We devise a new labelled transition system *for session descriptors* in which actions represent values rather than types, we give a semantic characterization of the subsessioning relation in terms of a set-theoretic interpretation of session descriptors. The same interpretation is used to give semantics to a complete set of Boolean operators for session descriptors. The process language mixes and synthesizes several techniques that are scattered all over

² In our system label-based choices are a special cases in which the branches of a choice are selected on singleton types.

the literature. In particular it borrows the type-based dynamic selection of external choices and the technique of tagged channels from [2] and the use of polarized channels to ensure subject reduction from [13]; its typing discipline improves existing stack-based typing techniques [10], by sparing redundancies and resulting in very compact and, we believe, relatively readable rules.

2. Session types

2.1 Type syntax

As we said in the introduction we want to add to some host language the following signals: $u!(e)$, $u?(x : t)$, $u!(u)$, $u?(z : \eta)$, and connect $c(z)$. Then we associate every channel— c , z , or u —with a prescription of its behaviour, a session type or a session descriptor, so as to ensure that, whenever a session is started, it carries on soundly interacting until it terminates.³ The behavioural prescriptions characterize all the possible conversations that may take place on channels and have the form prescribed by the metavariable η for session descriptors and $\text{begin}.\eta$ for session types in Table 1.

(types)	t	::=	$\dots \mid \text{begin}.\eta \mid \neg t \mid t \wedge t \mid t \vee t \mid v$
(descriptors)	η	::=	$\text{end} \mid \alpha.\eta \mid \eta \oplus \eta \mid \eta + \eta$
(actions)	α	::=	$!t \mid ?t \mid !\chi \mid ?\chi$
(sieves)	χ	::=	$\eta \mid \neg\chi \mid \chi \wedge \chi \mid \chi \vee \chi$

Table 1. Type syntax

Two different kinds of communication can take place on channels: a channel is used either (*i*) to send/receive some value of type t (signals $u!(\cdot)$ and $u?(\cdot)$) or (*ii*) to “delegate”/“resume” some open session to a different process (signals $u!(\cdot)$ and $u?(\cdot)$). In the behavioural type of a channel we use $?t$ and $!t$ to denote that the (process using the) channel will respectively wait for and send some value of type t , and use $? \eta$ and $! \eta$ (actually, $? \chi$ and $! \chi$, see later on) to denote that the (process that uses the) channel will respectively wait for and send some channel which already started a conversation and will continue it according to the behaviour described by the session descriptor η . In particular, a descriptor $\alpha.\eta$ states that the (process using the) channel will perform one of the communication actions α described above and then will behave according to η ; a descriptor end states that the session on the channel has successfully ended; a descriptor $\eta_1 \oplus \eta_2$ states that the (process that uses that) channel will internally choose to behave according to either η_1 or η_2 ; a descriptor $\eta_1 + \eta_2$ states that the (process that uses that) channel gives to the communicating partner the choice to behave according to either η_1 or η_2 . In what follows we adopt the convention that the prefix operator has precedence over the choice operators and we will use parentheses for enforcing precedence. For instance, $(!t.\eta) + \text{end}$ and $!t.\eta + \text{end}$ denote the same session descriptor, which is different from $!(\eta + \text{end})$. Types t are inherited from the host language (this is stressed in Table 1 by the dots in the production for types), to which we add singleton types (denoted by a value v , the only one they contain), Boolean combinators (i.e. \vee , \wedge , and \neg), and *session types* of the form $\text{begin}.\eta$ which classify yet-to-be-opened session channels whose conversation follows the descriptor η . The interest of session types is that they can be used to type higher-order communications in which session channels are communicated over other channels; session types will also extend the type system of the host language which can thus use session channels as first class values.

³Since sessions may be nested, termination will be ensured under the hypothesis that every subsession of a started session will eventually start and terminate as well (see Theorem 3.1).

The importance of Boolean combinators for types is shown by the following example where we assume Int to be a subtype of Real :

$$?\text{Real}.! \text{Int}.\text{end} + ? \text{Int}.! \text{Bool}.\text{end} \quad (1)$$

The session descriptor above declares that if a process (that uses a channel with that behaviour) receives a real number, then it will answer by sending an integer, while if it receives an integer it will answer by sending a Boolean. A partner process establishing a conversation on such a channel knows that if it sends a real that is not an integer, then it should be ready to receive an integer while if it sends an integer, then it must be ready to receive an integer or a Boolean value (notice how the type of the argument drives the selection of the external choice). That is, its conversation will be represented by the following descriptor ($t \setminus s$ stands for $t \wedge \neg s$):

$$!(\text{Real} \setminus \text{Int}).? \text{Int}.\text{end} + ! \text{Int}?(\text{Bool} \vee \text{Int}).\text{end} \quad (2)$$

We see that Boolean combinators immediately arise when describing the behaviour of an interacting process. They are also useful when considering equivalences. For instance, it is intuitively clear that (1) is equivalent to

$$?(\text{Real} \setminus \text{Int}).! \text{Int}.\text{end} + ? \text{Int}!(\text{Bool} \vee \text{Int}).\text{end} \quad (3)$$

But the crucial role of Boolean combinators can be shown by slightly modifying (1) so that it performs only input actions:

$$?\text{Real}.? \text{Int}.\text{end} + ? \text{Int}.? \text{Bool}.\text{end} \quad (4)$$

In this case the descriptor declares that after receiving an integer it will either wait for another integer or for a Boolean value. If an interacting process sends an integer, then in order to be sure that the conversation will not be stuck it must next send a value that is both an integer and a Boolean. Since there is no such a value, the only way to successfully interact with (4) is to make sure that interacting processes will only send reals that are not integers: $!(\text{Real} \setminus \text{Int}).! \text{Int}.\text{end}$. In conclusion, the only way to describe the sessions that can successfully interact with (4) is to use negation (for the sake of completeness note that (4) is equivalent to $?(\text{Real} \setminus \text{Int}).? \text{Int}.\text{end} + ? \text{Int}?(\text{Bool} \wedge \text{Int}).\text{end}$ which is equivalent to $?(\text{Real} \setminus \text{Int}).? \text{Int}.\text{end}$ since the right summand of the previous choice can never successfully complete a conversation). A similar discussion can be done for delegation, that is, when actions are over session descriptors, rather than types. This is why we added Boolean combinations of session descriptors too (we dub them *sieves*) and actions have the form $? \chi$ and $! \chi$ rather than $? \eta$ and $! \eta$.

We want both types and session descriptors to be recursively definable. This is important for types since it allows us to represent recursive data structures (e.g., DTDS) while for session descriptors it allows us to represent services that provide an unbounded number of interactions such as (the service whose behaviour is the solution of the equation) $\eta = \text{end} + ? \text{Int}.\eta$ which describes a session that accepts as many integers as wished by the interacting process. In order to support recursive terms, we resort to a technique already used in [11, 4] where instead of introducing an explicit finite syntax for recursive terms, we directly work with possibly infinite regular term trees that satisfy some contractivity conditions; these conditions ensure that terms are semantically meaningful. This yields to the following definition for our types:

DEFINITION 2.1 (Types). *The types of our system are the possibly infinite regular trees coinductively generated by the productions in Figure 1 that satisfy the following conditions:*

1. on every infinite branch of a type there are infinitely many occurrences of “begin”;
2. on every infinite branch of a session descriptor there are infinitely many occurrences of “.” (the prefix constructor);

3. for every subterm of the form $\alpha.\eta$, the tree $\alpha.\eta$ is not a subtree of α .

The first two conditions are contractivity restrictions that rule out meaningless terms such as (the solutions of the equations) $t = t \vee t$ or $\eta = \eta \oplus \eta$; technically they provide a well-founded order we use in proofs. The third condition states that recursion cannot escape prefixes and thus it rules out terms such as $\eta = ?\eta.\text{end}$; this restriction generalizes the typing technique used in all works on (recursive) session types that forbids delegation of a channel over itself [16, 19] (strictly speaking we disallow types that in the cited works are not inhabited by any program) while, technically, it allows us to stratify the definition of the subtyping and subsessioning relations, stratification used in the proof of Theorem 2.6.

We do not specify any particular property for the types of the host language. Of course, if the host language has some type constructors (e.g. products, arrows, etc.) the first contractivity condition can be relaxed to requiring that on every infinite branch there are infinitely many occurrences of type constructors. The only condition that we impose on the host language is on values (thus those of the host language as well) which must satisfy the following *strong disjunction property* for unions:

$$\vdash v : t_1 \vee t_2 \iff \vdash v : t_1 \text{ or } \vdash v : t_2 \quad (5)$$

This condition *may* be restrictive only in the case that the host language already provides a union type combinator since, otherwise, it can be easily enforced by requiring that every session channel is associated with exactly one (most specific, because of subtyping) session type.

Henceforward, we will use t to range over *types*, θ and η to range over *session descriptors*, χ to range over *sieves*, ψ to range over all of them, and often omit the word “session” when speaking of session descriptors. We reserve v for values, whose definition and typing is left unspecified: we assume as understood that values for a *session type* $\text{begin}.\eta$ are channels explicitly associated with or tagged by that type (or, because of subtyping, by a $\text{begin}.\eta'$ subtype of $\text{begin}.\eta$: more about that later on).

As we already hinted in the introduction we do not include in our session descriptors a construct for parallel composition (as for instance it is done in [17, 1]). Indeed while we think that internal and external choices are necessary to safely approximating the behaviour of a generic session (a general service must be able to offer some choices to a client and, according to the interaction with the client, make some internal choices that determine the prosecution of the session), we reckon that the introduction of parallel composition would limit the application of our theory to fewer programming languages. The reason is that session interaction is a two-parties synchronization, therefore it can mostly be simulated by internal and external choices via some expansion laws. If session atomic synchronization involved more than two parties, then this would no longer be true. By not introducing a parallel composition we let different type systems to use different expansion laws and thus type different kinds of parallel composition of processes (interleaving, restricted parallelism, asymmetric parallelism, and so on): if we added a parallel composition to our types we would thus fix its semantics and limit the application of our theory only to calculi/languages in which the parallel composition of processes had a matching semantics. In this respect we completely embrace the conclusions of [14].

2.2 Type and session semantics

The semantics of both session descriptors and types—and more generally most of the constructions of this work—crucially relies on the notion of *duality*. In this section we first informally define duality to devise a somewhat informal denotational semantics for

types and descriptors, then we give the formal definition of duality in terms of a labelled transition system for descriptors.

2.2.1 Set-theoretic interpretations

In the previous section we argued that a complete set of Boolean combinators must be used if we want to describe the set of partners that safely interact with a given descriptor. Since we want the semantics of Boolean combinators to be intuitive and easy to understand we base their definition on a set-theoretic interpretation. In particular, we interpret every type constructor as the set of its values and the Boolean combinators as the corresponding set-theoretic operations. In other terms, we seek for an interpretation of types $\llbracket \cdot \rrbracket$ such that $\llbracket t \rrbracket = \{v \mid \vdash v : t\}$ and that $\llbracket t \wedge s \rrbracket = \llbracket t \rrbracket \cap \llbracket s \rrbracket$, $\llbracket t \vee s \rrbracket = \llbracket t \rrbracket \cup \llbracket s \rrbracket$, and $\llbracket \neg t \rrbracket = \mathcal{V} \setminus \llbracket t \rrbracket$ (where \mathcal{V} denotes the set of all values). The same interpretation can then be used to *define* the subtyping relation (denoted by “ $<$ ”). That is

$$t <: s \stackrel{\text{def}}{\iff} \llbracket t \rrbracket \subseteq \llbracket s \rrbracket$$

The technical machinery to define an interpretation with such properties and solve the several problems its definition raises (e.g. the circularity between the subtyping relation and the typing of values) already exists and can be found in the work on Semantic Subtyping [11]: we take it for granted and no longer bother about it if not for session types that are dealt with in Section 2.3. The interpretation of types justifies the use we do henceforward of the notation $v \in t$ to denote that v has type t .

The next problem is to give a set-theoretic interpretation to session descriptors, as we have Boolean combinations on them too. This interpretation is not required to be precise or mathematically meaningful but only to ensure that conversations do not get stuck. To this aim, rather than giving the set of values (or whatever they would be, since session descriptors classify just “chunks” of conversation) contained in a descriptor, it suffices to characterize all the possible behaviours common to all channels that implement a session. In other terms, the semantics of a session descriptor can be characterized by the set of partners with whom the interaction will never get stuck (a sort of realizability semantics). This is captured by the notion of *duality*: two session descriptors η and θ are *dual* if any conversation between two channels which follow respectively the prescriptions of η and θ will never get stuck. So, for instance, the descriptor (1) in the previous section is dual to the descriptor (2). But $! \text{Int}.\text{?(Bool} \vee \text{Int)}. \text{end}$ is dual to (1), too. Note also that some session descriptors have no dual, e.g. $\text{?(Bool} \wedge \text{Int)}. \text{end}$, since no process can send a value that is both a Boolean and an integer (the intersection is empty). Such descriptors constitute a pathological case, since no conversation can take place on channels conforming to them. Thus we will focus our attention on descriptors for which at least a dual exists, and that we dub *viable descriptors*. We write $\eta \bowtie \theta$ if η and θ are dual (clearly, duality is a symmetric relation). Then, we can define the interpretation of a descriptor as the set of its duals: $\llbracket \eta \rrbracket = \{\theta \mid \eta \bowtie \theta\}$; extend it set-theoretically to sieves: $\llbracket \chi \wedge \chi' \rrbracket = \llbracket \chi \rrbracket \cap \llbracket \chi' \rrbracket$, $\llbracket \chi \vee \chi' \rrbracket = \llbracket \chi \rrbracket \cup \llbracket \chi' \rrbracket$, $\llbracket \neg \chi \rrbracket = \mathcal{S} \setminus \llbracket \chi \rrbracket$ (where \mathcal{S} denotes the set of all *viable* descriptors); and use it to semantically define the subsieving (and subsessioning) relation (denoted by “ \leq ”):

$$\chi \leq \chi' \stackrel{\text{def}}{\iff} \llbracket \chi \rrbracket \subseteq \llbracket \chi' \rrbracket \quad (6)$$

Duality plays a central role also in defining the semantics of types. Indeed we said that the semantics of a type constructor is the set of its values, hence we have to define the values of the type constructor $\text{begin}.\eta$. As we hinted in Section 2.1 we can take as a value of a session type a channel tagged by that type *or by a subtype*. Therefore to define values we need to determine when a session type is subtype of another, that is, when we can safely use a channel of some session type where a channel of

different type is expected. The intuition is that a channel of the smaller type must be less “tolerant” than the one it replaces, that is it must accept the same or less inputs and return the same or more outputs. Since it “can offer less and do more” it will also be more demanding with its duals, so it will have a set of duals smaller than or equal to the duals of the one it replaces, as less descriptors will comply with the extra requirements it imposes. So the intuition—that we formalize by equation (11) in Section 2.3—is that $\text{begin}.\eta <: \text{begin}.\eta'$ if and only if $\eta \leq \eta'$. Since we want our types to satisfy the strong disjunction property (5), then channels must be tagged by types of the form $\text{begin}.\eta$ (and not, say, $\text{begin}.\eta \vee \text{begin}.\eta'$), which yields the following interpretation for session types: $\llbracket \text{begin}.\eta \rrbracket = \{c^{\text{begin}.\eta'} \mid \eta' \leq \eta\}$, that is

$$\llbracket \text{begin}.\eta \rrbracket = \{c^{\text{begin}.\eta'} \mid \forall \theta, \theta \bowtie \eta' \Rightarrow \theta \bowtie \eta\} \quad (7)$$

The next step is to formally define the duality relation for which we have to characterize the observables of the session descriptors.

2.2.2 Semantics of session descriptors

The formal semantics of a descriptor can be given by resorting to the labelled transition system (LTS) defined by the rules in Table 2 plus the symmetric of rules (TR2-TR6). In the table μ ranges over actions of the form $!v$, or $?v$, or $! \eta$, or $? \eta$, or \checkmark .

(TR1) $\frac{}{\text{end} \xrightarrow{\checkmark} \text{end}}$	(TR2) $\frac{}{\eta \oplus \eta' \longrightarrow \eta}$	(TR3) $\frac{\eta \longrightarrow \eta'}{\eta + \eta'' \longrightarrow \eta' + \eta''}$	
(TR4) $\frac{\eta \xrightarrow{\mu} \eta'}{\eta + \eta'' \xrightarrow{\mu} \eta'}$	(TR5) $\frac{\eta \xrightarrow{!v}}{\eta + \eta' \longrightarrow \eta}$	(TR6) $\frac{\eta \xrightarrow{! \eta''}}{\eta + \eta' \longrightarrow \eta}$	
(TR7) $\frac{v \in t}{?t.\eta \xrightarrow{?v} \eta}$	(TR8) $\frac{v \in t}{!t.\eta \xrightarrow{!v} \eta}$	(TR9) $\frac{\eta \in \chi}{? \chi.\eta' \xrightarrow{? \eta} \eta'}$	(TR10) $\frac{\eta \in \chi}{! \chi.\eta' \xrightarrow{! \eta} \eta'}$

Table 2. Labelled transition system for session descriptors.

Rules (TR1-TR4) are straightforward: `end` emits a “tick” (TR1); an internal choice silently decides the behaviour it will successively follow (TR2); an external choice either performs an internal silent move (TR3) or it emits a signal μ that it offers as a possible choice to the interacting partner (TR4). Note that internal moves in one branch of an external choice do not preempt the behaviour of the other branch. This is typical of process languages with two distinct choice operators, such as CCS without τ 's [7].

The remaining rules are somewhat less common. Rules (TR7-TR8) state that the synchronization is performed on single values (strictly speaking, on singleton types) rather than on types. This is closer to what happens in practice, since $!t.\eta$ indicates that the descriptor is ready to emit some value of type t (TR8), while $?t.\eta$ indicates that the descriptor is ready to accept any value of type t (TR7). While this approach is reminiscent of the so-called *early semantics* in process algebras [18] (but note that here it is applied at type level rather than at process level), there is a technical reason to use values rather than types, which we explain after defining the subsessioning relation.

Rules (TR9-TR10) follow the same idea as (TR7-TR8), and state that actions on descriptors emit a more precise information than what they declare. To understand this point we need to give some more details. First note that a session descriptor η , despite it is usually called “session type” in the literature, is not a “real” type

since it does not type any value. Session descriptors do not classify values but, rather, they keep track of the residual conversation that is allowed on a given session channel (whose “real” type is of the form $\text{begin}.\eta$). Therefore we cannot directly apply the same technique as for rules (TR7-TR8) since there does not exist any value for session descriptors. To mimic the behaviour of rules (TR7-TR8) we resort to the informal semantics we described in Section 2.2.1 where a type is interpreted as the set of its values and a descriptor—actually, a sieve—as the set of its duals: therefore, an action on a type emits the same action on its values, so an action on a sieve emits the same action on its duals, where we use $\eta \in \chi$ to denote that $\eta \in \llbracket \chi \rrbracket$.

Rules (TR5-TR6) state that outputs are irrevocable. This is a characteristic peculiar to our system and is reminiscent of Castelli and Hennessy’s treatment of external choices in the asynchronous CCS [5]. Roughly speaking, imagine a process offering two different outputs in an external choice. Then we can think of two possible implementations for such a choice. In one case the choice is an abstraction for a simple handshaking protocol that the communicating processes engage in order to decide which value is exchanged. This implementation does not fit very well a distributed scenario where processes are loosely coupled and communication latency may be important. In the second—and in our opinion closer to practice—case, the sender process autonomously decides which value to send. Rules (TR5-TR6) state that the decision is irrevocable in the sense that the sender cannot revoke its output and try with the other one. This behaviour is obtained by rules (TR5-TR6) by assimilating an external choice over output actions to an internal choice in which the process silently decides to send some particular value. In this respect the symmetry of input and output actions in rules (TR7-TR8)—but the same holds for (TR9-TR10) as well—may be misleading: we implicitly assumed that when a process waits for a value of type t it is ready to accept *any* value of type t (the choice of the particular value is left to the sender) while when a process sends a value of type t , it internally decides *a particular* value of that type. We will break this symmetry in the formal notion of duality (Definition 2.5) to be defined next.

2.2.3 Duality

The discussion on the labelled transition system suggests that two dual descriptors can either agree on termination (so both emit \checkmark) or one of the two descriptors autonomously chooses to send an output that the other descriptor must be ready to receive. In order to formalise the notion of duality it is then handy to characterise outputs (when an output action *may* happen) and inputs (when an input action *must* happen). As usual we write \Longrightarrow for the reflexive and transitive closure of \longrightarrow ; we write $\xRightarrow{\mu}$ for $\Longrightarrow \xrightarrow{\mu} \Longrightarrow$; we write $\eta \xrightarrow{\mu}$ if there exists η' such that $\eta \xrightarrow{\mu} \eta'$, and similarly for $\xRightarrow{\mu}$; we write $\eta \not\rightarrow$ if there exists no η' such that $\eta \longrightarrow \eta'$.

DEFINITION 2.2 (May and Must Actions). *We say that η may output μ , written $\eta \downarrow \mu$, if there exists η' such that $\eta \Longrightarrow \eta' \not\rightarrow$ and $\eta' \xrightarrow{\mu}$ and μ is either $!v$, or $! \eta$, or \checkmark .*

We say that η must input μ , written $\eta \Downarrow \mu$, if $\eta \Longrightarrow \eta' \not\rightarrow$ implies $\eta' \xrightarrow{\mu}$ and μ is either $?v$, or $? \eta$, or \checkmark .

As usual we write $\eta \not\downarrow \mu$ if not $\eta \downarrow \mu$ and $\eta \not\Downarrow \mu$ if not $\eta \Downarrow \mu$.

Intuitively $\eta \downarrow \mu$ states that for a particular internal choice η will offer an output μ as an option, while $\eta \Downarrow \mu$ states that the input μ will be offered whatever internal choice η will do. For example $! \text{Int}.\text{end} \oplus \text{end} \downarrow !3$ and $! \text{Int}.\text{end} \oplus \text{end} \downarrow \checkmark$; on the other hand we have $! \text{Int}.\text{end} + \text{end} \not\downarrow \checkmark$, since $! \text{Int}.\text{end} + \text{end} \not\rightarrow \text{end}$. Similarly we have $? \text{Int}.\text{end} \oplus ? \text{Real}.\text{end} \Downarrow ?3$ because the action $?3$ is always guaranteed independently of the internal choice, whereas

$?Int.end \oplus ?Real.end \Downarrow \sqrt{2}$ because $?Int.end \oplus ?Real.end \longrightarrow ?Int.end$ and $?Int.end \Downarrow \sqrt{2}$.

The previous definition induces two notions of convergence. Clearly convergence is a necessary condition for a session descriptor to have a dual.

DEFINITION 2.3 (May and Must Converge). *We say that η may converge, written $\eta \downarrow$, if for all η' such that $\eta \Longrightarrow \eta' \not\rightarrow$ we have $\eta' \downarrow \mu$ for some μ . We say that η must converge, written $\eta \Downarrow$, if $\eta \downarrow \mu$ for some μ . As usual, we use $\eta \Downarrow$ and $\eta \Downarrow$ to denote their respective negations.*

Note that the two contractivity conditions of Definition 2.1 rule out behaviours involving infinite sequences of consecutive internal decisions. Therefore we will only consider strongly convergent processes, namely processes for which there does not exist an infinite sequence of \longrightarrow reductions.

The labelled transition system describes the *subjective evolution* of a session descriptor from the point of view of the process that uses a communication channel having that (residual) type. The last notion we need allows us to specify the evolution of a session descriptor from the dual point of view of the process at the other end of the communication channel. For example, we have $?Real.!Int.end + ?Int.!Bool.end \xrightarrow{?3} !Bool.end$ (the process receiving the integer value 3 knows that it has taken the right branch and now will send a Boolean value). However, the process sending the integer value 3 on the other end of the communication channel does not know whether the receiver has taken the left or the right branch, and both branches are actually possible. From the point of view of the sender, it is as if the receiver will behave according to the session descriptor $!Int.end \oplus !Bool.end$, which accounts for all of the possible states in which the receiver can be. The *objective evolution* of a session descriptor after an action μ is defined next.

DEFINITION 2.4 (Successor). *Let $\eta \xrightarrow{\mu}$. The successor of η after μ , written $\eta\langle\mu\rangle$, is defined as: $\eta\langle\mu\rangle = \oplus \{\eta' \mid \eta \xrightarrow{\mu} \eta'\}$.*

For example, we have $(?Real.!Int.end + ?Int.!Bool.end)\langle?3\rangle = !Int.end \oplus !Bool.end$ but $(?Real.!Int.end + ?Int.!Bool.end)\langle\sqrt{2}\rangle = !Int.end$. Note that $\eta\langle\mu\rangle$ is well defined because there is always a finite number of residual η' such that $\eta \xrightarrow{\mu} \eta'$. This is a direct consequence of the contractivity conditions on session descriptors.

We now have all the ingredients for formally defining duality.

DEFINITION 2.5 (Duality). *Let the dual of a label μ , written $\bar{\mu}$, be defined by: (i) $\bar{\checkmark} = \checkmark$; (ii) $\bar{\dagger v} = \dagger v$; (iii) $\bar{\dagger \eta} = \dagger \eta$; where $\bar{!} = ?$ and $\bar{?} = !$. Then $\eta_1 \bowtie \eta_2$ is the largest relation between session descriptors such that one of the following condition holds:*

1. $\eta_1 \Downarrow \checkmark$ and $\eta_2 \Downarrow \checkmark$;
2. $\eta_1 \downarrow$ and $\eta_1 \downarrow \mu$ implies $\eta_2 \downarrow \bar{\mu}$ and $\eta_1\langle\mu\rangle \bowtie \eta_2\langle\bar{\mu}\rangle$;
3. $\eta_2 \downarrow$ and $\eta_2 \downarrow \mu$ implies $\eta_1 \downarrow \bar{\mu}$ and $\eta_1\langle\bar{\mu}\rangle \bowtie \eta_2\langle\mu\rangle$.

The intuition behind the above definition is that a dual *must* accept every input that its partner *may* output, or they must both agree on termination. Conditions (2) and (3) are the same requirement, and both of them are needed to ensure symmetry of duality. For example, we have $?Real.!Int.end + ?Int.!Bool.end \bowtie !Int.(Int \vee Bool).end$, but $?Real.!Int.end + ?Int.!Bool.end \not\bowtie !Int.!Int.end$ because the descriptor on the right is not sure that its partner will answer with an integer. However $?Real.!Int.end + ?Int.!Bool.end \bowtie !(Real \setminus Int).?Int.end$. As another example, we have $?Int.end \oplus ?Real.end \bowtie !Int.end$ because $?Int.end \oplus ?Real.end \Downarrow \forall$ for every $v \in Int$, however $?Int.end \oplus ?Real.end \not\bowtie !\sqrt{2}.end$ because $?Int.end \oplus ?Real.end \Downarrow \sqrt{2}$.

The reader may have observed that there is a circularity in the definitions of duality and of the labelled transition system. This

is evident in rules (TR7-TR8) since the rules emit a dual of the sieve; that is, the relation $\eta \in \chi$ is defined in terms of $\llbracket \chi \rrbracket$ whose definition is given in terms of the duality relation. Less evident is the circularity of rules (TR9-TR10), where it resides in the fact that these rules emit values of a given type; if this type has the form $begin.\eta$, then its values are all the channels of the form $c^{begin.\eta'}$ such that $\theta \bowtie \eta'$ implies $\theta \bowtie \eta$ for all θ (cf. equation (7)): so also the definition of the relation $v \in t$ depends on that of duality. The following theorem proves that this circularity is not one.

THEOREM 2.6 (Well-foundedness). *The definitions of $\eta \in \chi$, $v \in t$ and $\eta \bowtie \eta'$ are well founded.*

A corollary of this theorem is that the definitions of subsessioning $\eta \leq \eta'$ and subsieving $\chi \leq \chi'$ (the former being a special case of the latter) given by the equation (6) in Section 2.2.1 are well founded as well.

The notion of duality is also a useful tool for better understanding the semantics of session descriptors. Let us revisit part of the LTS in the light of duality:

Output of values. Rules (TR7-TR8) in Section 2.2 state that a descriptor is ready to respectively input and output some value of type t . If prefixes emitted the specified type rather than values, then it would no longer be possible to prove the following equations (where “=” denotes the equality induced by the relation \leq):

$$?(t_1 \vee t_2).\eta = ?t_1.\eta + ?t_2.\eta \quad (8)$$

$$!(t_1 \vee t_2).\eta = !t_1.\eta + !t_2.\eta \quad (9)$$

$$!(t_1 \vee t_2).\eta = !t_1.\eta \oplus !t_2.\eta \quad (10)$$

In particular, the righthand-sided descriptors would no longer be smaller than the lefthand-sided ones. Consider for example an instance of (9) where we take $t_1 \equiv Int$ and $t_2 \equiv Bool$ (here and henceforward we use “ \equiv ” to denote syntactic equality). It is clear that the two descriptors in the equation share the same set of duals (that is, they have the same semantics): the duals of both descriptors are descriptors that accept both an integer and a Boolean and then are dual of η . With the current definition of the LTS this holds true: whatever signal the lefthand descriptor emits will be matched by every dual of the righthand since, in both cases, these signals will be on values of type $Int \vee Bool$. Here is where the strong disjunction property on union types (5) is used: the lefthand descriptor cannot emit a value that is neither an Int nor $Bool$. If the transition system had emitted types rather than values, then the duals of the righthand descriptor would not be able to match the signal $!(Int \vee Bool)$ since each summand of the righthand descriptor could at most emit Int or $Bool$. We could have introduced some extra definition of sets of emitted signals and saturated these sets with unions for internal and external sums, but the current solution avoids all this clutter.

A similar reasoning holds for rules (TR9-TR10) because of the disjunction property we are going to prove next.

Internal choices, intersections, and disjoint unions. Using the definition of duality it is easy to see that $\llbracket \eta \oplus \eta' \rrbracket = \llbracket \eta \wedge \eta' \rrbracket$ since the duals of an internal choice must comply with both possible choices and thus be duals of both of them. Using this property it is easy to prove that sieves satisfy a disjunction property even stronger than the one for types, as the disjunction holds not only for single elements but for all the subsets of a union:

PROPOSITION 2.7. $\theta \leq \chi_1 \vee \chi_2 \iff \theta \leq \chi_1$ or $\theta \leq \chi_2$.

This property plays a crucial role in proving decidability of \leq .

2.3 Subtyping

Now that we have defined the duality relation, and therefore sub-sessioning, we can also formally define the subtyping relation.

The types defined in Section 2.1 include three type combinators (union, intersection, and negation), one type constructor $\text{begin}.\eta$, plus other basic types and type constructors that we left unspecified (typically, real , bool , \times , \dots). We define the subtyping relation semantically using the technique defined in [11] and outlined in Section 2.2.1, according to which types are interpreted as the set of their values, type combinators are interpreted as the corresponding set-theoretic operations, and subtyping is interpreted as set containment. As a consequence, testing a subtyping relation is equivalent to testing whether a type is empty, since by simple set-theoretic transformations we have that $t_1 <: t_2$ if and only if $t_1 \wedge \neg t_2 <: \emptyset$ (where we use \emptyset to denote the empty type, that is the type that has no value). Again by simple set-theoretic manipulations, every type can be rewritten in disjunctive normal form, that is a union of intersections of types. Furthermore, since type constructors are pairwise disjoint (there is no value that has both a session type and, say, a product type—or whatever type constructor is inherited from the host language), then these intersections are uniform since they intersect either a given type constructor, or its negation (see [3, 11] for details). In conclusion, in order to define our subtyping relation all we need is to decide when $\bigvee_{k \in K} (\bigwedge_{i \in I_k} \text{begin}.\eta_i \wedge \bigwedge_{j \in J_k} \neg \text{begin}.\eta_j) <: \emptyset$. Since a union of sets is empty if and only if every set in the union is empty, by applying the usual De Morgan laws we can reduce this problem to deciding the inclusion $\bigwedge_{i \in I} \text{begin}.\eta_i <: \bigvee_{j \in J} \text{begin}.\eta_j$.

As regards session channels, we notice that a value has type $(\text{begin}.\eta) \wedge (\text{begin}.\eta')$ if and only if it has type $\text{begin}.\eta \oplus \eta'$. By the strong disjunction property on union types (5) a session channel is in the union of $\text{begin}.\eta$ types if it is in a particular $\text{begin}.\eta$ of this union. Therefore the semantic subtyping relation for the types of Section 2.1 is completely defined by (the semantic subtyping framework of [11] and) the following equation

$$\bigwedge_{i \in I} \text{begin}.\eta_i <: \bigvee_{j \in J} \text{begin}.\eta_j \iff \exists j \in J : \bigoplus_{i \in I} \eta_i \leq \eta_j \quad (11)$$

The equation above may be better understood by inspecting the special case when I and J are singletons:

$$\text{begin}.\eta_1 <: \text{begin}.\eta_2 \iff \eta_1 \leq \eta_2$$

For instance we have that $?Int.end \leq ?Real.end$ since every descriptor that is dual of $?Int.end$ is also dual of $?Real.end$. Similarly $\text{begin}.?Int.end <: \text{begin}.?Real.end$ since if a process that uses a channel of type $\text{begin}.?Real.end$ is well typed, then the process obtained by replacing this channel for a different one of type $\text{begin}.?Int.end$ is well typed as well: it will receive an integer number where a real number is expected.

2.4 Coinductive characterizations

The subsessioning relation defined in terms of duality embeds the notion of safe substitutability because of its very definition, but it gives little insight on the properties enjoyed by \leq . This is a common problem of every semantically defined preorder relation based on *tests*, such as the well-known testing preorders [6] (the set of duals of a descriptor can be assimilated to the set of its successful tests). In order to gain some intuition over \leq and to obtain a useful tool that will help us studying its properties we will now provide an alternative coinductive characterization. Before doing so, we need to characterize first the class of descriptors that admit at least one dual descriptor. Recall that η is *viable* if there exists η' such that $\eta \times \eta'$. Any non-viable descriptor is the least element of \leq , which henceforward will be denoted by \perp .

DEFINITION 2.8 (Coinductive Viability). η^\times is the largest predicate over descriptors such that either

1. $\eta \downarrow$ and $\eta \downarrow \mu$ implies $\eta \langle \mu \rangle^\times$ for every μ , or

2. there exists μ such that $\eta \downarrow \mu$ and $\eta \langle \mu \rangle^\times$.

The definition provides us with a correct and complete characterization of viable descriptors, as stated in the next proposition.

PROPOSITION 2.9. η^\times if and only if η is viable.

We can now read the statement of Definition 2.8 in the light of the result of the above proposition: Definition 2.8 explains that a descriptor is viable if either (1) it emits an output action regardless of its internal state and every successor after every possible output action is viable too or (2) it guarantees at least one input action such that the corresponding successor is viable too.

DEFINITION 2.10 (Coinductive Subsession). $\eta \leq \eta'$ is the largest relation between session descriptors such that η^\times implies η'^\times and

1. $\eta' \not\Downarrow$ and $\eta' \downarrow \mu$ imply $\eta \downarrow \mu$ with $\eta \langle \mu \rangle \leq \eta' \langle \mu \rangle$, and
2. $\eta \downarrow \mu$ and $\eta \langle \mu \rangle^\times$ imply $\eta' \downarrow \mu$ with $\eta \langle \mu \rangle \leq \eta' \langle \mu \rangle$, and
3. $\eta \downarrow$ and $\eta' \downarrow$ imply $\eta \downarrow \checkmark$ and $\eta' \downarrow \checkmark$.

The definition states that any viable descriptor η may be a sub-session of η' only if η' is also viable. This is obvious since we want the duals of η to be duals of η' as well. Furthermore, condition (1) requires that any output action emitted by the larger descriptor must also be emitted by the smaller descriptor, and the respective continuations must be similarly related. This can be explained by noticing that a descriptor dual of η in principle will be able to properly handle only the outputs emitted by η ; thus in order to be also dual of η' it must also cope with η' outputs, which must thus be included in those of η , hence the condition. The requirement $\eta' \not\Downarrow$ makes sure that η' really emits some output actions. Without this condition we would have $?Int.end \not\leq ?Int.end + end$ as the descriptor on the r.h.s. emits \checkmark which is not emitted by the l.h.s. However, it is trivial to see that $?Int.end \leq ?Int.end + end$. Condition (2) requires that any input action guaranteed by the smaller descriptor must also be guaranteed by the larger descriptor. Again this can be explained by noticing that a descriptor dual of η may rely on the capability of η of receiving a particular value/descriptor in order to continue the interaction without error. Hence, any guarantee provided by the smaller descriptor η must be present in the larger descriptor η' as well. The additional condition $\eta \langle \mu \rangle^\times$ considers only guaranteed input actions that have a viable dual, for a guaranteed input action with a non-viable dual is practically useless. Without such condition we would have, for instance, that $?Int.!0.end + ?Bool.end \not\leq ?Bool.end$, because the descriptor on the l.h.s. guarantees the action $?3$ which is not guaranteed by the descriptor of the r.h.s. of $\not\leq$. It is clear however that in this case the subsessioning relation must hold since the l.h.s. and r.h.s. have the same set of duals. Finally, condition (3) captures the special case in which a descriptor emitting output actions ($\eta \downarrow$) is smaller than a descriptor guaranteeing input actions ($\eta' \downarrow$). This occurs only when η may internally decide to terminate ($\eta \downarrow \checkmark$) and η' guarantees termination ($\eta' \downarrow \checkmark$). In this case, every dual of η must be ready to terminate and to receive any output action emitted by η , hence it will also be dual of η' which guarantees termination but does not emit any output action.

We end this subsection by stating that the coinductive and the semantic definitions of subsessioning coincide, so from now on we will use \leq to denote both.

THEOREM 2.11. $\eta_1 \leq \eta_2 \iff \eta_1 \leq \eta_2$.

2.5 Properties of the subsession relation

Table 3 shows some relevant rules regarding \leq (we omitted rules (E1–E3) and (I1–I3) being the usual idempotency, commutativity, and associativity laws of $+$ and \oplus). Aside from providing further insight on the properties of \leq , these rules are also used in the following for proving the existence of the normal forms for session

descriptors and the correctness of the algorithms. In the table we write \emptyset to denote either \emptyset (the empty type) or \perp (the least sieve) according to the context.

(E4)	$\eta + (\eta' \oplus \eta'')$	$=$	$(\eta + \eta') \oplus (\eta + \eta'')$	
(E5)	$\alpha.\eta + \alpha.\eta'$	$=$	$\alpha.(\eta \oplus \eta')$	
(E6)	$?t.\eta + ?s.\eta$	$=$	$?(t \vee s).\eta$	
(E7)	$?x.\eta + ?x'.\eta$	$=$	$?(\chi \vee \chi').\eta$	
(E8)	$\eta + \perp$	$=$	η	
(14)	$\eta \oplus (\eta' + \eta'')$	$=$	$(\eta \oplus \eta') + (\eta \oplus \eta'')$	
(15)	$\alpha.\eta \oplus \alpha.\eta'$	$=$	$\alpha.(\eta \oplus \eta')$	
(16)	$?t.\eta \oplus ?s.\eta'$	$=$	$?(t \wedge s).(\eta \oplus \eta')$	
(17)	$?x.\eta \oplus ?x'.\eta'$	$=$	$?(\chi \wedge \chi').(\eta \oplus \eta')$	
(18)	$\eta \oplus \perp$	$=$	\perp	
(B1)	$? \psi.\eta$	$=$	\perp	$(\psi = \emptyset)$
(B2)	$! \psi.\eta$	$=$	\perp	$(\psi = \emptyset)$
(B3)	$?t.\eta \oplus ?x.\eta'$	$=$	\perp	
(B4)	$? \psi.\eta \oplus ! \psi'.\eta'$	$=$	\perp	
(B5)	$? \psi.\eta \oplus \text{end}$	$=$	\perp	
(O1)	$! \psi.\eta + \text{end}$	$=$	$! \psi.\eta$	$(\psi \neq \emptyset)$
(O2)	$! \psi.\eta + ? \psi'.\eta'$	$=$	$! \psi.\eta$	$(\psi \neq \emptyset)$
(O3)	$! \psi.\eta + ! \psi'.\eta'$	$=$	$! \psi.\eta \oplus ! \psi'.\eta'$	$(\psi, \psi' \neq \emptyset)$
(O4)	$!t.\eta \oplus !s.\eta$	$=$	$!(t \vee s).\eta$	$(t, s \neq \emptyset)$
(O5)	$!x.\eta \oplus !x'.\eta$	$=$	$!(\chi \vee \chi').\eta$	$(\chi, \chi' \neq \perp)$
(S1)	$?t.\eta$	\leq	$?(t \vee s).\eta$	
(S2)	$?x.\eta$	\leq	$?(\chi \vee \chi').\eta$	
(S3)	$!(t \vee s).\eta$	\leq	$!t.\eta$	$(t \neq \emptyset)$
(S4)	$!(\chi \vee \chi').\eta$	\leq	$!x.\eta$	$(\chi \neq \perp)$

Table 3. Remarkable equalities and inequalities.

Rules (E1–E8) state the fundamental properties of the external choice operator. Rule (E5) shows that an external choice may actually hide an internal choice if it combines descriptors having a common prefix. This is a well-known axiom in the testing theories [6] and it also shows that the external choice does not coincide with the set-theoretic union operator (the internal choice, on the other hand, does coincide with the set-theoretic intersection). Rule (E6) shows the interaction between input actions (over types) and the external choice operator: the value received from the channel is chosen externally, it cannot be negotiated by the receiver. Rule (E7) is similar to rule (E6), except that it deals with sieves. Rule (E8) states that \perp is the neutral element of the external choice.

Rules (11–19) state the fundamental properties of the internal choice operator. Rule (15) is the distributivity law of the prefix operator over the internal choice (the same law does not hold for the external choice operator). Rule (16) shows the interaction between input actions over types and the internal choice operator. A dual of the descriptor on the l.h.s. of $=$ does not know whether the descriptor is ready to receive a value of type t or of type s . Thus, the only possibility is to send a value that has both types. As a consequence, if t and s are disjoint types, namely if $t \wedge s = \emptyset$, then both descriptors are \perp (see rule (B1) below). Rule (17) is similar to rule (16), except that it deals with sieves. Rule (18) states that \perp is the absorbing element of the internal choice.

Rules (B1–B5) characterize non-viable descriptors, namely those descriptors that have no dual. Rules (B1–B2) deal with communications of values from empty types and delegations of sessions with non-viable descriptors. Since these descriptors are completely inert (they do not emit any visible action), they are comparable to the canonical non-viable descriptor \perp . Rule (B3) states the disjunction between values and descriptors: no value is a descriptor,

and no descriptor is a value. Rule (B4) states the directionality of our communication model. In order to be viable, a descriptor cannot simultaneously allow both input and output actions. The only exception to this rule is when the output actions are offered in an external choice, see rules (O1–O3) below. Rule (B5) is similar to rule (B4), except that it deals with end and input actions.

Rules (O1–O5) characterize the peculiar properties of output actions. In every rule the side condition ensures that the output action is not inert (see rule (B2) above). Rules (O1–O2) state that an output action composed in external choice with a end or an input action preempts the alternative action. Rule (O3) states that external and internal choices of output actions are indistinguishable, since these actions are irrevocable. Rule (O4) shows the interaction between output actions over types and the internal choice operator: the value sent over the channel is decided internally by the sender, it will not be negotiated with the receiver. Rule (O5) is similar to rule (O4), except that it deals with sieves.

Rules (S1–S2) show the standard covariant property of inputs: the duals of a session that is capable of receiving values of type t will also be duals of a session that is capable of receiving more values. Rule (S2) is similar to rule (S1) except that it deals with input of descriptors and it can be explained in the same way using the intuition that sieves stand for the set of their duals. Rules (S3–S4) complement rules (S1–S2) with dual properties for output actions, where we have contravariance. Note that in both cases we need one extra hypothesis, namely that $t \neq \emptyset$ and $\chi \neq \perp$. This guarantees that the larger descriptor will actually output some value/descriptor whenever the smaller one does so.

We conclude this section with two remarks. First of all, the rules of Table 3 allow us to derive the following decomposition laws:

$$\begin{aligned} ?t.\eta + ?s.\eta' &= ?(t \setminus s).\eta + ?(s \setminus t).\eta' + ?(t \wedge s).(\eta \oplus \eta') \\ !t.\eta \oplus !s.\eta' &= !(t \setminus s).\eta \oplus !(s \setminus t).\eta' \oplus !(t \wedge s).(\eta \oplus \eta') \end{aligned}$$

the latter rule holding when none of the sets $t \setminus s$, $s \setminus t$, and $t \wedge s$ is empty. Similar rules can be derived for inputs and outputs of sieves, as opposed to types. These rules play a fundamental role in all the algorithms that will follow because they allow us to eventually rewrite external and internal sums so that every summand of the sum begins with a prefix that is disjoint from (emits labels that are not emitted by) the prefix of any other summand.

The second remark concerns the interaction of \leq with the operators of session descriptors. It is easy to see that \leq is preserved by the prefix and the internal choice operators. In the latter case, this follows from the fact that \oplus coincides with the intersection operator in the set-theoretic interpretation of session descriptors. However, as we have already seen while discussing rule (E5), $+$ does not correspond to a Boolean operation and this ultimately makes $+$ quite subtle, as \leq is not respected by $+$ in general. For example, by rule (S1) we have $?Int.end \leq ?Real.end$ however $?Int.end + ?\sqrt{2}.!3.end \not\leq ?Real.end + ?\sqrt{2}.!3.end$. The reason is that in widening $?Int.end$ to $?Real.end$ we create an interference with the term $?\sqrt{2}.!3.end$ because of the guaranteed action $?\sqrt{2}$. Such interferences are not avoided even when we operate with $=$ (as opposed to \leq). For instance, according to rule (16) we have $?(Int \vee \sqrt{2}).end \oplus ?Int.!3.end = ?Int.(end \oplus !3.end)$, but $?(Int \vee \sqrt{2}).end \oplus ?Int.!3.end + ?\sqrt{2}.!4.end \neq ?Int.(end \oplus !3.end) + ?\sqrt{2}.!4.end$. Here the action $?\sqrt{2}$ is not guaranteed by $?(Int \vee \sqrt{2}).end \oplus ?Int.!3.end$ and (16) tells us that in practice the capability of $?(Int \vee \sqrt{2}).end$ of receiving $\sqrt{2}$ is useless. However, removing this capability may also remove interferences in the context of an external choice, making (16) unsafe in general. Finally, rule (B4) must be used with care within an external choice because its output capability makes the descriptor on the l.h.s. of $=$ to be observable (it may autonomously

emit an action), whereas \perp is totally inert. For instance we have $?Int.end \oplus !Int.!0.end = \perp$ and $\perp + ?Bool.!3.end = ?Bool.!3.end$, but $(?Int.end \oplus !Int.!0.end) + ?Bool.!3.end = (?Int.end + ?Bool.!3.end) \oplus (!Int.!0.end + ?Bool.!3.end) = \perp$. Rule (B5) suffers from a similar problem, which is slightly less severe because end denotes a terminated descriptor.

2.6 Algorithms

In order to use our type system we must be able to decide the relations we introduced in the previous sections, namely subsieving (and subsessioning), subtyping, and duality.

Subsieving. Let us start to show how to decide that a sieve is smaller than another. Since Boolean combinators have a set-theoretic interpretation we can apply exactly the same reasoning we did for types in Section 2.3. Namely, deciding $\chi \leq \chi'$ is equivalent to deciding $\chi \wedge \neg\chi' \leq \perp$. The l.h.s. can be rewritten in disjunctive normal form whose definition for sieves is (we convene that $\bigvee_{i \in \emptyset} \chi_i = \sum_{i \in \emptyset} \eta_i = \perp$):

DEFINITION 2.12 (Disjunctive normal form). *A sieve is in disjunctive normal form if it is of the form $\bigvee_{i \in I} \bigwedge_{j \in J} \lambda_{ij}$ where λ_{ij} denote descriptor literals, that is either η or $\neg\eta$.*

Next, we can check emptiness of each element of the union separately, reducing the problem to checking the following relation: $\bigwedge_{i \in I} \eta_i \leq \bigvee_{j \in J} \eta_j$. Since this is equivalent to $\bigoplus_{i \in I} \eta_i \leq \bigvee_{j \in J} \eta_j$, we can apply the strong disjunction property (Proposition 2.7) we stated for descriptors and obtain

$$\bigwedge_{i \in I} \eta_i \leq \bigvee_{j \in J} \eta_j \iff \exists j \in J : \bigoplus_{i \in I} \eta_i \leq \eta_j$$

which is precisely the same problem that has to be solved in order to decide the subtyping relation (cf. equation (11)). In conclusion, in order to decide both subsieving and subtyping it suffices to decide subsessioning.

Subsessioning. To decide whether two descriptors are in subsessioning relation we define a normal form for descriptors and, more generally, sieves (the latter occurring in the prefixes of the former).

DEFINITION 2.13 (Strong normal form). *A sieve χ in disjunctive normal form is in strong normal form if*

1. if $\chi \equiv \bigvee_{i \in I} \bigwedge_{j \in J} \lambda_{ij}$, then for all $i \in I, j \in J$, λ_{ij} is in strong normal form and $\bigwedge_{j \in J} \lambda_{ij} \neq \perp$ for all $i \in I$;
2. if $\chi \equiv \neg\eta$, then η is in strong normal form;
3. otherwise χ is either of the form $\bigoplus_{i \in I} !\psi_i.\eta_i\{\oplus end\}$ or $\sum_{i \in I} ?\psi_i.\eta_i\{+ end\}$, where for all $i \in I$, $\psi_i \neq \emptyset$, ψ_i and η_i are in strong normal form and for all $i, j \in I$, $i \neq j$ implies $\psi_i \wedge \psi_j = \emptyset$, and end is possibly missing.

Transformation in strong normal form is effective:

THEOREM 2.14 (Normalization). *For every sieve χ it is possible effectively to construct χ' in strong normal form such that $\chi = \chi'$.*

Finally, to check that two descriptors are in relation we rewrite both of them in strong normal form, check that neither is \perp , and then apply the algorithm whose core rules are given in Table 4.

Rule (MIX-CHOICES) states that an internal choice is smaller than an external one if and only if they both have an end summand. Rule (EXT-CHOICES) states that it is safe to widen external choices whereas rule (INT-CHOICES) states that it is safe to narrow internal ones. Both rules are used in conjunction with (PREFIX), which states covariance over descriptor continuations. Note that rule (PREFIX) relates two descriptors only if they have the same prefix. Therefore before applying (EXT-CHOICES) and (INT-CHOICES) we have to transform the descriptors so that prefixes

(END) $\frac{}{end \leq end}$	(PREFIX) $\frac{\eta \leq \eta'}{\alpha.\eta \leq \alpha.\eta'}$	(MIX-CHOICES) $\frac{}{\bigoplus_{i \in I} \eta_i \oplus end \leq \sum_{j \in J} \eta'_j + end}$
(EXT-CHOICES) $\frac{I \subseteq J \quad \eta_i \leq \eta'_i \ (\forall i \in I)}{\sum_{i \in I} \eta_i \leq \sum_{j \in J} \eta'_j}$	(INT-CHOICES) $\frac{J \subseteq I \quad \eta_j \leq \eta'_j \ (\forall j \in J)}{\bigoplus_{i \in I} \eta_i \leq \bigoplus_{j \in J} \eta'_j}$	

Table 4. Algorithmic subsessioning structural rules.

on the two sides that have a non-empty intersection are rewritten in several summands so as to find the same prefix on both sides: this is done by repeated applications of the decomposition laws described in Section 2.5. The corresponding algorithmic rules can be found in the long version of the paper.

THEOREM 2.15 (Soundness and Completeness). *The algorithm is sound and complete with respect to \leq and it terminates.*

Duality. As regards duality, we show how to effectively construct the canonical dual of a descriptor η , which is defined as the least descriptor in the set-theoretic interpretation of η . Then checking duality of two descriptors reduces to computing the canonical dual of one of the two and then check subsessioning.

Constructing the canonical dual of a descriptor η is straightforward once η is in strong normal form (which can be effectively done by Theorem 2.14). Then it just suffices to change every $?$ into $!$, every $+$ into \oplus and viceversa and coinductively apply this transformation to the continuations leaving end descriptors unchanged. Regularity ensures that the coinductive transformation terminates (by using memoization techniques). Showing that the obtained session descriptor is the canonical dual of η is a trivial exercise.

3. Process language

In this section we show how to use our theory to type a process calculus so that well-typed processes satisfy the progress property.

3.1 Syntax

The main design criterion for our process calculus is minimality: we define the smallest calculus that allows us to use all the characteristics of our session types. Thus we consider a calculus, whose syntax is given in Table 5, in which processes are single threads that can just emit (a slight generalisation of) the five signals we described in the Introduction or be composed by internal and external choices. Session conversations take place at the upper level of systems where processes run in parallel.

(prefixes)	π	::=	$u!(e) \mid u?(x : t) \mid u!(u) \mid u?(z : \chi) \mid \text{connect } a(z)$
(processes)	P, Q	::=	$\mathbf{0} \mid \pi.P \mid P \oplus P \mid P + P$
(channels)	u	::=	$z \mid \bar{k} \mid \tilde{k}$
(sessions)	a	::=	$x \mid c^{\text{begin}, \eta}$
(expressions)	e	::=	$a \mid \dots$
(systems)	\mathbb{S}, \mathbb{T}	::=	$P \mid \mathbb{S} \parallel \mathbb{S}$

Table 5. Syntax of processes and systems

Processes $u!(e).P$ and $u?(x : t).P$ are communication processes. They are used to communicate either values of the host language or (yet-to-be-used) session channels (both returned by the

R-CONNECT $\frac{}{\text{connect } c^{\text{begin.}\eta}(z).P \xrightarrow{c(z:\eta)} P}$	R-SEND $\frac{e \downarrow v}{k!(e).P \xrightarrow{k!(v)} P}$	R-RECEIVE $\frac{}{k?(x:t).P \xrightarrow{k?(x:t)} P}$	R-SENDS $\frac{}{k!(k_1).P \xrightarrow{k!(k_1)} P}$	R-RECEIVES $\frac{}{k?(z:\chi).P \xrightarrow{k?(z:\chi)} P}$
R-EXTCH1 $\frac{P \xrightarrow{\ell} P' \quad \ell \neq \tau}{P + Q \xrightarrow{\ell} P'}$	R-EXTCH2 $\frac{P \xrightarrow{\tau} P'}{P + Q \xrightarrow{\tau} P' + Q}$	R-EXTCH3 $\frac{P \xrightarrow{k!(v)} P}{P + Q \xrightarrow{\tau} P}$	R-EXTCH4 $\frac{P \xrightarrow{k!(k_1)} P}{P + Q \xrightarrow{\tau} P}$	R-INTCH $\frac{}{P \oplus Q \xrightarrow{\tau} P}$

Table 6. Process reduction rules.

expression e). The latter are either variables—ranged over by x —or (session type) values. Session values are names (actually, channel names) tagged by a session type (as explained in Section 2.2.1 we use $c^{\text{begin.}\eta}$ rather than the less restrictive c^t to ensure the strong disjunction property (5) for union types). Processes $u!(|u|).P$ and $u?(|z:\chi|).P$ are *delegation processes* by which a conversation already started on the channel object of these actions is respectively delegated and resumed.

Both communication and delegation take place on *channels*, ranged over by u . These are either channel variables ranged over by z , or *internal channels*, denoted by k or \tilde{k} . We say that k and \tilde{k} are *dual* and we state that channel duality is an involution, i.e., $\tilde{\tilde{k}} = k$. Dual channels represent the two end-points of a session and they are greyed to stress that such channels occur only at runtime (they cannot be written by the programmer). The use of two end-point channels is a technique that we borrow from [13] where they are called polarized channels.

Besides the idle process denoted by $\mathbf{0}$, there is $\text{connect } a(z).P$, the *connect process* that connects on the name of a and starts a conversation using the local channel z (whose occurrences inside P are bound by $\text{connect } a(z)$) and following the prescriptions of the (session) type of a . Finally, $P + P$ and $P \oplus P$ denote *external* and *internal* choices, respectively.

Since types are recursive so are processes. Therefore the processes of our calculus are possibly infinite regular trees that are generated by the productions in Table 5 and that satisfy the contractivity condition requiring that on every infinite branch there are infinitely many applications of the prefixed process. Contractivity rules out processes of the form, e.g., $P = P \oplus P$ and—as for types—it provides a well-founded order used in the proofs.

All forms of interaction (communication, delegation, and connection) happen between two distinct processes that are composed in parallel in *systems*, ranged over by \mathbb{S}, \mathbb{T} , and they uncoil according to the operational semantics we describe next.

3.2 Operational semantics

According to the reduction semantics described in Table 6, processes may emit one of the following labels

$$\ell ::= \tau \mid k!(v) \mid k?(x:t) \mid k!(|k_1|) \mid k?(|z:\chi|) \mid c(z:\eta)$$

that is, either an invisible signal τ corresponding to an internal choice performed by the process or, roughly, one of the five signals we discussed in the Introduction. Note that the signal for connection $c(z:\eta)$ carries a session descriptor: it indicates that the process is willing to connect on name c and, using a local channel z , have a conversation described by η .

Rule R-CONNECT states that connection can take place only on concrete channels (not variables) and it publishes the name c , the variable z , and the protocol η of the service/conversation offered by the process. Rules for send and receive are pretty straightforward: just notice that all subjects are internal channels (so they correspond to some concrete channels that succeeded a connection: see rule

CONNECTION in Table 7) and only values are communicated (every expression that is object of a communication is first evaluated). Rule R-INTCH states that internal choices correspond to silent moves while the remaining four rules describe the behaviour of external choices. These propagate internal moves (R-EXTCH2), offer visible signals for the interacting partner to make a choice (R-EXTCH1), and state the irrevocability of outputs (R-EXTCH3 and R-EXTCH4) as the process can silently reduce to any subprocess that may perform an output. Symmetric rules are omitted.

These signals may synchronize at the system level giving rise to three different forms of interaction—connection, communication, and delegation—described in Table 7 (two trivial rules for lift-

CONNECTION $\frac{\Sigma \vdash \mathbb{S} \xrightarrow{c(z:\eta)} \Sigma \vdash \mathbb{S}' \quad \Sigma \vdash \mathbb{T} \xrightarrow{c(z':\eta')} \Sigma \vdash \mathbb{T}' \quad \eta \bowtie \eta' \quad k \notin \text{dom}(\Sigma)}{\Sigma \vdash \mathbb{S} \parallel \mathbb{T} \xrightarrow{\tau} \Sigma, k:\eta, \tilde{k}:\eta' \vdash \mathbb{S}'[k/z] \parallel \mathbb{T}'[\tilde{k}/z']}$
COMMUNICATION $\frac{\Sigma \vdash \mathbb{S} \xrightarrow{k!(v)} \Sigma \vdash \mathbb{S}' \quad \Sigma \vdash \mathbb{T} \xrightarrow{\tilde{k}(x:t)} \Sigma \vdash \mathbb{T}' \quad v \in t}{\Sigma, k:\eta, \tilde{k}:\eta' \vdash \mathbb{S} \parallel \mathbb{T} \xrightarrow{\tau} \Sigma, k:\eta!(v), \tilde{k}:\eta'(?v) \vdash \mathbb{S}' \parallel \mathbb{T}'[v/x]}$
DELEGATION $\frac{\Sigma \vdash \mathbb{S} \xrightarrow{k!(k')} \Sigma \vdash \mathbb{S}' \quad \Sigma \vdash \mathbb{T} \xrightarrow{\tilde{k}(z:\chi)} \Sigma \vdash \mathbb{T}' \quad \Sigma(k') \leq \chi}{\Sigma, k:\eta, \tilde{k}:\eta' \vdash \mathbb{S} \parallel \mathbb{T} \xrightarrow{\tau} \Sigma, k:\eta!(\Sigma(k')), \tilde{k}:\eta'(?(\Sigma(k'))) \vdash \mathbb{S}' \parallel \mathbb{T}'[k'/z]}$

Table 7. System reduction rules.

ing labels emitted by processes at the system level are omitted for brevity). The rules show that process selection is very dynamic. Recall that in our framework external choices are resolved at run-time according to the type of the communicated value or the descriptor of the delegated session. We see here that we have a third form of dynamic selection, insofar as a process may start a session on a channel of a given name only with processes that declare that on the same name they are willing to start a dual session. Thus it is possible to have services (materialized by session type values) that have the same name (e.g. `search`) but implement incompatible contracts: they will never synchronize together. More precisely rule CONNECTION checks whether two connection signals on the same name declare dual session descriptors and, if so, it connects them by spawning a fresh ($k \notin \text{dom}(\Sigma)$) internal channel and its dual. The local variables used for the connection are replaced by the internal channel and its dual. The internal channel names and session descriptors are stored in the *session environment* Σ , which contains judgements of the shape $k:\eta$. Since the two interacting partners are given different fresh names k and \tilde{k} we avoid interference in synchronization, as well as confusion in session environments.

Communications and delegations take place on internal channels and only after a connection. In both cases the corresponding rules in Table 7 check whether the input and output signals are

$\frac{}{\Gamma, x : t \vdash x : t}$	$\frac{}{\Gamma \vdash c^{\text{begin}.\eta} : \text{begin}.\eta}$	$\frac{\text{T-SUB}}{\Gamma \vdash e : t' \quad t' <: t}{\Gamma \vdash e : t}$	$\frac{\text{T-WEAK}}{\Gamma \vdash P : \Delta}{\Gamma \vdash P : (\Delta \cdot u : \text{end})}$	$\frac{\text{T-STRENGTH}}{\Gamma \vdash P : (\Delta \cdot u : \text{end})}{\Gamma \vdash P : \Delta}$	$\frac{\text{T-ZERO}}{\Gamma \vdash \mathbf{0} : -}$
$\frac{\text{T-CONNECT}}{\Gamma \vdash P : (\Delta \cdot z : \eta) \quad \Gamma \vdash a : \text{begin}.\eta}{\Gamma \vdash \text{connect } a(z).P : \Delta}$		$\frac{\text{T-RECEIVE}}{\Gamma, x : t \vdash P : (\Delta \cdot u : \eta)}{\Gamma \vdash u?(x : t).P : (\Delta \cdot u : ?t.\eta)}$	$\frac{\text{T-SEND}}{\Gamma \vdash e : t \quad \Gamma \vdash P : (\Delta \cdot u : \eta)}{\Gamma \vdash u!(e).P : (\Delta \cdot u : !t.\eta)}$		
$\frac{\text{T-RECEIVES}}{\Gamma \vdash P : (x : \eta' \cdot u : \eta) \quad \chi \leq \eta'}{\Gamma \vdash u?(z : \chi).P : (u : ?\chi.\eta)}$		$\frac{\text{T-SENDS}}{\Gamma \vdash P : (\Delta \cdot u : \eta) \quad \eta' \leq \chi}{\Gamma \vdash u!(u').P : ((\Delta \cdot u' : \eta') \cdot u : !\chi.\eta)}$			
$\frac{\text{T-INTCH}}{\Gamma \vdash P : (\Delta \cdot u : \eta_1) \quad \Gamma \vdash Q : (\Delta \cdot u : \eta_2)}{\Gamma \vdash P \oplus Q : (\Delta \cdot u : \eta_1 \oplus \eta_2)}$	$\frac{\text{T-EXTCH}}{\Gamma \vdash P : (\Delta \cdot u : \eta_1) \quad \Gamma \vdash Q : (\Delta \cdot u : \eta_2)}{\Gamma \vdash P + Q : (\Delta \cdot u : \eta_1 + \eta_2)}$		$\frac{\text{T-SYS}}{\Gamma \vdash P : \Delta}{\Gamma \Vdash P : \text{set}(\Delta)}$	$\frac{\text{T-PAR}}{\Gamma \Vdash \mathbb{S} : \Lambda_1 \quad \Gamma \Vdash \mathbb{T} : \Lambda_2}{\Gamma \Vdash \mathbb{S} \parallel \mathbb{T} : \Lambda_1 \cup \Lambda_2}$	

Table 8. Typing rules for processes and systems

compatible: in communications, it is checked that the value has the expected type; in delegations, it is checked that the delegated channel implements a session matching the input sieve. If the check is successful, the processes synchronise and the session environment is updated by computing the successors of the session descriptors corresponding to the channels on which interaction has occurred.

Finally note that the dynamic checks in these three rules, more than for soundness, are needed and used to drive the computation, since external choices are dynamically selected by using the type of the communicated value, or the descriptor of the delegated session, or both the name and the session descriptor of a connection request.

3.3 Typing

The original motivation for introducing session types [15, 16] was to ensure that values sent and received in communication protocols were of appropriate types and that the two partners always agreed on how to continue the conversation. A type system ensuring also the progress property, i.e., that a started session cannot get stuck if the required connections are available, was first proposed in [10].

In the present calculus as in [9] the operational semantics itself ensures that there cannot be a type mismatch in communications, since all checks are performed at the moment of the synchronisation. So for example the system $k!(3) \parallel \bar{k}?(x : \text{Bool})$ is stuck. Clearly the above system cannot be generated, since a CONNECTION rule can be executed only when the two session descriptors of the common session channel are dual. We present in this section a type system which prevents also any deadlock due to the interleaving of two or more sessions.

A first simple example of deadlock is given by

$$\text{connect } a^{t_1}(z_1).\text{connect } b^{t_2}(z_2).z_2?(x : \text{Int}).z_1!(3) \\ \parallel \text{connect } a^{t_2}(z_3).\text{connect } b^{t_1}(z_4).z_3?(x : \text{Int}).z_4!(2)$$

where $t_1 \equiv \text{begin}.\text{!Int}.\text{end}$ and $t_2 \equiv \text{begin}.\text{?Int}.\text{end}$. After two executions of the CONNECTION rule, both processes starve waiting for values that are never sent.

More subtle examples of deadlock spring from session delegation, whereby a (sequential) process can receive the dual of a channel it already owns, making synchronization impossible. Consider

$$\text{connect } a^{t_1}(z_1).\text{connect } b^{t_2}(z_2).z_2!(z_1) \parallel \text{connect } a^{t_3}(z_3) \\ \text{connect } b^{t_4}(z_4).z_4?(z : ?\text{Int}.\text{end}).z?(x : \text{Int}).z_3!(2)$$

where $t_1 \equiv \text{begin}.\text{?Int}.\text{end}$, $t_2 \equiv \text{begin}.\text{!}(\text{?Int}.\text{end}).\text{end}$, $t_3 \equiv \text{begin}.\text{!Int}.\text{end}$, $t_4 \equiv \text{begin}.\text{?}(\text{?Int}.\text{end}).\text{end}$. This phenomenon may also jeopardise subject reduction, as discussed in [19].

A typing discipline that avoids both problems outlined above is defined in Table 8. The judgements for processes have the form

$\Gamma \vdash P : \Delta$ where Γ is a *type environment* (a mapping from variables to types) and Δ is a *session stack*. The latter is a mapping from channels to session descriptors to which identifiers for ended sessions can be freely added and removed (rules T-WEAK, T-STRENGTH) and is used to record the session descriptors of the channels used in P . It is organised as a stack (the rightmost element being the top) to keep track of the current session, that is the most recently created one. The stack allows us to avoid the first example of deadlock, by organising sessions as nested critical regions in which a channel cannot be used unless all nested sessions have been consumed (either because they ended or because they were delegated to some other process). Actions are allowed only if their subject is the current session channel, the one on the top of the stack (rules T-CONNECT, T-SEND, T-RECEIVE, T-SENDS, and T-RECEIVES) and they are recorded in the conclusion. Similarly, two processes can be composed only if they share the same active channel (rules T-INTCH, T-EXTCH). We thus improve the proposal of [10]. In addition, the rules for communication check that type constraints are satisfied while sieve constraints are checked by the delegation rules. The second example of deadlock is avoided by requiring that the only other internal channel which can occur in a process accepting a delegation is the channel on which the delegation took place (rule T-RECEIVE).

The typing discipline is lifted to systems by merging all channel assumptions, disregarding the order in which they appear (by the operator set), as stated by the last two rules in Table 8.

The type system satisfies the subject-reduction property. This is used to prove the following progress property.

THEOREM 3.1 (Progress). *A well-typed closed system \mathbb{S} where no internal channel occurs has the progress property, that is $\vdash \mathbb{S} \xrightarrow{\tau} \Sigma \vdash \mathbb{S}' \xrightarrow{\tau} \text{implies that either } \mathbb{S}' \text{ does not contain internal channels or } \mathbb{S}' \xrightarrow{c(z:\eta)}$.*

It is interesting to note that if we disregard the order in the session stack and allow an arbitrary stack in the premise of rule T-RECEIVE, then we get a typing discipline which enjoys subject reduction (thus it ensures sound interactions) but no longer guarantees progress.

4. Conclusion

In this work we have defined a clean, simple, and general theory of session types whose features are semantically justified. In order to achieve this goal, we had to subvert the usual session type presentations, where the subtyping (and subsession) relations are introduced first, and then it is shown that they are sound. Here we have focused on duality as the main characterizing feature, and then

we have semantically defined subtyping and subsessioning in terms of duality. This is exactly the *modus operandi* adopted in the testing theories for behavioural equivalences of processes [6, 7]. In our case the passing of a test is characterized by the duality relation, and two session descriptors are equivalent if they pass the same tests, namely if the corresponding sets of duals are the same.

We believe that the type system satisfies our initial requirements of simplicity and compositionality: session descriptors boil down to two kinds of prefixed terms whose actions may be specified by Boolean operators and two kinds of choices that can be freely intermixed. There is an apparent redundancy in the language of session descriptors that is due to the presence of two different operators, the internal sum and the conjunction between sieves, whose set-theoretic interpretations coincide. However, the two operators play rather different, but equally essential, roles: the internal choice is a *behavioural* operator, whereas sieve conjunction is a *pattern* operator. We claim that any attempt to unify these two operators, by merging session descriptors and sieves together, disrupts the whole theory. The reason is that the notion of duality, which is intrinsically a behavioural one, does not match well with (most) Boolean combinators, especially with disjunction (which does *not* coincide with the external choice $+$, as we have seen) and negation (defining the transition system of a negated session descriptor is puzzling at least). On the other hand, it is easy to see that sieves and their Boolean combinators play a fundamental role in the definition of the strong normal form, which in turn is pivotal to the definition of all the algorithms presented in Section 2.6. Overall the separation of behaviours and patterns, which looks embarrassingly natural in the end (we have been stuck on this point for long), is what allows us to provide semantic characterizations of all the notions in the formalism. The alternatives to sieves that we considered along the way all resulted in either syntactical restrictions (like those in current session types, where different branches must be tagged by means of disjoint labels) or incompleteness results (the inability of proving subsessioning for some categories of session descriptors).

There is a whole spectrum of possibilities regarding the *communication model* that may underlie the theory. In this work we have focused on output irrevocability because it closely corresponds to the communication model adopted by distributed systems in practice and also because it allows us to draw a closer comparison with other works on session types, where at every stage of an interaction only one partner has the floor and no handshaking ever occurs. A solution at the opposite side of the spectrum involves a handshaking phase before every interaction, to determine a message that the receiver is capable to handle and that the sender is able to produce, which is the model adopted in [4]. As expected, as the communication model gains expressive power, the subsession (subcontract in [4]) relation gets stricter. For example, while our subsessioning relation naturally accounts for width subtyping (the extension of external choices with additional, but unrelated branches), the subcontract relation in [4] does so only by means of explicit coercions. There is also an interesting compromise between these two solutions, consisting in removing rules (TR5) and (TR6) from the transition relation of session descriptors and leaving the rest unchanged. In this case the two communicating partners may agree on who is entitled to send and who is entitled to receive a value, although it is still not possible to negotiate the specific value exchanged during the communication. The impact of this change to the theory is not trivial (an indication comes from the fact that the normal form changes radically, since it now makes sense to mix input and output actions in the same descriptor), and we plan to investigate it in some future work.

The core of our work is the theory of session types. Our claim of simplicity does not contrast with the relative complexity of the typing discipline in Section 3.3, which nonetheless improves on

similar proposals [10]. When trying to define a typing discipline that statically enforces the progress property, session types suffer from the natural limitation of typing channels rather than whole processes. In fact, despite our judgements have the form $\Gamma \vdash P : \Delta$, the Δ is not a type for P but just a mapping from channels to session descriptors which prescribes the use of channels inside P . By defining a typing discipline that describes the behavioural type of processes rather than that of each channel used by processes we could better catch the mutual dependencies of interactions on different channels and thus ensure progress without imposing all the present restrictions (a first step in this direction is done in [8]). Of course this would correspond also to a paradigm shift, where we focus on the orchestration of different components rather than on single conversations. The application of the semantic framework we have designed to this scenario looks like a challenging, yet interesting direction for future research.

References

- [1] L. Caires and H. T. Vieira. Typing conversation in the conversation calculus. Technical report, CITI, 2008.
- [2] G. Castagna, R. De Nicola, and D. Varacca. Semantic subtyping for the π -calculus. *Theor. Comput. Sci.*, 398(1-3):217–242, 2008.
- [3] G. Castagna and A. Frisch. A gentle introduction to semantic subtyping. In *PPDP '05*, 2005. Joint ICALP-PPDP keynote talk.
- [4] G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. Extended version of the article in the Proc. of *POPL '08*, submitted, available on the authors' web pages, 2008.
- [5] I. Castellani and M. Hennessy. Testing theories for asynchronous languages. In *FST&TCS '98*, LNCS 1530. Springer, 1998.
- [6] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theor. Comput. Sci.*, 34:83–133, 1984.
- [7] R. De Nicola and M. Hennessy. CCS without τ 's. In *TAPSOFT/CAAP'87*, LNCS 249. Springer, 1987.
- [8] M. Dezani-Ciancaglini, U. de' Liguoro, and N. Yoshida. On progress for structured communications. In *TGC'07*, LNCS 4912, 2008.
- [9] M. Dezani-Ciancaglini, S. Drossopoulou, E. Giachino, and N. Yoshida. Bounded session types for object-oriented languages. In *FMCO'06*, LNCS 4709. Springer, 2007.
- [10] M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, and S. Drossopoulou. A distributed object oriented language with session types. In *TGC'05*, LNCS 3705. Springer, 2005.
- [11] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *The Journal of ACM*, 2008. To appear.
- [12] S. Gay. Bounded polymorphism in session types. *MSCS*, 2008. To appear.
- [13] S. Gay and M. Hole. Subtyping for session types in the pi-calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
- [14] M. Hennessy and A. Ingólfssdóttir. A theory of communicating processes with value-passing. In *ICALP'90*, LNCS 443. Springer, 1990.
- [15] K. Honda. Types for dyadic interaction. In *CONCUR'93*, LNCS 715. Springer, 1993.
- [16] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP'98*, LNCS 1381. Springer, 1998.
- [17] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004.
- [18] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theor. Comput. Sci.*, 114(1):149–171, 1993.
- [19] N. Yoshida and V. T. Vasconcelos. Language primitives and type disciplines for structured communication-based programming revisited. In *SecRet'06*, ENTCS 171, 2007.