Foundation of Session Types (extended abstract)

Giuseppe Castagna[1] Mariangiola Dezani-Ciancaglini[2] Elena Giachino[1,2] Luca Padovani[3]

[1]PPS (CNRS) - Université Denis Diderot - Paris, France
[2]Dipartimento di Informatica - Università degli Studi di Torino - Torino, Italy
[3]Istituto 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 show that all monomorphic dyadic session types proposed in the literature are particular cases of our session types.*

1 Introduction

Sessions are a common and widespread mechanism of interaction in distributed architectures. Two processes willing to interact establish a connection on a shared public channel. In this connection they agree on some private channel on which to have a conversation, dubbed session. The conversation follows a given protocol which describes the kind and order of the messages exchanged on the private channel. In general, a protocol does not specify a unique sequence of interactions. At any point of the interaction the rest of the conversation for a process may depend upon the kind of messages received by the process on the private channel and/or the internal state of the process. When the decision is exclusively based on the received messages one speaks of an *external choice*. When the decision is taken autonomously by the process one speaks of an *internal choice*. The messages exchanged during a session may be synchronization signals, basic values (e.g., integers, booleans, strings), names of public channels (those used to start sessions), or even names of private channels of already started sessions. In the last case one speaks of *delegation* since by sending to some other process the private channel of a session, the process delegates the receiver to continue that session.

Static descriptions of the behavior of sessions (i.e., their protocol) should permit the detection of communication mismatches and session deadlocks, ensuring successful termination of every session. Types are a good candidate for such a description, except that typical type systems for process algebrae are unfit to type the private channels on which sessions take place, since these channels can carry messages of different types. To obviate this limitation Honda *et al.* introduced *session types* [19, 20] that describe the sequences of messages exchanged on a private session channel and their possible branching based on labels. To that end they enrich the language of types *and of processes* with specialized signals for connections, for delegation, and signals carrying labels that drive choices in combination with label-based branching primitives. Since then, several variants of session types have been put forward (see Section 3). They vary according to the programming language they target, the type containment relations and the specific features they aim to capture. As Honda *et al.*, they rely on label-based primitives that tie them to the particular problem they tackle and may hinder their adoption in general purpose languages.

In this work we present a basic and unified foundation of session types that aims at being as much language independent as possible. To achieve language independence, we design our types around the standard π-calculus: session connections, interactions, and delegations will be imagined as instances of π-calculus communications. We suppose branching as being implemented by classic process algebra internal and external choices [11], with just a single modification: we allow the branch of an external choice to be selected according to the type of the message being communicated, as opposed to the channel on which communication occurs. This modification fits nicely the session-based communication model, where messages are exchanged over a unique, private channel.

Our approach has many positive upshots. First, all monomorphic, dyadic session types proposed in the literature are particular instances of the session types discussed here. Second, having dissociated control from a particular linguistic construct, that is label-driven branching, we can more easily type the native branching constructs of a language we want to endow with session types, thus avoiding clumsy language extensions. As an aside, the language independence is further increased by the fact that all our definitions are semantic-based, rather than syntax-oriented. Third, we enhance compositionality of branching constructs because the result of the combination of different branches is automatically computed at type level, without the need of introducing new labels or of renaming existing ones to avoid clashes on shared labels. Last but not least, replacing labels with values and types increases expressiveness: values are first class (so they can result from computations and communicated on channels) and types enable the definition of finer grained disciplines for branch selection.

The rest of the paper is organized as follows. Section 2 defines syntax and set-theoretic semantics of our session

types. The subtyping and subsessioning relations that follow arise as natural consequences of our semantic-based framework. We provide a coinductive characterization of subsessioning that sheds light on the properties of subsessioning and finally we describe algorithms to decide all key relations defined in the article. Section 3 provides a more technical discussion about how our approach subsumes and improves existing session types proposals, and it shows an example of session that can be defined by a generic π-calculus process typable in our formalism. Section 4 summarizes the contributions of our work, draws connections with some of the most closely related papers, and sketches future directions of research. Proofs, details, and further technical content can be found in the full version available on the net [5].

2 Session types

2.1 Type syntax

As said in the introduction we have two kinds of channels: public ones that are used to connect and establish a private channel for the conversation, and these private ones. At type level this distinction corresponds to two different syntactic categories. Public channels are associated with a *session type* of the form $\mathsf{begin}.\eta$. This type classifies channels ready to initiate a conversation on some private channel that will follow the description η. Thus, private channels are classified by *session descriptors*, ranged over by η. Session descriptors and types are defined by the grammar:

$$\begin{array}{lrcl}
\text{(types)} & \mathsf{t} & ::= & \cdots \mid \mathsf{begin}.\eta \mid \neg\mathsf{t} \mid \mathsf{t} \wedge \mathsf{t} \mid \mathsf{t} \vee \mathsf{t} \mid \mathsf{v} \\
\text{(descriptors)} & \eta & ::= & \mathsf{end} \mid \alpha.\eta \mid \eta \oplus \eta \mid \eta + \eta \\
\text{(actions)} & \alpha & ::= & !\mathsf{t} \mid ?\mathsf{t} \mid !\chi \mid ?\chi \\
\text{(sieves)} & \chi & ::= & \eta \mid \neg\chi \mid \chi \wedge \chi \mid \chi \vee \chi
\end{array}$$

Participants of a session use their (private) session channel either to exchange values (of some type) or to delegate other session channels (of some descriptor). In descriptors we use $?\mathsf{t}$ and $!\mathsf{t}$ 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 (i.e., catch) and send (i.e., delegate) some channel which already started a conversation and will continue it according to the behavior 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) the 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) the channel gives 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 to enforce precedence. For instance, $(!\mathsf{t}.\eta) + \mathsf{end}$ and $!\mathsf{t}.\eta + \mathsf{end}$ denote the same session descriptor, which is different from $!\mathsf{t}.(\eta + \mathsf{end})$. Types t are inherited from the host language (this is stressed in the grammar above by the ellipsis in the production for types), to which we add (unless they are already provided by the host) 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 $\mathsf{begin}.\eta$ which classify yet-to-be-used public 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 the names of public channels are communicated over other channels; session types will also extend the type system of the host language which can thus use names of public channels as first class values.

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

$$?\mathsf{Real}.!\mathsf{Int}.\mathsf{end} + ?\mathsf{Int}.!\mathsf{Bool}.\mathsf{end} \tag{1}$$

The session descriptor above declares that if a process (that uses a channel with that behavior) 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 message drives the selection of the external choice). That is, its conversation will be represented by the following descriptor ($\mathsf{t} \setminus \mathsf{s}$ stands for $\mathsf{t} \wedge \neg\mathsf{s}$):

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

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

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

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

$$?\mathsf{Real}.?\mathsf{Int}.\mathsf{end} + ?\mathsf{Int}.?\mathsf{Bool}.\mathsf{end} \tag{4}$$

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: $!(\mathsf{Real} \setminus \mathsf{Int}).!\mathsf{Int}.\mathsf{end}$. 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 $?(\mathsf{Real} \setminus \mathsf{Int}).?\mathsf{Int}.\mathsf{end} + ?\mathsf{Int}.?(\mathsf{Bool} \wedge \mathsf{Int}).\mathsf{end}$ which is equivalent to $?(\mathsf{Real} \setminus \mathsf{Int}).?\mathsf{Int}.\mathsf{end}$ 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 behavior is the solution of the equation) $\eta = \mathsf{end} + ?\mathsf{Int}.\eta$ 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 [15, 7] 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.

Definition 2.1 (Types) *The types of our system are the possibly infinite regular trees coinductively generated by the productions in the grammar at the beginning of this section 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) $\mathsf{t} = \mathsf{t} \vee \mathsf{t}$ 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 $\eta = ?\eta.\mathsf{end}$; this restriction generalizes the typing technique used in all works on (recursive) session types that forbids delegation of a channel over itself [20, 27] (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 we use in proofs.

We do not specify any particular property for the types of the host language. 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 which must satisfy the following *strong disjunction property* for unions:

$$\vdash \mathsf{v} : \mathsf{t}_1 \vee \mathsf{t}_2 \quad \Longleftrightarrow \quad \vdash \mathsf{v} : \mathsf{t}_1 \text{ or } \vdash \mathsf{v} : \mathsf{t}_2 \qquad (5)$$

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* $\mathsf{begin}.\eta$ are channels explicitly associated with or tagged by that type (or, because of subtyping, by a $\mathsf{begin}.\eta'$ subtype of $\mathsf{begin}.\eta$: more about that later on).

We do not include in our session descriptors a construct for parallel composition (as opposed to [21, 3], for example). Since we assume an interleaving semantics of parallel composition, having two different choices is enough for faithfully describing possibly concurrent actions by means of well-known expansion laws (see [11] for an example).

2.2 Semantics of types and descriptors

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 outline a denotational semantics for types and descriptors, then we give the formal definition of duality in terms of a labeled 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 $[\![.]\!]$ such that $[\![\mathsf{t}]\!] = \{\mathsf{v} \mid \vdash \mathsf{v} : \mathsf{t}\}$ and that $[\![\mathsf{t} \wedge \mathsf{s}]\!] = [\![\mathsf{t}]\!] \cap [\![\mathsf{s}]\!]$, $[\![\mathsf{t} \vee \mathsf{s}]\!] = [\![\mathsf{t}]\!] \cup [\![\mathsf{s}]\!]$, and $[\![\neg \mathsf{t}]\!] = \mathscr{V} \setminus [\![\mathsf{t}]\!]$ (where $\mathscr{V}$ denotes the set of all values). The same interpretation can then be used to *define* the subtyping relation (denoted by "$<:$") as follows:

$$\mathsf{t} <: \mathsf{s} \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad [\![\mathsf{t}]\!] \subseteq [\![\mathsf{s}]\!]$$

The technical machinery to define an interpretation with such properties and solve the 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 [15]: we take it for granted and no longer bother about it if not for session types that are dealt with in Section 2.3. This interpretation of types justifies the use we do henceforward of the notation $\mathsf{v} \in \mathsf{t}$ 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. 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 $!\mathsf{Int}.?(\mathsf{Bool} \vee \mathsf{Int}).\mathsf{end}$ is dual to (1), too.

Note also that some session descriptors have no dual, for example $?(\mathsf{Bool} \wedge \mathsf{Int}).\mathsf{end}$, since no process can send a value that is both a Boolean and an integer: the intersection is empty.[1] 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 one dual exists, and that we dub *viable descriptors*. We write $\eta \bowtie \theta$ if η and θ are dual (duality is a symmetric relation). Then, we can define the interpretation of a descriptor as the set of its duals: $[\![\eta]\!] = \{\theta \mid \eta \bowtie \theta\}$; extend it set-theoretically to sieves: $[\![\chi \wedge \chi']\!] = [\![\chi]\!] \cap [\![\chi']\!]$, $[\![\chi \vee \chi']\!] = [\![\chi]\!] \cup [\![\chi']\!]$, $[\![\neg\chi]\!] = \mathscr{S} \setminus [\![\chi]\!]$ (where $\mathscr{S}$ denotes the set of all *viable* descriptors); and use it to semantically define the subsieving (and subsessioning) relation (denoted by "$\leq$"):

$$\chi \leq \chi' \quad \stackrel{\text{def}}{\iff} \quad [\![\chi]\!] \subseteq [\![\chi']\!] \tag{6}$$

Duality plays a central role also in defining the semantics of types. 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 $\mathsf{begin}.\eta$. As suggested in Section 2.1, we can take as a value of a session type a public 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 a different (larger) type is expected. The key is to understand how a public session channel is "used". We make the assumption—matched by everyday practice—that there is unique way to consume a public channel c of type $\mathsf{begin}.\eta$, by invoking the service associated with c and starting a conversation that conforms to the protocol described by η. Thus it is safe to replace c with a different channel d of a smaller type $\mathsf{begin}.\eta'$ only if the conversation, which follows the protocol described by η and which was originally intended to occur with the service associated with c, works seamlessly with the service associated with d. This happens if at each step of the conversation the service associated with d is willing to receive at least all the messages accepted by c and never sends any message that c would not send. Roughly speaking, the service associated with d is "more tolerant" than the one associated with c. Of course, there are fewer services that, as d, support a η' conversation, since they must be able to satisfy more demanding clients. Therefore, passing from $\mathsf{begin}.\eta$ to $\mathsf{begin}.\eta'$ corresponds to restricting the set of possible services one can safely use, that is to say, reducing the sets of possible duals. So the intuition—that we will formalize in Section 2.3 by equation (8)—is that $\mathsf{begin}.\eta' <: \mathsf{begin}.\eta$ if and only if η' has fewer duals than η, that is by (6), $\eta' \leq \eta$. For instance we have that $?\mathsf{Int}.\mathsf{end} \leq ?\mathsf{Real}.\mathsf{end}$ since every descriptor that is dual of $?\mathsf{Int}.\mathsf{end}$ is also dual of $?\mathsf{Real}.\mathsf{end}$. Similarly $\mathsf{begin}.?\mathsf{Int}.\mathsf{end} <: \mathsf{begin}.?\mathsf{Real}.\mathsf{end}$ since if a process that uses a channel of type $\mathsf{begin}.?\mathsf{Real}.\mathsf{end}$ is well typed, then the process obtained by replacing this channel for a different one of type $\mathsf{begin}.?\mathsf{Int}.\mathsf{end}$ is well typed as well: it will receive an integer number in a place where it expects a real number.

Since we want our types to satisfy the strong disjunction property (5), then a public channel c must be tagged by types of the form $\mathsf{begin}.\eta$ (and not, say, $\mathsf{begin}.\eta \vee \mathsf{begin}.\eta'$), which yields the following interpretation for session types: $[\![\mathsf{begin}.\eta]\!] = \{\mathsf{c}^{\mathsf{begin}.\eta'} \mid \forall\theta,\ \theta \bowtie \eta' \Rightarrow \theta \bowtie \eta\}$, that is

$$[\![\mathsf{begin}.\eta]\!] = \{\mathsf{c}^{\mathsf{begin}.\eta'} \mid \eta' \leq \eta\} \tag{7}$$

The next step is to formally define the duality relation.

2.2.2 Semantics of session descriptors

The formal semantics of a descriptor can be given by resorting to the labeled transition system (LTS) defined by the rules

$$(\text{TR}1)\ \frac{}{\mathsf{end} \xrightarrow{\checkmark} \mathsf{end}} \qquad (\text{TR}2)\ \frac{}{\eta \oplus \eta' \longrightarrow \eta} \qquad (\text{TR}3)\ \frac{\eta \longrightarrow \eta'}{\eta + \eta'' \longrightarrow \eta' + \eta''}$$

$$(\text{TR}4)\ \frac{\eta \xrightarrow{\mu} \eta'}{\eta + \eta'' \xrightarrow{\mu} \eta'} \qquad (\text{TR}5)\ \frac{\eta \xrightarrow{!\mathsf{v}} \eta''}{\eta + \eta' \longrightarrow \eta} \qquad (\text{TR}6)\ \frac{\eta \xrightarrow{!\eta''} \eta'''}{\eta + \eta' \longrightarrow \eta}$$

$$(\text{TR}7)\ \frac{\mathsf{v} \in \mathsf{t}}{?\mathsf{t}.\eta \xrightarrow{?\mathsf{v}} \eta} \qquad (\text{TR}8)\ \frac{\mathsf{v} \in \mathsf{t}}{!\mathsf{t}.\eta \xrightarrow{!\mathsf{v}} \eta} \qquad (\text{TR}9)\ \frac{\eta \in \chi}{?\chi.\eta' \xrightarrow{?\eta} \eta'} \qquad (\text{TR}10)\ \frac{\eta \in \chi}{!\chi.\eta' \xrightarrow{!\eta} \eta'}$$

plus the symmetric of rules (TR2-TR6). In the rules μ ranges over actions of the form $!\mathsf{v}$, or $?\mathsf{v}$, or $!\eta$, or $?\eta$, or $\checkmark$.

[1] This shows that our duality is semantically defined: $?(\mathsf{Bool} \wedge \mathsf{Int}).\mathsf{end}$ is *not* dual of $!(\mathsf{Bool} \wedge \mathsf{Int}).\mathsf{end}$ as a syntactic approach would suggest; both descriptors have no dual.

Rules (TR1-TR4) are straightforward. Rules (TR7-TR8) state that the synchronization is performed on single values (strictly speaking, on singleton types) rather than on generic types. This is closer to what happens in practice, since $!\mathsf{t}.\eta$ indicates that the descriptor is ready to emit some value of type t (TR8), while $?\mathsf{t}.\eta$ 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 [24] (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 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 $\mathsf{begin}.\eta$). 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 behavior 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, as 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 $\eta \in [\![\chi]\!]$.

Rules (TR5-TR6) state that outputs are irrevocable. This is a characteristic peculiar to our system and is reminiscent of Castellani and Hennessy's treatment of external choices in the asynchronous CCS [8]. 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 behavior 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 labeled 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 formalize the notion of duality it is then handy to characterize 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 $\stackrel{\mu}{\Longrightarrow}$ for $\Longrightarrow\stackrel{\mu}{\longrightarrow}\Longrightarrow$; we write $\eta \stackrel{\mu}{\longrightarrow}$ if there exists η' such that $\eta \stackrel{\mu}{\longrightarrow} \eta'$, and similarly for $\stackrel{\mu}{\Longrightarrow}$; we write $\eta \not\longrightarrow$ 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 $\eta \Longrightarrow \eta' \not\longrightarrow$ and $\eta' \stackrel{\mu}{\longrightarrow}$ and μ is either $!\mathsf{v}$, or $!\eta$, or $\checkmark$.*

We say that η must input *μ, written $\eta \Downarrow \mu$, if $\eta \Longrightarrow \eta' \not\longrightarrow$ implies $\eta' \stackrel{\mu}{\longrightarrow}$ and μ is either $?\mathsf{v}$, 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 $!\mathsf{Int.end} \oplus \mathsf{end} \downarrow !3$ and $!\mathsf{Int.end} \oplus \mathsf{end} \downarrow \checkmark$; on the other hand we have $!\mathsf{Int.end} + \mathsf{end} \not\downarrow \checkmark$, since $!\mathsf{Int.end} + \mathsf{end} \not\longrightarrow \mathsf{end}$. Similarly we have $?\mathsf{Int.end} \oplus ?\mathsf{Real.end} \Downarrow ?3$ because the action $?3$ is always guaranteed independently of the internal choice, whereas $?\mathsf{Int.end} \oplus ?\mathsf{Real.end} \not\Downarrow ?\sqrt{2}$ because $?\mathsf{Int.end} \oplus ?\mathsf{Real.end} \longrightarrow ?\mathsf{Int.end}$ and $?\mathsf{Int.end} \not\Downarrow ?\sqrt{2}$.

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 $\eta \Longrightarrow \eta' \not\longrightarrow$ 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 $\eta \not\downarrow$ and $\eta \not\Downarrow$ to denote their respective negations.*

Note that the two contractivity conditions of Definition 2.1 rule out behaviors 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 labeled 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 $?\mathsf{Real}.!\mathsf{Int}.\mathsf{end} + ?\mathsf{Int}.!\mathsf{Bool}.\mathsf{end} \xrightarrow{?3} !\mathsf{Bool}.\mathsf{end}$ (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 $!\mathsf{Int}.\mathsf{end} \oplus !\mathsf{Bool}.\mathsf{end}$, which accounts for all of the possible states in which the receiver can be after the reception of 3. The *objective evolution* of a session descriptor after an action μ is defined next.

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

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

We now have all the ingredients for formally defining duality.[2]

Definition 2.5 (Duality) *Let the* dual *of a label μ, written $\overline{\mu}$, be defined by: (i) $\overline{\checkmark} = \checkmark$; (ii) $\overline{\dagger\mathsf{v}} = \dagger\mathsf{v}$; (iii) $\overline{\dagger\eta} = \dagger\eta$; where $\overline{!} = ?$ and $\overline{?} = !$. Then $\eta_1 \bowtie \eta_2$ is the largest symmetric 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 $\eta_2 \Downarrow \overline{\mu}$ and $\eta_1\langle\mu\rangle \bowtie \eta_2\langle\overline{\mu}\rangle$ for every μ.*

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. For example, we have $?\mathsf{Real}.!\mathsf{Int}.\mathsf{end} + ?\mathsf{Int}.!\mathsf{Bool}.\mathsf{end} \bowtie !\mathsf{Int}.?(\mathsf{Int} \vee \mathsf{Bool}).\mathsf{end}$, but $?\mathsf{Real}.!\mathsf{Int}.\mathsf{end} + ?\mathsf{Int}.!\mathsf{Bool}.\mathsf{end} \not\bowtie !\mathsf{Int}.?\mathsf{Int}.\mathsf{end}$ because the descriptor on the right is not sure that its partner will answer with an integer. However $?\mathsf{Real}.!\mathsf{Int}.\mathsf{end} + ?\mathsf{Int}.!\mathsf{Bool}.\mathsf{end} \bowtie !(\mathsf{Real} \setminus \mathsf{Int}).?\mathsf{Int}.\mathsf{end}$. As another example, we have $?\mathsf{Int}.\mathsf{end} \oplus ?\mathsf{Real}.\mathsf{end} \bowtie !\mathsf{Int}.\mathsf{end}$ because $?\mathsf{Int}.\mathsf{end} \oplus ?\mathsf{Real}.\mathsf{end} \Downarrow ?\mathsf{v}$ for every $\mathsf{v} \in \mathsf{Int}$, however $?\mathsf{Int}.\mathsf{end} \oplus ?\mathsf{Real}.\mathsf{end} \not\bowtie !\sqrt{2}.\mathsf{end}$ because $?\mathsf{Int}.\mathsf{end} \oplus ?\mathsf{Real}.\mathsf{end} \not\Downarrow ?\sqrt{2}$.

Using the definition of duality it is easy to see that $[\![\eta \oplus \eta']\!] = [\![\eta \wedge \eta']\!]$ 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 2.6 *$\theta \leq \chi_1 \vee \chi_2 \iff \theta \leq \chi_1$ or $\theta \leq \chi_2$.*

This property is essential to prove decidability of $\leq$.

2.3 Subtyping

Now that we have defined the duality relation, and therefore subsessioning, 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 $\mathsf{begin}.\eta$, plus other basic types and type constructors (inherited from the host language) that we left unspecified (typically, Real, Bool, $\times$, ...). We define the subtyping relation semantically using the technique defined in [15] 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 $\mathsf{t}_1 <: \mathsf{t}_2$ if and only if $\mathsf{t}_1 \wedge \neg\mathsf{t}_2 <: \emptyset$ (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 [6, 15] for details). In conclusion, in order to define our subtyping relation all we need is to decide when $\bigvee_{k\in K}(\bigwedge_{i\in I_k} \mathsf{begin}.\eta_i \wedge \bigwedge_{j\in J_k} \neg\mathsf{begin}.\eta_j) <: \emptyset$. 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 $\bigwedge_{i\in I} \mathsf{begin}.\eta_i <: \bigvee_{j\in J} \mathsf{begin}.\eta_j$.

As regards session channels, we notice that a value has type $(\mathsf{begin}.\eta) \wedge (\mathsf{begin}.\eta')$ if and only if it has type $\mathsf{begin}.(\eta \oplus \eta')$. Also note that $\mathsf{begin}.\eta <: \mathsf{begin}.\eta_1 \vee \mathsf{begin}.\eta_2$ if and only if $\mathsf{begin}.\eta <: \mathsf{begin}.\eta_1$ or $\mathsf{begin}.\eta <: \mathsf{begin}.\eta_2$. Therefore the semantic subtyping relation for the types of Section 2.1 is completely defined by (the semantic subtyping framework of [15] and) the following equation

$$\bigwedge_{i\in I} \mathsf{begin}.\eta_i <: \bigvee_{j\in J} \mathsf{begin}.\eta_j \iff \exists j\in J : \bigoplus_{i\in I} \eta_i \leq \eta_j \quad (8)$$

Note that when in the equation above I and J are singletons it reduces to

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

that is the form discussed at the end of Section 2.2.1.

[2] Duality and LTS may give the impression of being circularly defined. In the full version [5] we prove that this circularity is only apparent and the definitions well-founded thanks to the stratification we hinted at in Section 2.1.

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 [10] (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 the class of descriptors that admit at least one dual. Recall that η is *viable* if there exists η' such that $\eta \bowtie \eta'$. Any non-viable descriptor is the least element of $\leq$, which henceforward will be denoted by $\bot$.

Definition 2.7 (Coinductive Viability) *$\eta^{\bowtie}$ is the largest predicate over descriptors such that either*

1. *$\eta \downarrow$ and $\eta \downarrow \mu$ implies $\eta\langle\mu\rangle^{\bowtie}$ for every μ, or*
2. *there exists μ such that $\eta \Downarrow \mu$ and $\eta\langle\mu\rangle^{\bowtie}$.*

The definition provides us with a correct and complete characterization of viable descriptors:

Proposition 2.8 *$\eta^{\bowtie}$ if and only if η is viable.*

We can now read the statement of Definition 2.7 in the light of the result of the above proposition: Definition 2.7 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 2.9 (Coinductive Subsession) *$\eta \leqq \eta'$ is the largest relation between session descriptors such that $\eta^{\bowtie}$ implies $\eta'^{\bowtie}$ and*

1. *$\eta' \not\Downarrow$ and $\eta' \downarrow \mu$ imply $\eta \downarrow \mu$ with $\eta\langle\mu\rangle \leqq \eta'\langle\mu\rangle$, and*
2. *$\eta \Downarrow \mu$ and $\eta\langle\mu\rangle^{\bowtie}$ imply $\eta' \Downarrow \mu$ with $\eta\langle\mu\rangle \leqq \eta'\langle\mu\rangle$, 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 subsession 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 $?\mathsf{Int}.\mathsf{end} \not\leqq ?\mathsf{Int}.\mathsf{end} + \mathsf{end}$ 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 $?\mathsf{Int}.\mathsf{end} \leqq ?\mathsf{Int}.\mathsf{end} + \mathsf{end}$. 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 $\eta\langle\mu\rangle^{\bowtie}$ 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 $?\mathsf{Int}.!\emptyset.\mathsf{end} + ?\mathsf{Bool}.\mathsf{end} \not\leqq ?\mathsf{Bool}.\mathsf{end}$, 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 $\not\leqq$. 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 2.10 $\eta_1 \leqq \eta_2 \iff \eta_1 \leq \eta_2$.

It is possible to derive several interesting algebraic laws from the definition of $\leq$. These are discussed in the full version [5] of the article and used in the proofs of the existence of normals forms and of correctness of the algorithms. Here we just state the following derivable decomposition laws:

$$?\mathsf{t}.\eta + ?\mathsf{s}.\eta' = ?(\mathsf{t} \setminus \mathsf{s}).\eta + ?(\mathsf{s} \setminus \mathsf{t}).\eta' + ?(\mathsf{t} \wedge \mathsf{s}).(\eta \oplus \eta')$$
$$!\mathsf{t}.\eta \oplus !\mathsf{s}.\eta' = !(\mathsf{t} \setminus \mathsf{s}).\eta \oplus !(\mathsf{s} \setminus \mathsf{t}).\eta' \oplus !(\mathsf{t} \wedge \mathsf{s}).(\eta \oplus \eta')$$

the latter rule holding when none of the sets $\mathsf{t} \setminus \mathsf{s}$, $\mathsf{s} \setminus \mathsf{t}$, and $\mathsf{t} \wedge \mathsf{s}$ 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 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.

2.5 Algorithms

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 $\chi \wedge \neg\chi' \leq \bot$. The l.h.s. can be rewritten in disjunctive normal form whose definition for sieves is (we convene that $\bigvee_{i\in\emptyset} \chi_i = \sum_{i\in\emptyset} \eta_i = \bot$):

Definition 2.11 (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 2.6) 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 (8)). 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 2.12 (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 $i \in I, j \in J$, λ_{ij} is in strong normal form and $\bigwedge_{j\in J}\lambda_{ij} \neq \bot$ 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 $\bigoplus_{i\in I}!\psi_i.\eta_i\{\oplus \mathsf{end}\}$ or $\sum_{i\in I}?\psi_i.\eta_i\{+\mathsf{end}\}$, where for all $i \in I$, $\psi_i \neq \varnothing$, ψ_i and η_i are in strong normal form and for all $i, j \in I$, $i \neq j$ implies $\psi_i \wedge \psi_j = \varnothing$, and* end *is possibly missing.*

Transformation in strong normal form is effective:

Theorem 2.13 (Normalization) *For every sieve χ it is possible to effectively 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 $\bot$, and then apply the algorithm whose core rules are given below:

(END) $\dfrac{}{\mathsf{end} \leq \mathsf{end}}$ (PREFIX) $\dfrac{\eta \leq \eta'}{\alpha.\eta \leq \alpha.\eta'}$ (MIX-CHOICES) $\dfrac{}{\bigoplus_{i\in I}\eta_i \oplus \mathsf{end} \leq \sum_{j\in J}\eta'_j + \mathsf{end}}$

(EXT-CHOICES) $\dfrac{I \subseteq J \qquad \eta_i \leq \eta'_i \ {}^{(\forall i\in I)}}{\sum_{i\in I}\eta_i \leq \sum_{j\in J}\eta'_j}$ (INT-CHOICES) $\dfrac{J \subseteq I \qquad \eta_j \leq \eta'_j \ {}^{(\forall j\in J)}}{\bigoplus_{i\in I}\eta_i \leq \bigoplus_{j\in J}\eta'_j}$

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 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 stated at the end of Section 2.4. The corresponding algorithmic rules can be found in the full version [5].

Theorem 2.14 (Soundness and Completeness) *The algorithm is sound and complete with respect to $\leq$ and it terminates.*

Duality. Duality can be reduced to subsessioning since $\eta \bowtie \eta'$ if and only if $\overline{\eta} \leq \eta'$, where we write $\overline{\eta}$ for the *canonical dual* of η, namely the least descriptor in the set-theoretic interpretation of η. Computing $\overline{\eta}$ is trivial once η is in strong normal form (see Theorem 2.13): it suffices to change every ? into !, every $+$ into $\oplus$ and viceversa, and to coinductively apply the transformation to the continuations leaving end descriptors unchanged. Regularity ensures that the transformation terminates (by using memoization techniques) and showing that the obtained session descriptor is the canonical dual of η is a trivial exercise.

3 A flavor of typing

All monomorphic dyadic session type theories proposed in the literature are particular cases of the session descriptors discussed here. In this section we just hint at why this holds true and outline the new usages that our discipline makes possible. All details are given in the full version [5].

The first version of session descriptors can be traced back to the seminal work of Honda, Vasconcelos, and Kubo [20] in the context of process algebras. In the same context [2] enhances sessions with correspondence assertions. Later on Vasconcelos, Gay, and Ravara have extended the approach to multithreaded functional languages [26]. In all these works session descriptors embed two n-ary operators for internal and external choice, which are strictly coupled with the communication of labels that indicate the selected branch in a choice. The session descriptors of [20, 2, 26] can be written in our model as follows:

$$\begin{array}{rcl} \eta & ::= & \mathsf{end} \mid \alpha.\eta \mid \bigoplus_{i\in I}!\ell_i.\eta_i \mid \Sigma_{i\in I}?\ell_i.\eta_i \\ \alpha & ::= & !\mathsf{t} \mid ?\mathsf{t} \mid !\eta \mid ?\eta \end{array}$$

where we consider labels as singleton types. The same syntax can be used to describe the session descriptors developed for CORBA [25], Boxed Ambients [16], and the ones used to type the Conversation Calculus [3].

In the context of object-oriented languages, session type theories assuring type safety and progress were investigated in MOOSE [13], AMOOSE [9], and MOOSE$_{<:}$ [12]. The

syntax of all session descriptors discussed in the above papers, but for [12] where bounded polymorphism is considered, are special cases of our model, after identifying recursive types with their infinite unfolding. More specifically, the syntax of the session descriptors in [13] and [9] can be reduced to:

$$\eta \;::=\; \mathsf{end} \mid \alpha.\eta \mid !\mathsf{true}.\eta \oplus !\mathsf{false}.\eta \mid ?\mathsf{true}.\eta + ?\mathsf{false}.\eta$$
$$\alpha \;::=\; !\mathsf{t} \mid ?\mathsf{t} \mid !\eta \mid ?\eta$$

since these works deal with sessions where branching is controlled by means of boolean conditions.

A calculus that amalgamates the notion of session-based communication with the one of object-oriented programming is presented in [4, 1]. In these works sessions and methods are unified, channels are implicit, and delegation is realized by means of session calls. Branching is determined by the runtime type of the object being communicated. Session descriptors with this form of dependencies can be written very naturally in our model as:

$$\eta \;::=\; \mathsf{end} \;\mid\; \textstyle\bigoplus_{i\in I} !\mathsf{C}_i.\eta_i \;\mid\; \Sigma_{i\in I} ?\mathsf{C}_i.\eta_i$$

where C_i are class names in [1] and generic class names in [4]. [3] In the full version [5] we show how we can straightforwardly enhance the typing of [1] using our session descriptors: in particular we enable *session overloading*, that is, we allow the same session name to be declared with different session descriptors in the class hierarchy. At run time an appropriate session body will be chosen using the duality and subsieving relations.

The most interesting observation on process typing is that with our framework sessions can be typed in the standard π-calculus with internal/external choices and bound-/free outputs (with the single modification on the type-based selection of external choices we described in the introduction), *without primitive operators tailored to session-based communications* (in the spirit of Kobayashi's [23]). The intuition is that bound outputs, written $\mathsf{c}!(x : \eta)$, are session initiations where c is the public channel of the session and x the session private channel of descriptor η; free outputs are reserved for session communications/delegations, and inputs are either session communications or session connections according to whether they are meant to synchronize with free or bound outputs, respectively. For example, the following process models a node that handles communications described by a protocol η and delegates communications described by unknown protocols to a sibling node, in a token-ring fashion:

NODE(mypublicname, nextpublicname, η, P) =
  mypublicname?$(x : ?\top.\mathsf{end})$. /* accept */
    $x?(y : \eta).P$ /* catch&handle */
  + $x?(y : \neg\eta)$. /* catch&delegate */
    nextpublicname!$(z : !\top.\mathsf{end})$. /* request */
    $z!y$ /* throw */

The node waits for a connection on its public channel mypublicname and, once the connection is made, catches on the established session channel x a delegated session y of an arbitrary descriptor ($\top$ is the top sieve). If the delegated session is of protocol η (this is checked by using an external choice), then the node handles the session in the process P, otherwise it connects to a sibling node nextpublicname (via a bound output) and delegates y to it (via a free output). Both mypublicname and nextpublicname are public channels of type $\mathsf{begin}.!\top.\mathsf{end}$. All details on typing the generic π-calculus outlined here are given in the full paper [5].[4]

4 Conclusions and related work

We have defined a semantic theory of session types by subverting the usual session type presentations, where the subtyping (and subsession) relations are introduced first, and then shown to be sound. Here we have focused on duality as the main characterizing feature, and defined subtyping and subsessioning in terms of it.

We claim that our theory of session types is minimal—insofar as the addition of any further constructor such as parallel composition or labeled synchronization would restrict the programming language to which the framework could be applied—and yet complete. The key ingredients of our theory are communication primitives, behavioral composition operators for describing branching points, and boolean composition operators for types and sieves. All the existing proposal of session types of comparable expressiveness can be obtained by using suitable combinations of these ingredients and our session types can be used for typing generic π-calculus processes without any dedicated primitive for session management.

The relation of our work with some others defining theories of session types is already explored in Section 3. Subtyping relations for session types are studied in [18, 17]. In these works the definition of the subtyping relation is driven by the observation that one can safely replace a session by another that externally offers more choices and internally can make less choices. Since in the cited works choices are guarded 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 external choices that are driven by the messages being exchanged rather than by labels: when a session offers an output it will be able to synchronize only with the branches of a choice that accept that output. The resulting subtyping relation generalizes the safe substitutability principle of the existing settings by permitting (a combination of) branches of a choice to subsume another set of branches. Furthermore, while in the cited works the subtyping relation is defined coinductively and axiomatically, we characterize the

[3] The cited works hardcode different selection policies for branching. We can easily reproduce them all by using Boolean combinators on classes.

[4] The type system in the full version ensures not only *fidelity* of communications (i.e., once a session is started the communications are in the expected order and exchange data of the expected types), but also *progress* (i.e., a session can only be stuck at the initialization of a new session).

relation semantically, and this captures directly the desired safety property.

Moving from label-driven to type-driven branch selection may seem a regression, insofar as it demands run-time type checking. This is not so in practice. First, when sessions are used as in any of the existing session types proposals, branching can be easily optimized by reducing run-time type checking to label/class matching. Second, general value-based dispatching can be implemented very efficiently anyway [14], with the exception of session type values which may require to check subsessioning. In any case, it is reasonable to assume that the matching overhead is negligible with respect to latency time of communications typical of service-oriented computing.

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 [8] for asynchronous CCS, where outputs cannot be blocked even if they guard external choices (we called this property "output irrevocability"). However, in our setting output signals are allowed to have a continuation. From a technical viewpoint in this work we introduce several novelties. We devise a new labeled 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. As regards future research, the natural continuation of this work is to extend the semantic framework we propose to the features that we cannot account for yet. Specifically, we aim at studying polymorphism (to model session descriptors of [17]), exploring communication models other than output irrevocability, and extending our types to multi-party sessions.

References

[1] L. Bettini, S. Capecchi, M. Dezani-Ciancaglini, E. Giachino, and B. Venneri. Session and union types for object-oriented programming. In *Concurrency, Graphs and Models*, LNCS 5065. Springer, 2008.

[2] E. Bonelli, A. Compagnoni, and E. Gunter. Correspondence Assertions for Process Synchronization in Concurrent Communications. *J. Funct. Progr.*, 15(2):219–248, 2005.

[3] L. Caires and H. T. Vieira. Conversation Types. In *ESOP '09*, LNCS 5502. Springer, 2009.

[4] S. Capecchi, M. Coppo, M. Dezani-Ciancaglini, S. Drossopoulou, and E. Giachino. Amalgamating Sessions and Methods in Object-Oriented Languages with Generics. *Theor. Comput. Sci.*, 410(2-3):142–167, 2008.

[5] G. Castagna, M. Dezani-Ciancaglini, E. Giachino, and L. Padovani. Foundation of session types. Full version at `http://hal.archives-ouvertes.fr/hal-00334435`.

[6] G. Castagna and A. Frisch. A gentle introduction to semantic subtyping. In *PPDP '05 and ICALP '05*, 2005. Joint ICALP-PPDP keynote talk.

[7] G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. Extended version of the article in *POPL '08*, submitted, available on authors' web pages, 2008.

[8] I. Castellani and M. Hennessy. Testing theories for asynchronous languages. In *FST&TCS '98*, LNCS 1530. Springer, 1998.

[9] M. Coppo, M. Dezani-Ciancaglini, and N. Yoshida. Asynchronous session types and progress for object-oriented languages. In *FMOODS'07*, LNCS 4468, 2007.

[10] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theor. Comput. Sci*, 34:83–133, 1984.

[11] R. De Nicola and M. Hennessy. CCS without τ's. In *TAPSOFT/CAAP'87*, LNCS 249. Springer, 1987.

[12] M. Dezani-Ciancaglini, S. Drossopoulou, E. Giachino, and N. Yoshida. Bounded session types for object-oriented languages. In *FMCO'06*, LNCS 4709. Springer, 2007.

[13] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In *ECOOP'06*, LNCS 4067. Springer, 2006.

[14] A. Frisch. Regular tree language recognition with static information. In *IFIP TCS*, pages 661–674. Kluwer, 2004.

[15] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *J. ACM*, 55(4):1–64, 2008.

[16] P. Garralda, A. Compagnoni, and M. Dezani-Ciancaglini. BASS: Boxed Ambients with Safe Sessions. In *PPDP'06*, pages 61–72. ACM Press, 2006.

[17] S. Gay. Bounded Polymorphism in Session Types. *MSCS*, 18(5):895–930, 2008.

[18] S. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.

[19] K. Honda. Types for dyadic interaction. In *CONCUR'93*, LNCS 715. Springer, 1993.

[20] 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.

[21] A. Igarashi and N. Kobayashi. A Generic Type System for the Pi-Calculus. *Theor. Comput. Sci*, 311(1-3):121–163, 2004.

[22] N. Kobayashi. Type systems for concurrent programs. In *FMC'03*, LNCS 2757. Springer, 2003.

[23] N. Kobayashi. Type systems for concurrent programs. Extended version of [22], Tohoku University, 2007.

[24] R. Milner, J. Parrow, and D. Walker. Modal Logics for Mobile Processes. *Theor. Comput. Sci.*, 114(1):149–171, 1993.

[25] A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the Behavior of Objects and Components using Session Types. *Fund. Inf.*, 73(4):583–598, 2006.

[26] V. T. Vasconcelos, S. Gay, and A. Ravara. Typechecking a Multithreaded Functional Language with Session Types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006.

[27] N. Yoshida and V. T. Vasconcelos. Language primitives and type disciplines for structured communication-based programming revisited. In *SecRet'06*, ENTCS 171, 2007.