Contracts for Mobile Processes

Giuseppe Castagna[1] and Luca Padovani[2]

[1] CNRS, Laboratoire Preuves, Programmes et Systèmes, Université Paris Diderot – Paris 7
[2] Istituto di Scienze e Tecnologie dell'Informazione, Università di Urbino

Abstract. Theories identifying well-formed systems of processes—those that are free of communication errors and enjoy strong properties such as deadlock freedom—are based either on session types, which are inhabited by channels, or on contracts, which are inhabited by processes. Current session type theories impose overly restrictive disciplines while contract theories only work for networks with fixed topology. Here we fill the gap between the two approaches by defining a theory of contracts for so-called mobile processes, those whose communications may include delegations and channel references.

1 Introduction

Research on communicating processes has recently focused on the study of static analysis techniques for characterizing those systems that enjoy desirable properties such as the conformance to specifications, the absence of communication errors, and progress. By progress we mean the guarantee that the system will either evolve into a so-called successful state, in which all of its components have completely carried out their task, or that the system will continue to evolve indefinitely, without ever getting stuck.

Two approaches have been studied in great depth: those based on *session types*, and those based on *behavioral contracts*. Session types arise in a type theoretic framework as an evolution of standard channel types [29], by taking into account the fact that the same channel can be used at different times for sending messages of different kind and type. For example, session types defined in [21] describe potentially infinite, dyadic conversations between processes as sequences of messages that are either sent or received. The same approach has been applied to multi-threaded functional programming languages [30], as well as to object-oriented languages [15, 3]. Behavioral contracts arise in a process algebraic framework where the behavior of the component of a system is approximated by means of some term in a process algebra. The contract does not usually reveal how the component is implemented, but rather what its observable behavior is. Theories of contracts for Web services have been introduced in [12] and subsequently extended in [25, 14, 28]. Independently, theories of contracts have been developed for reasoning on Web service choreographies in [6, 7].

Traditionally, the approaches based on session types have always allowed the description of systems where channels and opened sessions can be communicated and delegated just as plain messages. Conversely, all of the works on contracts mentioned above rely on CCS-like formalisms, which makes them suitable for describing systems whose topology is fixed and where the role of each component does not change over time. This work aims at being a first step in filling the gap that concerns the expressiveness of the two approaches and in defining a contract language to describe processes that collaborate not only by exchanging messages, but also by delegating tasks, by dynamically initiating new conversations, by joining existing conversations.

Since we want to describe conversations where exchanged messages may include channels, it is natural to propose a contract language based on the π-calculus. However, quite a few aspects force us to depart from standard presentations. The first aspect regards the interpretation of *actions*. A contract should permit to describe not only the communication of channels, but also of other kinds of messages. Thus we will have actions of the form $c?\texttt{Int}$ or $c!\texttt{String}$ for denoting the ability to receive (resp. send) a value of type `Int` (resp. `String`). In fact, we will generalize input and output actions so that their subjects are possibly structured *patterns* describing all and only the messages that can be exchanged on a channel at a given time. Since we are interested in describing the observable behavior of a process rather than its internal structure, we will mostly focus on two operators $+$ and $\oplus$ representing external and internal choices respectively. This is consistent with the works on session types and is also the approach taken in some presentations of CCS [17, 20] and of π-calculus [26]. For example, the contract $c?\texttt{Int}.T + d?\texttt{String}.S$ describes a process that behaves as T if it receives a number on the channel c, and that behaves as S if it receives a string on the channel d. Dually, the contract $c!\texttt{Int}.T \oplus d!\texttt{String}.S$ describes a process that internally decides whether to send an integer or a string, and that behaves as T or as S accordingly. Additionally, we let branching be dependent also on the (type of) exchanged messages, not just on the channel on which communication occurs. This allows us to model behaviors that are affected by the actual values that are communicated, which is just what happens in session type theories with branching and selection constructs whose behavior is driven by labels. For instance, the contract $c?\texttt{Int}.T + c?\texttt{String}.S$ describes a process that behaves either as T if it receives an integer value on the channel c or as S if it receives a string value *on the same channel*. Unlike the π-calculus and along the lines of [1], we distinguish several "terminal" behaviors: $\mathbf{0}$ represents a deadlock process that cannot make any further progress; $\mathbf{1}$ represents successful termination; Ω represents an autonomous process that is capable of making indefinite progress.

With respect to current literature on session types we innovate also from the methodological point of view. Current presentations of session type theories begin by defining the semantics of types and then prove that well-typed processes enjoy certain properties. We embrace a *testing* approach [17, 5, 20] and go the other way round: we start by defining the properties we are interested in and then study the types that ensure them. More precisely, we start from the concept of *well-formed system*, which is a parallel composition of participants—each one abstracted by means of its contract—that mutually satisfy each other: no participant ever gets stuck waiting for messages that are never sent or trying to send messages that no one else is willing to receive. Well-formedness gives us the one essential notion that drives the whole theory: a component T is *viable* if there is a well-formed system that includes T. Viability roughly corresponds to the notion of well-typed process in session type theories. Also, well-formedness gives us a semantic way of relating contracts: T and S are *equivalent* if they yield well-formed systems when combined with the same components. The equivalence relation induced by well-formedness can be used for verifying whether a participant respects its specification given as a contract, or for understanding when two participants are equivalent by comparing their contracts, or for querying a database of participants by means of their contract, along the lines of what has been done in the CCS context. Additionally,

the very same equivalence relation can be used for assessing properties of participants and systems: a contract T is viable if and only if it is not equivalent to $\mathbf{0}$; a system S is well-formed if $S \simeq \mathbf{1} + S$; a system S has indefinite progress if $S \simeq \Omega + S$.

We can identify three main contributions of this work. First, we propose a natural extension of the contract language presented in [12, 25, 14] for describing processes that communicate channels in addition to plain messages. Second, we define a notion of well-formed system as a system whose stable states, those not allowing any further progress, are such that *all* components have terminated successfully. The resulting equivalence relation is inspired to the classical *must* preorder for mobile processes [5, 26], except that we shift the attention from the success of a particular process (the "test" in the classical framework), to the success of all the components in the system. Well-formedness extends the notion of correct composition in [6, 7] to a name-passing scenario. Finally, we show how contracts provide an alternative and somehow naïve approach to the typing of processes. Indeed we believe that the increasing complexity of session type systems comes from the fact that they are used for guaranteeing properties they were not originally designed for. Session types, which characterize channels, are types with a behavioral flavor. We say "flavor" because, as a matter of facts, channels (like any other value) do not expose any behavior *per se*. Session types do not characterize the behavior of the channels they type; they just *reflect* the behavior of the processes that use those channels. This is evident in the inference rules of every session type system, where a process is *not* associated with a type that abstracts its behavior, but with a set of type assignments that provides a partial and distributed description of how the process independently uses each of its channels. It is thus unsurprising that session types systems guaranteeing global properties—most notably progress—must necessarily rely on draconian restrictions on the usage of sessions (see [18, 4, 13] for a showcase of such conditions). Conversely, contracts faithfully capture the behavior of processes and the temporal dependencies between communications occurring on different channels. This significantly widens the spectrum of systems that are declared well formed.

Overview. We develop our theory of contracts in §2 and put it at work on a core session calculus in §3, where we encode various examples, most of which from existing bibliography. §4 discusses differences between our approach and those with session types, presents more related work and hints at future research. For space reasons proofs, alternative characterizations, decidability, and deduction systems for the relations defined in §2 are omitted and can be found in the extended version available online.

2 Contracts

The syntax of contracts makes use of an infinite set $\mathscr{N}$ of *channel names* ranged over by $a, b, c, \ldots$ and of an infinite set $\mathscr{X}$ of *channel variables* ranged over by $x, y, z, \ldots$; let $\mathscr{V}$ be the set of *basic values* ranged over by $v, u, \ldots$; we let $\alpha, \beta, \ldots$ range over elements of $\mathscr{N} \cup \mathscr{X}$ and $m, n, \ldots$ range over *messages*, namely elements of $\mathscr{N} \cup \mathscr{V}$. Contracts, ranged over by $T, S, \ldots$, action prefixes, ranged over by $\pi, \pi', \ldots$, and patterns, ranged over by $f, g, \ldots$ are defined by the rules below:

$$
\begin{array}{rcl}
T & ::= & \mathbf{0} \mid \mathbf{1} \mid \Omega \mid \pi.T \mid T + T \mid T \oplus T \mid T \,|\, T \mid (\nu a)T \\
\pi & ::= & \alpha?f \mid \alpha!f \mid \alpha!(a) \\
f & ::= & \mathbb{0} \mid x \mid m \mid \mathtt{B} \mid (x) \mid f \vee f \mid f \wedge f \mid \neg f
\end{array}
$$

The syntax of patterns makes use of a fixed set of basic types ranged over by $\mathtt{B}$ such as $\mathtt{Int}, \mathtt{Bool}, \mathtt{Real}, \ldots$. Patterns describe the (possibly infinite) set of messages that are sent or received by an action occurring in a contract. Their semantics is fairly standard and *pattern matching* rules, described below, derive matching relations of the form $m \in f \rightsquigarrow \sigma$ where $\sigma : \mathscr{X} \to \mathscr{N}$ is a substitution whose domain $\mathtt{dom}(\sigma)$ is always finite:

$$m \in m \rightsquigarrow \emptyset \qquad a \in (x) \rightsquigarrow \{a/x\}$$

$$\frac{v : \mathtt{B}}{v \in \mathtt{B} \rightsquigarrow \emptyset} \qquad \frac{m \in f \rightsquigarrow \sigma}{m \in f \vee g \rightsquigarrow \sigma} \qquad \frac{m \in f \rightsquigarrow \sigma \qquad m \in g \rightsquigarrow \sigma'}{m \in f \wedge g \rightsquigarrow \sigma\sigma'} \qquad \frac{m \notin f}{m \in \neg f \rightsquigarrow \emptyset}$$

The empty pattern $\mathbb{0}$ is the one that matches no message. The singleton pattern m matches the message m. The type pattern $\mathtt{B}$ matches all and only basic values of type $\mathtt{B}$. The variable pattern x represents a singleton pattern that is going to be instantiated (x must be bound at some outer scope), whereas the capture pattern (x) binds the variable x and matches every channel. The disjunction, conjunction, and negation patterns implement the standard boolean operators and must obey the standard syntactic constraints: the two branches of a disjunction must bind the same variables and the two branches of a conjunction must bind disjoint variables. We write $\mathtt{fv}(f)$ and $\mathtt{bv}(f)$ for the sets of free and bound variables occurring in a pattern. Their definition is standard, here we just recall that $\mathtt{bv}(\neg f) = \emptyset$ regardless of f. Also, we write $f \setminus g$ for $f \wedge \neg g$. Let f be a closed pattern, namely a pattern such that $\mathtt{fv}(f) = \emptyset$; the semantics of f, notation $[\![f]\!]$, is the set of messages that are matched by f, that is $[\![f]\!] = \{m \mid \exists \sigma : m \in f \rightsquigarrow \sigma\}$. We impose an additional constraint on patterns occurring in output actions, which we call output patterns from now on. An output pattern f is *valid* if it matches a *finite* number of names, that is $[\![f]\!] \cap \mathscr{N}$ is finite. Basically this means that a process cannot "guess" channel names. If it sends a channel, then either it is a public channel, or a channel it has received earlier, or it is a fresh channel (pattern validity is formalized in the long version of the paper).

Contracts include three terminal behaviors: $\mathbf{0}$ is the behavior of a deadlocked process that executes no further action; $\mathbf{1}$ is the behavior of the successfully terminated process; Ω is the behavior of a process that can autonomously evolve without any further interaction with the environment. A contract $\pi.T$ describes a process that executes an action π and then behaves according to T. There are three kinds of actions: an input action $\alpha?f$ describes the ability to input any message that matches f on the channel α; output actions $\alpha!f$—i.e., *free* outputs—describe a similar output capability. Actions of the form $\alpha!(a)$—i.e., *bound* outputs—describe the creation and extrusion of a fresh, private channel a, namely the establishment of a connection between the sender of the message and the receiver of the message. The contracts $T + S$ and $T \oplus S$ respectively describe the external and internal choices between the behaviors described by T and S. In an external choice, the process will behave as either T or S according to the environment. In an internal choice, the process will internally and autonomously decide to behave as either T or S. We also include two static operators for combining contracts, but only as a convenient way of describing systems as terms $(\nu a_1) \cdots (\nu a_n)(T_1 \mid T_2 \mid \cdots \mid T_m)$ where T_i is a description of the i-th participant, which is essentially a sequential process, the a_i's are the private channels used by the participants to communicate with each other, and the participants execute in parallel. It can be shown that restriction and

parallel composition are redundant and their effects can be expressed by suitable combinations of actions (including bound output actions) and the two choice operators.

To cope with infinite behaviors we use a technique we introduced in [14] and allow contracts (but not patterns) to be infinite, provided that they satisfy the following conditions: (1) their syntax tree must be regular, namely it must be made of a finite number of different subtrees; (2) the syntax tree must contain a finite number of occurrences of the parallel composition operator and of the restriction operator. The reason of such a choice is that we want to account for infinite behaviors in a syntax-independent way. In process algebra literature infinite behaviours are obtained by adding syntax for defining recursive processes: usually these appear in the form of recursively defined processes $\texttt{rec}\ X.T$, or of starred processes T^*, or of a separate set of mutually recursive declarations. These are nothing but finite representations of the infinite (syntax) trees obtained by unfolding or expanding them. We believe that it is far simpler and enlightening to get rid of syntactic constraints and work directly on infinite trees by relying on the theory developed by Bruno Courcelle [16]. In our theory we do not consider every possible infinite tree but just those that satisfy the two conditions above. The first condition – regularity – powers down the expressiveness of the language to include only and all contracts that can be generated by the `rec`-expressions or mutually recursive declarations customary in the process algebra literature (it is well-known that in a nondeterministic setting the $*$ operator is not as expressive as recursive definitions). Nevertheless all results stated in this work hold also for non regular trees, except the decidability ones of course. The second condition, which requires that only finitely many restrictions and parallel compositions occur in a contract, ensures that every contract describes a finite-state system (and, incidentally, keeps the whole theory decidable). Once the results are established for infinite trees, then it is straightforward to transpose them to any concrete syntax chosen and verify the consequences of this choice (see [14, 13] for a more detailed discussion on similar restrictions).

We give contracts the labeled operational semantics defined by the rules below, plus the symmetric of the rules for the binary operators. Labels are generated by the grammar $\mu ::= \checkmark \mid c?m \mid c!m \mid c!(a)$ and we use standard definitions $\texttt{bn}(\cdot)$ and $\texttt{fn}(\cdot)$ of bound and free names for labels and contracts:

$$\Omega \longrightarrow \Omega \qquad \mathbf{1} \xrightarrow{\checkmark} \mathbf{1} \qquad T \oplus S \longrightarrow T \qquad c!m.T \xrightarrow{c!m} T \qquad \frac{m \in f \rightsquigarrow \emptyset \qquad m \neq f}{c!f.T \longrightarrow c!m.T}$$

$$\frac{a \neq c}{c!(a).T \xrightarrow{c!(a)} T} \qquad \frac{m \in f \rightsquigarrow \sigma}{c?f.T \xrightarrow{c?m} T\sigma} \qquad \frac{T \xrightarrow{\mu} T'}{T+S \xrightarrow{\mu} T'} \qquad \frac{T \longrightarrow T'}{T+S \longrightarrow T'+S}$$

$$\frac{T \longrightarrow T'}{T \mid S \longrightarrow T' \mid S} \qquad \frac{T \xrightarrow{\mu} T' \qquad \mu \neq \checkmark \qquad \texttt{bn}(\mu) \cap \texttt{fn}(S) = \emptyset}{T \mid S \xrightarrow{\mu} T' \mid S} \qquad \frac{T \xrightarrow{\checkmark} T' \qquad S \xrightarrow{\checkmark} S'}{T \mid S \xrightarrow{\checkmark} T' \mid S'}$$

$$\frac{T \xrightarrow{c!m} T' \qquad S \xrightarrow{c?m} S'}{T \mid S \longrightarrow T' \mid S'} \qquad \frac{T \xrightarrow{c!(a)} T' \qquad S \xrightarrow{c?a} S' \qquad a \notin \texttt{fn}(S)}{T \mid S \longrightarrow (\nu a)(T' \mid S')}$$

$$\frac{T \xrightarrow{\mu} T' \qquad a \notin \texttt{fn}(\mu) \cup \texttt{bn}(\mu)}{(\nu a)T \xrightarrow{\mu} (\nu a)T'} \qquad \frac{T \xrightarrow{c!a} T' \qquad a \neq c}{(\nu a)T \xrightarrow{c!(a)} T'} \qquad \frac{T \longrightarrow T'}{(\nu a)T \longrightarrow (\nu a)T'}$$

The contract Ω provides an unbound number of internal transitions, while no transition stems from $\mathbf{0}$. The contract $\mathbf{1}$ emits a label $\checkmark$ denoting successful termination and reduces to itself. The internal choice $T \oplus S$ may silently reduce to either T or S. A contract $c!f.T$ first silently chooses one particular message m that matches f, and then emits it. The operational semantics requires that no name is captured (the substitution resulting from matching must be $\emptyset$) but this follows from validity of output patterns. A contract $c!(a).T$ emits a fresh ($a \neq c$) channel name a and reduces to T. A contract $c?f.T$ is capable of receiving any message m that matches f and reduces to T where all the captured channel names in m are substituted for the corresponding capture variables. We omit the precise definition of substitution, which is standard except that it applies also to free variables occurring in patterns. An external choice $T + S$ offers all visible actions that are offered by either T or S. Both the external choice and the parallel composition are preserved under internal choices of their component contracts. In particular, the rule for $+$ indicates that this operator is a truly external choice. Any visible action emitted by either T or S is also emitted by their parallel composition, provided that the action is different from $\checkmark$ and that no capture of free names occurs. The parallel composition of two contracts is successfully terminated only if both contracts are. Then we have the usual synchronization rules for parallel composition, where pairs of dual actions react and give rise to an internal action. Finally, visible transitions whose labels have names that differ from a restricted one are propagated outside restrictions, and so are invisible transitions. The output of a channel name becomes a connection whenever it escapes its restriction. In the following we write $T \not\longrightarrow$ (respectively $T \stackrel{\mu}{\not\longrightarrow}$) if there is no T' such that $T \longrightarrow T'$ (respectively $T \stackrel{\mu}{\longrightarrow} T'$); if $T \not\longrightarrow$, then we say that T is *stable*; we write $\Longrightarrow$ for the reflexive, transitive closure of $\longrightarrow$; we write $\stackrel{\mu}{\Longrightarrow}$ for $\Longrightarrow\stackrel{\mu}{\longrightarrow}\Longrightarrow$.

Remark 1. Our syntax is redundant insofar as $\mathbf{0}$ and Ω can be encoded respectively as the infinite processes (solution of the equations) $X = X + X$ and $X = X \oplus X$. Both are regular and finite state and, according to the LTS, the former cannot perform any transition while the latter can only perform internal transitions and rewrite to itself. ■

Remark 2. Patterns allow us to capture different flavors of the π-calculus: standard π-calculus is obtained by using variable and singleton patterns in outputs and capture patterns in inputs. Adding singleton patterns in inputs gives us the π-calculus with matching ($[x = y]T$ can be encoded as $(\nu c)(c!x \mid c?y.T)$ for c not free in T); negation adds mismatch ($([x \neq y]T \equiv (\nu c)(c!x \mid c?\neg y.T)$); the polyadic π-calculus is obtained by adding the product constructor to patterns. ■

Example 1. The contract

$$T_{\text{Seller}} \stackrel{\text{def}}{=} \texttt{store}?(x).x?\texttt{String}.(x!\texttt{error}.\mathbf{1} \oplus x!\texttt{Int}.(x?\texttt{ok}.\texttt{ship}!x.\mathbf{1} + x?\texttt{quit}.\mathbf{1}))$$

describes the behavior of a "seller" process that accepts connections on a public channel `store`. The conversation continues on the private channel x sent by the "buyer": the seller waits for the name of an item, and it sends back either the value `error` indicating that the item is not in the catalog, or the price of the item. In this case the seller waits for a decision from the buyer. If the buyer answers `ok`, the seller delegates some "shipper" process to conclude the transaction, by sending it the private channel x via the public channel `ship`. If the buyer answers `quit`, the transaction terminates immediately. ■

We now formalize the notion of *well-formed system* and we do so in two steps. First we characterize the compliance of a component T with respect to another component S, namely the fact that the component T is capable of correct termination if composed with another component that behaves according to S; then, we say that a system is well formed if every component of the system is compliant with the rest of the system.

Definition 1 (compliance). *Let $\#$ be the least symmetric relation such that $\checkmark \# \checkmark$, $c!m\# c?m$, and $c!(a) \# c?a$. Let $[T \mid S]$ stand for the system $T \mid S$ possibly enclosed within restrictions. We say that T is* compliant with *S, notation $T \lhd S$, if $T \mid S \Longrightarrow [T' \mid S']$ and $T' \not\longrightarrow$ implies that there exist μ_1 and μ_2 such that $\mu_1 \# \mu_2$ and $T' \xrightarrow{\mu_1}$ and $S' \overset{\mu_2}{\Longrightarrow}$.*

According to the definition, T is compliant with S if *every* computation of $T \mid S$ leading to a state where the residual of T is stable is such that either the residual of T has successfully terminated and the residual of S will eventually terminate successfully, or the two residuals can eventually synchronize. The transition labeled by μ_2 is weak because the synchronization may only be available after some time. Notwithstanding this, the availability of μ_2 is guaranteed because compliance quantifies over *every* possible computation. Observe that Ω is compliant with every S (but not viceversa). This is obvious, since Ω denotes indefinite progress without any support from the environment.

Compliance roughly corresponds to the notion of "passing a test" in the classical testing framework [20, 5, 26]: T compliant with S is like saying that S *must* pass the test T. There is an important difference though: in the classical framework, divergence is a catastrophic event that may prevent the test from reaching a successful state. This is sometimes justified as the fact that a diverging component may eat up all the computational power of a system, making the rest of the system starve for progress. In a setting where processes run on different machines/cores this justification is not applicable. Actually, divergence turns out to characterize good systems, those that do have progress. In particular, divergence of the test T is ignored, since it implies that T is making progress autonomously; divergence of the contract S being tested is ignored, in the sense that all the visible actions it provides are guaranteed. For example, we have $c?a.\mathbf{1} \lhd c!a.\mathbf{1} + \Omega$. In this sense, $+$ is a "strongly external" choice (if compared to the external choice in [20, 26]) since it guarantees the visible actions in the converging branches.

Definition 2 (well-formed system). *Let $\prod_{i\in\{1,\ldots,n\}} T_i$ stand for the system $T_1 \mid \cdots \mid T_n$, where $\prod_{i\in\emptyset} = \mathbf{1}$ by definition. Let $S \equiv \prod_{i\in\{1,\ldots,n\}} T_i$. We say that the system S is* well formed *if $T_k \lhd \prod_{i\in\{1,\ldots,n\}\setminus\{k\}} T_i$ for every $1 \leq k \leq n$. We say that T and S are* dual*, notation $T \bowtie S$, if $T \mid S$ is a well-formed system.*

For example, we have that $c?a.\mathbf{1} + c?b.\mathbf{1} \mid c!a.\mathbf{1} \oplus c!b.\mathbf{1}$ is well formed but $c?a.\mathbf{1} \mid c!a.\mathbf{1} \oplus c!b.\mathbf{1}$ is not. Similarly, $c!\mathtt{Int}.\mathbf{1} \mid c?\mathtt{Int}.\mathbf{1} + c?\neg\mathtt{Int}.\mathbf{0}$ is well formed but $c!\mathtt{Int} \vee a.\mathbf{1} \mid c?\mathtt{Int}.\mathbf{1} + c?\neg\mathtt{Int}.\mathbf{0}$ is not because $c!\mathtt{Int} \vee a.\mathbf{1} \mid c?\mathtt{Int}.\mathbf{1} + c?\neg\mathtt{Int}.\mathbf{0} \Longrightarrow \mathbf{1} \mid \mathbf{0}$. The systems $\mathbf{1}$ and Ω are trivially well formed: the former has terminated, and hence is compliant with the rest of the system, which is empty and consequently terminated as well; the latter is compliant with every system, and thus with the empty system as well.

Duality is the symmetric version of compliance, namely it considers the success of the test as well as of the contract being tested. Technically this corresponds to restricting the set of tests (or observers) of a contract T to those contracts that not only are satisfied by T but that additionally satisfy T.

Example 2. The contracts

$$T_{\text{Buyer}} \stackrel{\text{def}}{=} \texttt{store!}(c).c!\texttt{String}.(c?\texttt{error}.\mathbf{1} + c?\texttt{Int}.c!\texttt{ok}.c!\texttt{String}.\mathbf{1})$$
$$T_{\text{Shipper}} \stackrel{\text{def}}{=} \texttt{ship?}(x).x?\texttt{String}.\mathbf{1}$$

describe the behavior of a client that is always willing to buy the requested item regardless of its price, and of a shipper that asks the buyer's address before terminating successfully. The system $T_{\text{Buyer}} \mid T_{\text{Shipper}} \mid T_{\text{Seller}}$ is well formed since the only maximal computation starting from these contracts ends up in $\mathbf{1} \mid \mathbf{1} \mid \mathbf{1}$. ■

Well-formedness is a property of whole systems, but it is of little help when we want to reason about the single components of a system. For example, the contract Ω by itself is a well-formed system and any system in which one of its components has contract $\mathbf{0}$ is clearly ill formed. The contract T_{Buyer} in Example 2 is not a well-formed system by itself, yet it is very different from $\mathbf{0}$: there exist components that, combined with T_{Buyer}, make a well-formed system, while this is impossible for $\mathbf{0}$. In this sense we say that T_{Buyer} is *viable* and $\mathbf{0}$ is not.

Definition 3 (viability). *We say that T is* viable*, notation $T^{\bowtie}$, if $T \bowtie S$ for some S.*

Our notion of viability roughly corresponds to the notion of *well-typed process* in theories of session types. There is a fundamental difference though: a locally well-typed process (in the sense of session types) yields a well-typed system when completed by *any* context that is itself well-typed, whereas for a contract to be viable it suffices to find *one particular* context that can yield a well-formed system. This intuition suggests that well-typedness of a process is a much stricter requirement than viability, and explains why the guarantee of global properties such as system well-formedness comes at the cost of imposing severe constraints on the behavior of the single components. As an example that shows the difference between well-typedness and viability, consider the contracts $T \stackrel{\text{def}}{=} c?a.\mathbf{1} + c?b.\mathbf{0}$ and $S \stackrel{\text{def}}{=} c?a.\mathbf{0} + c?b.\mathbf{1}$. Both are viable when taken in isolation, but their composition is not: an hypothetical third component interacting with $T \mid S$ cannot send $c!b$ because T might read it and reduce to $\mathbf{0}$, and symmetrically it cannot send $c!a$ because S might read it and reduce to $\mathbf{0}$.

Duality induces a semantic notion of equivalence between contracts. Informally, two contracts are equivalent if they have the same duals.

Definition 4 (subcontract). *We say that T is a* subcontract *of S, notation $T \preceq S$, if $T \bowtie R$ implies $S \bowtie R$ for every R. Let $\approx$ be the equivalence relation induced by $\preceq$.*

For example $T \oplus S \preceq T$, namely it is safe to replace a process with a more deterministic one. Then we have $T \preceq \mathbf{1} + T$ and $T \preceq \Omega + T$, namely it is safe to replace a process with another one that, in addition to exposing the behavior of the original process, is also able to immediately terminate with success or is immediately able to make autonomous progress. Observe that $\mathbf{0}$, $\mathbf{1}$, and Ω are pairwise different: $\mathbf{0}$ is the least element of $\preceq$ and $T \approx \mathbf{0}$ means that there is no context that can guarantee progress to T, hence the process with contract T is ill-typed; $\mathbf{1} \not\approx \Omega$ because $\mathbf{1} \bowtie \mathbf{1}$ but $\mathbf{1} \not\bowtie \Omega$. Indeed $\mathbf{1}$ denotes eventual termination, while Ω denotes indefinite progress.

Just as for the π-calculus, it is easy to see that $\preceq$ is a precongruence for action prefixes without bound variables, and we have that $T\sigma \preceq S\sigma$ for every σ such that $\mathtt{dom}(\sigma) = \mathtt{bv}(f)$ implies $c?f.T \preceq c?f.S$. Additionally, $\preceq$ is a precongruence with respect to $\oplus$: it suffices to observe that $R \bowtie T \oplus S$ if and only if $R \bowtie T$ and $R \bowtie S$. It is equally easy to see however that $\preceq$ is not a precongruence with respect to $+$, because of the relation $\mathbf{0} \preceq T$. For example $\mathbf{0} \preceq c?a.\mathbf{0}$, but $c?a.\mathbf{1} + \mathbf{0} \not\preceq c?a.\mathbf{1} + c?a.\mathbf{0}$. This makes it difficult to axiomatize $\preceq$, since it is not possible to replace equals for equals in arbitrary contexts. Furthermore, the usefulness of the relation $\mathbf{0} \preceq T$ is questionable, since it allows the replacement of a deadlocked process with anything else. But if the process is already deadlocked, for sure the system it lives in is ill-formed from the start and it makes little sense to require that the system behaves well after the upgrade. Thus, we will also consider the closure of $\preceq$ with respect to external choice.

Definition 5 (strong subcontract relation). *Let $\sqsubseteq$ be the largest relation included in $\preceq$ that is a precongruence with respect to $+$, namely $T \sqsubseteq S \stackrel{\text{def}}{\iff} T + R \preceq S + R$ for every R. We write $\simeq$ for the equivalence relation induced by $\sqsubseteq$.*

Unlike $\preceq$, we have $\mathbf{0} \not\sqsubseteq T$ in general, but $\pi.\mathbf{0} \sqsubseteq \pi.T$ does hold. In fact, it is interesting to investigate whether the loss of the law $\mathbf{0} \preceq T$ in particular, and the use of $\sqsubseteq$ instead of $\preceq$ in general, have any significant impact on the theory. The following result shows that this is not the case:

Theorem 1. *If $T^{\bowtie}$ and $T \preceq S$, then $T \sqsubseteq S$.*

Namely, $\preceq$ and $\sqsubseteq$ *may* differ only when the $\preceq$-smaller contract is not viable. In practice, the use of $\sqsubseteq$ over $\preceq$ has no impact: the relation $T \sqsubseteq \mathbf{0}$ completely characterizes non-viable contracts and upgrading, specialization, and searching based on the subcontract relation make sense only when the smaller contract is viable. In fact, $\sqsubseteq$ allows us to reason on the properties of a process by means of its contract:

Proposition 1. *The following properties hold: (1) $T^{\bowtie}$ iff $T \not\simeq \mathbf{0}$; (2) $\mathbf{1} + T \sqsubseteq T$ iff $T \Longrightarrow T'$ implies $T' \stackrel{\checkmark}{\Longrightarrow}$; (3) $\Omega + T \sqsubseteq T$ iff $T \Longrightarrow T'$ implies $T' \longrightarrow$.*

Item (1) characterizes viable contracts, and hence describe processes that are well typed. Item (2) characterizes those contracts that, when reaching a stable state, are in a successful state; consequently, the property gives us a sufficient (but not necessary) condition for well-formedness. Item (3) characterizes those contracts that never reach a stable state, and hence describe processes that are capable of making indefinite progress.

3 Typing a core language of sessions

One possible way to assess the expressiveness of the contract language would be to provide a contract-based type system for one of the latest session type calculi in the literature. Unfortunately, this would spoil the simplicity of our approach: existing session type calculi are of increasing complexity and include ad-hoc operators designed to adapt session types to new usage scenarios. Typing such operators with our types would require twisted encodings, bringing back that very kind of complexity that our contracts

aim to avoid. For these reasons we prefer to introduce yet another session core calculus, use it to encode examples defined in other papers to motivate the introduction of restrictions and specialized constructs, and finally show that our types allow us to prove progress for these examples (and for more complex ones that escape current session type systems) without resorting to ad-hoc linguistic constructions.

The calculus we propose here—just as a proof-of-concept, not as an object of study—is a streamlined version of several calculi introduced in the literature (eg, [4, 18, 21]) and is defined by the following productions, where t is either `Bool` or `Int` and e ranges over unspecified expressions on these types:

$$\begin{array}{rcl} P & ::= & \mathtt{0} \mid \alpha!(a).P \mid \alpha!\langle e\rangle.P \mid \alpha?(x:t).P \mid \alpha!(\!|\alpha|\!).P \mid \alpha?(\!|x|\!).P \\ & \mid & \alpha \triangleleft \ell.P \mid \alpha \triangleright \{\ell_i : P_i\}_{i\in I} \mid P \mid P \mid (\nu a)P \mid \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ P \end{array}$$

For a reader knowledgeable of session types literature the syntax above needs no explanation. It boils down to a π-calculus that explicitly differentiates (channel) names, ranged over by $a, b, c, \ldots$ from variables, ranged over by $x, y, z, \ldots$ (only the former can be restricted, only the latter can be abstracted) and enriched with specific constructions for sessions, namely actions $\alpha!(\!|\alpha|\!)$ and $\alpha?(\!|x|\!)$ for session delegation[3] and actions $\alpha \triangleleft \ell$ and $\alpha \triangleright \{\ell_i : P_i\}_{i\in I}$ for label-based session branching. The calculus also includes both bound and free outputs, the former being used for session connection, the latter for communication. Infinite behaviour is obtained by considering terms coinductively generated by the productions, in the same way as we did for contracts and with the same restrictions.The semantics of the calculus is defined by an LTS whose most important rules are (see the online extended version for all rules):

$$c!(\!|a|\!).P \xrightarrow{c!a} P \qquad c?(\!|x|\!).P \xrightarrow{c?a} P\{^a/_x\} \qquad c \triangleleft \ell.P \xrightarrow{c!\ell} P$$

$$\frac{a \neq c}{c!(a).P \xrightarrow{c!(a)} P} \qquad \frac{e \downarrow v}{c!\langle e\rangle.P \xrightarrow{c!v} P} \qquad \frac{v : t}{c?(x:t).P \xrightarrow{c?v} P\{^v/_x\}} \qquad \frac{k \in I}{c \triangleright \{\ell_i : P_i\}_{i\in I} \xrightarrow{c?\ell_k} P_k}$$

The typing relation for the process calculus is coinductively defined by the following rules (where $\mathtt{Ch} \equiv \neg\neg(x)$ is the type of all channel names and the various ℓ's are reserved names that cannot be restricted or appear as subject of a communication).

$$\text{END}\ \ \Gamma \vdash \mathtt{0} : \mathbf{1} \qquad \text{NAME}\ \ \Gamma \vdash a : \mathtt{Ch} \qquad \text{VAR}\ \ \Gamma \vdash x : \Gamma(x)$$

$$\text{F-OUTPUT}\ \frac{\Gamma \vdash \alpha : \mathtt{Ch} \qquad \Gamma \vdash e : t \qquad \Gamma \vdash P : T}{\Gamma \vdash \alpha!e.P : \alpha!t.T} \qquad \text{INPUT}\ \frac{\Gamma \vdash \alpha : \mathtt{Ch} \qquad \Gamma, x : t \vdash P : T}{\Gamma \vdash \alpha?(x:t).P : \alpha?t.T}$$

$$\text{B-OUTPUT}\ \frac{\Gamma \vdash \alpha : \mathtt{Ch} \qquad \Gamma \vdash P : T}{\Gamma \vdash \alpha!(a).P : \alpha!(a).T} \qquad \text{C-SEND}\ \frac{\Gamma \vdash \alpha, \beta : \mathtt{Ch} \qquad \Gamma \vdash P : T}{\Gamma \vdash \alpha!(\!|\beta|\!).P : \alpha!\beta.T} \qquad \text{C-RECV}\ \frac{\Gamma \vdash \alpha : \mathtt{Ch} \qquad \Gamma, x : \mathtt{Ch} \vdash P : T}{\Gamma \vdash \alpha?(\!|x|\!).P : \alpha?(x).T}$$

[3] In our case one should rather speak of "session forwarding" or "session sharing" since, contrarily to session type systems, we do not enforce channel bi-linearity.

$$\text{CHOICE}\ \frac{\Gamma \vdash \alpha : \mathtt{Ch} \qquad \Gamma \vdash P : T}{\Gamma \vdash \alpha \triangleleft \ell.P : \alpha!\ell.T} \qquad \text{BRANCH}\ \frac{\Gamma \vdash \alpha : \mathtt{Ch} \qquad \Gamma \vdash P_i : T_i}{\Gamma \vdash \alpha \triangleright \{\ell_i : P_i\}_{i\in I} : \sum_{i\in I} \alpha?\ell_i.T_i}$$

$$\text{NEW}\ \frac{\Gamma \vdash P : T}{\Gamma \vdash (\nu a)P : (\nu a)T} \qquad \text{PAR}\ \frac{\Gamma \vdash P : T \qquad \Gamma \vdash Q : S}{\Gamma \vdash P \mid Q : T \mid S} \qquad \text{IF}\ \frac{\Gamma \vdash e : \mathtt{Bool} \quad \Gamma \vdash P : T \quad \Gamma \vdash Q : S}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q : T \oplus S}$$

The rules are straightforward since they perform a very simple abstraction on the content of communications. Although this typing permits a quite liberal usage of channels, it is sufficient to verify the progress property, as stated by the following theorem:

Theorem 2 (progress). *We say that a process* has succeeded *whenever it contains no action. If* $\vdash P : T$ *and* $\mathbf{1} + T \sqsubseteq T$ *and* $P \stackrel{\tau}{\Longrightarrow} Q \stackrel{\tau}{\not\longrightarrow}$*, then* Q *has succeeded.*

In words, for any residual Q of a process whose contract T satisfies $\mathbf{1} + T \sqsubseteq T$, if Q cannot make further progress ($Q \stackrel{\tau}{\not\longrightarrow}$), then it contains no further actions (i.e., it is a possibly restricted parallel composition of null processes $\mathtt{0}$) that is to say that all its components have successfully terminated.

We devote the rest of the section to the encoding of a few examples in our process language, some are taken from existing bibliography, others are new. The motivating example of [18] is (in our syntax) the following pair of processes

$$P = a?(\!|x|\!).b?(\!|y|\!).x!3.x?(z : \mathtt{Int}).y!\mathtt{true}.\mathtt{0}$$
$$Q = a!(c).b!(d).c?(z : \mathtt{Int}).d?(z' : \mathtt{Bool}).c!5.\mathtt{0}$$

The two processes initiate two sessions on (public) channels a and b and they exchange two integers over the former and a boolean value over the latter. The parallel composition of these two processes deadlocks. This composition type-checks in simple session type theories that verify the order of messages in each single session. The system in [18] ensures progress and thus rejects this composition. Progress is enforced by establishing a partial order on sessions and requiring that the usage of session channels respects this order. So in [18] the process P can be composed with $Q' = a!(c).b!(d).c?(z : \mathtt{Int}).c!5.d?(z' : \mathtt{Bool}).\mathtt{0}$ since in both processes the communication on b follows all communications on a. In our system the two processes have respectively type:

$$T_P = a?(x).b?(y).x!\mathtt{Int}.x?\mathtt{Int}.y!\mathtt{Bool}.\mathbf{1} \qquad T_Q = a!(c).b!(d).c?\mathtt{Int}.d?\mathtt{Bool}.c!\mathtt{Int}.\mathbf{1}$$

We have $T_P \mid T_Q \sqsubseteq \mathbf{0}$, hence the composition does not satisfy progress. In fact we can tell something more: $T_P \mid T_Q$ is not viable, namely there is no process, that composed with P and Q, allows us to obtain a well-formed system. As in [18] our system detects that $P \mid Q'$ is well formed, since $\mathbf{1} + (T_P \mid T_{Q'}) \sqsubseteq T_P \mid T_{Q'}$ (we let the reader figure out $T_{Q'}$). Furthermore while the system in [18] rejects the composition of Q with $P' = a?(\!|x|\!).b?(\!|y|\!).x!3.y!\mathtt{true}.x?(z : \mathtt{Int}).\mathtt{0}$, since it is not possible to order the actions of the two sessions, our system instead proves progress for this composition too.

Other restrictions introduced for session types to enforce progress are useless in our framework. In particular, we do not require linearity of session channels: a process can still use a channel that it has "delegated" without necessarily jeopardizing progress.

Consider the following two processes $P' = a?(\!|x|\!).b?(\!|y|\!).x!(\!|y|\!).x?(z:\mathtt{Int}).y!\mathtt{true}.0$ and $Q' = a!(c).b!(d).c?(\!|z|\!).c!5.z?(z':\mathtt{Bool}).0$. Although P' uses y after having delegated it, the contract T of $P' \mid Q'$ satisfies $\mathbf{1} + T \sqsubseteq T$. This freedom from linearity constraints instantly enables multi-party sessions, without the need of any dedicated syntax. Consider for instance the "Two Buyers" protocol of [22], which describes a conversation between a seller and two buyers, the first buyer being the initiator of the multi-party session. It can be rendered in our language as follows

$$\begin{aligned}
\text{Seller} &= a?(\!|x|\!).x?(\mathit{title}:\mathtt{String}).x!\langle \mathit{quote}\rangle.x!\langle \mathit{quote}\rangle. \\
&\qquad x \triangleright \{\mathtt{ok} : x?(\mathit{addr}:\mathtt{String}).x!\langle \mathit{date}\rangle.0,\ \mathtt{quit} : 0\} \\
\text{Buyer1} &= a!(c).a!(\!|c|\!).c!\langle \text{“War and Peace”}\rangle.c?(\mathit{quote}:\mathtt{Int}).b!(d).d!\langle \mathit{quote}/2\rangle.0 \\
\text{Buyer2} &= a?(\!|x|\!).x?(\mathit{quote}:\mathtt{Int}).b?(\!|y|\!).y?(\mathit{contrib}:\mathtt{Int}).\mathtt{if}\ \mathit{quote}-\mathit{contrib} \leq 99 \\
&\qquad \mathtt{then}\ x \triangleleft \mathtt{ok}.x!\langle \mathit{address}\rangle.x?(d:\mathtt{Date}).0\ \mathtt{else}\ x \triangleleft \mathtt{quit}.0
\end{aligned}$$

Buyer1 initiates the session on the public channel a by broadcasting twice the session channel c. This channel is received by Seller and Buyer2, used by Buyer1 to request a title to Seller, and used by Seller to send the price to both buyers and to conclude the transaction with Buyer2. As for [22] the communication between the two buyers takes place on a separate private channel y initiated on the public channel b. The three agents are really the same as the corresponding ones in [22] except that (i) connections are dyadic and do not have to explicitly state the name of the intended partner and (ii) Seller is not aware of the private channel used by the buyers to communicate together. As [22] our type system: (1) it ensures that the composition of the above agents is well typed since their contracts are

$$\begin{aligned}
\text{Seller} &: a?(x).x?\mathtt{String}.x!\mathtt{Int}.x!\mathtt{Int}.(x?\mathtt{ok}.x?\mathtt{String}.x!\mathtt{Date}.\mathbf{1} + x?\mathtt{quit}.\mathbf{1}) \\
\text{Buyer1} &: a!(c).a!(\!|c|\!).c!\mathtt{String}.c?\mathtt{Int}.b!(d).d!\mathtt{Int}.\mathbf{1} \\
\text{Buyer2} &: a?(x).x?\mathtt{Int}.b?(y).y?\mathtt{Int}.(x!\mathtt{ok}.x!\mathtt{String}.x?\mathtt{Date}.\mathbf{1} \oplus x!\mathtt{quit}.\mathbf{1})
\end{aligned}$$

and the parallel composition T of these contracts satisfies the law $\mathbf{1} + T \sqsubseteq T$ and (2) it would reject the protocol if the channel c were also used for the inter-buyer communication. Of course the property $\mathbf{1} + T \sqsubseteq T$ ensures that the system formed by the three agents has also progress, as does the system of [4] which extends [22] with progress.

The full expressive power of contracts emerges when specifying recursive protocols. To illustrate the point we encode a well-known variant of the Diffie-Hellman protocol, the Authenticated Group Key Agreement protocol A-GDH.2, defined in [2]. This protocol allows a group of n processes $P_1 \ldots P_n$ to share a common authenticated key s_n. The protocol assumes the existence of n channels $c_1 \ldots c_n$ each c_i being shared between P_{i-1} and P_i (for the sake of the simplicity we use a bootstrapping process P_0 absent in the original presentation). The original algorithm also assumes that the process P_n shares a secret key K_i with process P_i for $i \in [0,n)$. We spice up the algorithm so that the sharing of these keys is implemented via a private channel that is delegated at each step along the participants:

$$\begin{aligned}
P_0 &= c^1!\langle \mathsf{d}\rangle.c^1?(\!|y|\!).y \triangleleft \mathtt{ok}.0 \\
P_i &= c^i?(x:\mathsf{data}).c^{i+1}!\langle f(x)\rangle.c^{i+1}?(\!|y|\!).y \triangleleft \mathtt{key}.y!\langle \mathsf{K}_i\rangle.y?(s:\mathtt{Int}).c^i!(\!|y|\!).0 \qquad i\in[1,n) \\
P_n &= c^n?(x:\mathsf{data}).c^n!(c).Q \qquad \text{with } Q = c \triangleright \{\mathtt{ok}:0\ ,\ \mathtt{key}:c?(k:\mathtt{Int}).c!\langle g(x,k)\rangle.Q\}
\end{aligned}$$

In a nutshell, P_0 starts the protocol by sending some data d to P_1; each intermediate process P_i receives some data x from P_{i-1}, uses this data to compute new data $f(x)$ that it forwards to P_{i+1}; the terminal process P_n receives the data x from P_{n-1}, generates a new private channel c used for retrieving from every process P_i the key K_i and sending back an integer $g(x,\mathsf{K}_i)$ (from which P_i can deduce the key $\mathsf{s_n}$). Each intermediate process delegates the channel y to its predecessor and P_0 notifies to P_n the successful termination of the protocol (refer to Figures 2 and 3 of [2] for precise definitions of f and g).

The algorithm is quite complex to analyse for at least two reasons: (i) P_n is recursively defined since it does not know *a priori* how many processes take part in the protocol (actually it statically knows just the existence of channel c^n), and (ii) all the shared keys are transmitted over the same channel c (generated in the second action of P_n) which constitutes a potential source of interference that may hinder progress.

All the session type systems that enforce progress we are aware of fail to type check the above protocol. In particular the progress type systems for dyadic sessions [18] and multi-party ones [4] fail to type P_i because, as long as the c^i are considered private channels, it performs an output on a channel c^i in the continuation of the reception of a delegation; also, the protocol ends by emitting $\mathtt{ok}$ on the channel y, but every P_i process uses c^i before and after a synchronization on y, so it is not possible to find a well-founded order on y and the various c^i's since there is a double alternation of actions on y and c^i. Likewise, conversation types [11] can type all the processes of the protocol but progress cannot be ensured because there is no well-founded order for channels. As a matter of facts, in all these works progress is enforced by the techniques introduced by Kobayashi [23, 24] where types are inhabited by channels on which a well-founded usage (capability/obligation, in Kobayashi's terminology) order can be established. By inhabiting types by processes we escape the need of such an order and thus can type the processes of the protocol.

Classical Sessions Typing. The type system presented earlier in the section is enough to ensure the progress property stated in Theorem 2. However, in some contexts it may be desirable to enforce a stricter typing discipline in order to impose a particular communication model. Let us clarify this point with an example. Consider the system $c?(x:\mathtt{Int}).0 \,|\, c?(x:\mathtt{Bool}).0 \,|\, c!\langle 3\rangle.0 \,|\, c!\langle \mathtt{true}\rangle.0$ which is composed of four processes, two of which are sending on the channel c messages with different types (`Int` and `Bool`) while the remaining processes are waiting for two messages on the channel c (of type `Int` and `Bool`). Its contract is $S \stackrel{\text{def}}{=} c?\mathtt{Int}.\mathbf{1} \,|\, c?\mathtt{Bool}.\mathbf{1} \,|\, c!\mathtt{Int}.\mathbf{1} \,|\, c!\mathtt{Bool}.\mathbf{1}$ which satisfies $\mathbf{1} + S \sqsubseteq S$. Namely, the above system is free from communication errors despite it contains processes that are sending and receiving messages of different types on the same channel at the same time. Technically this happens because the operational semantics of contracts (and of processes) does not distinguish between *communication* and *matching*. In practice, the typing rules we have given reflect a particular communication model: a receiver waits until a message *of the expected type* is available. Thus, a well-formed system is such that any process that is blocked waiting for a message will *eventually* read a message of the expected type, and any process that is blocked trying to send a message will *eventually* deliver it to someone who is able to handle it.

This communication model is not the only reasonable one, especially in a distributed setting where asynchrony decouples communication from the ability to inspect the con-

tent of messages. In this setting, it could be reasonable to assume that a process waiting for messages sent on some channel c should be ready to handle *any* message sent on that channel at that particular time. Somewhat surprisingly, this communication model can be implemented without any change to the operational semantics of contracts and processes, but merely adjusting a few typing rules, those that deal with input actions:

$$\text{INPUT}\quad \frac{\Gamma \vdash \alpha : \mathtt{Ch} \qquad \Gamma, x : t \vdash P : T}{\Gamma \vdash \alpha?(x:t).P : \alpha?t.T + \alpha?\neg t.\mathbf{0}} \qquad \text{C-RECV}\quad \frac{\Gamma \vdash \alpha : \mathtt{Ch} \qquad \Gamma, x : \mathtt{Ch} \vdash P : T}{\Gamma \vdash \alpha?(\!|x|\!).P : \alpha?(x).T + \alpha?\neg\mathtt{Ch}.\mathbf{0}}$$

$$\text{BRANCH}\quad \frac{\Gamma \vdash \alpha : \mathtt{Ch} \qquad \Gamma \vdash P_i : T_i}{\Gamma \vdash \alpha \triangleright \{\ell_i : P_i\}_{i\in I} : \sum_{i\in I} \alpha?\ell_i.T_i + \alpha?\bigwedge_{i\in I} \neg\ell_i.\mathbf{0}}$$

Intuitively, the contract of a process waiting for messages states that the process is capable of receiving *any* message, but only those of some particular type will allow the process to continue. If the process receives a message that it is not able to handle, the process deadlocks, therefore compromising well-formedness of the system it belongs to. With these typing rules in place, we are making the assumption that at each step of a computation any channel is implicitly associated with a unique type of its messages (i.e., a set of labels, a set of values, a finite set of channel names, or a combination of these in case the language provides for boolean operators over types) and that every process that at that step waits for messages on that channel must be able to handle *every* message that is or may be sent on it. With the modified typing rules, the above system is typed by the contract $S' \stackrel{\text{def}}{=} c?\mathtt{Int}.\mathbf{1} + c?\neg\mathtt{Int}.\mathbf{0} \mid c?\mathtt{Bool}.\mathbf{1} + c?\neg\mathtt{Bool}.\mathbf{0} \mid c!\mathtt{Int}.\mathbf{1} \mid c!\mathtt{Bool}.\mathbf{1}$ and it is immediate to see that this system no longer enjoys progress, since $S' \sqsubseteq \mathbf{0}$.

Interestingly, this modified typing discipline gives rise to the same subtyping relation as the one defined by Gay and Hole for session types [19]. In particular, we have contravariance for outputs, covariance for inputs, and width subtyping for branching. More precisely if we define the subtyping relation for patterns (and thus for types) as $f \le g \stackrel{\text{def}}{=} [\![f]\!] \subseteq [\![g]\!]$, then it is not difficult to verify the soundness of the rules

$$\frac{f \le g \qquad T \sqsubseteq S}{c!g.T \sqsubseteq c!f.S} \qquad \frac{f \le g \qquad T \sqsubseteq S}{c?f.T + c?\neg f.\mathbf{0} \sqsubseteq c?g.S + c?\neg g.\mathbf{0}}$$

The two rules state that the implicit type of a channel (in the sense of session types: we do not assign any type to channels) can be contravariantly specialized for emission actions (F-OUTPUT), (C-SEND), (CHOICE) and covariantly specialized for reception actions (INPUT), (C-RECV), (BRANCH). For instance, when in the system above we deduce for the process $c?(x:t).P$ the contract $c?t.T + c?\neg t.\mathbf{0}$ we are implicitly assuming that the channel c in the system of Gay and Hole would have type $?[t].T'$ for some T' (which would be the projection of T over c); the rule on the right hand side states that it can be safely replaced by a channel whose (implicit) type is $?[s].T'$, provided that $t \le s$.

The language proposed in this section is just an example of how our contracts can be used. But one can, and indeed should, imagine to use them to type advanced process calculi with, for instance, type driven synchronization (e.g., the language PiST for ses-

sions defined in [13]) or first-class processes (contracts being assigned to processes, a typed calculus with higher-order processes looks as a natural next step).

4 Related Work and Conclusions

The latest works on session types witness a general trend to use more and more informative types, a trend that makes these approaches closer to the techniques of specification refinement. Here we push this trend to an extreme. Contracts are behavioral types that accurately capture the behavior of participants in a conversation by providing a relatively shallow abstraction over processes that respect them. We are at the edge between behavioral types, specification refinements, and abstract interpretation: contracts record the flow of communications only when channels are passed around and they abstract communication content into patterns; values are abstracted into possibly infinite set of values (i.e., types) and names into finite sets of names (i.e., valid output patterns). Inasmuch as shallow this abstraction is, it is enough to define a theory of contracts (whose comparison with testing theories was discussed in Sections 1 and 2) that allows us to reason *effectively* (see the online extended version for decidability results) about the correctness of systems and about the safe substitutability and well-typedness of components of a system: it makes us switch from undecidability to decidability. A similar approach is followed by [9], where the content of messages can be made opaque and thus abstracted; this is closer to, though grosser than, the refinement approach. The crucial difference between our approach and all other theories of contracts, [9] included, is that we keep track of names passed around in communications.

We discussed technical differences between contracts and session types all the presentation long. The key point is that contracts record the overall behavior of a process, while session types project this behavior on the various private channels the process uses. Providing partial views of the behavior makes session types more readable and manageable. At the same time it hinders their use in enforcing global properties—notably, progress—whence the need for awkward restrictions such as channel linearity, controlled nesting, scarce session interleaving, and global order on channel usage. In practice, the two approaches have both pros and cons. The contract approach is "optimistic" in that a process is considered well typed as long as there exists at least one context that composed with it yields a well-formed system; session types, instead, account for the nastier possible context. Thus, the contract-based approach widens the spectrum of well-typed (viable) processes but, as we have seen, the composition of viable processes is not necessarily viable. This implies that to prove viability of a parallel composition of processes our approach requires a global system analysis that effectively enumerates all control-flow paths, whereas session types allow each process component to be verified independently. However, if a process is not viable, then it is easy to exhibit a trace of actions that leads the process into an error state by looking at its contract. The session-based approach restricts the set of well-typed processes, but makes it easy to compose them. However, if the type checker detects an ill-typed process, it may be unable to provide any sensible information to help the programmer fix the problem.

An interesting, related approach is the one of Conversation Types [11]. Conversation Types are close to contracts, inasmuch as types provide a global and unique description

of the processes involved in a composition. The difference resides mainly in the formalism used to describe the behavior: contracts stick as close as possible to the π-calculus, while Conversation Types draw their inspiration from the structural features of spatial logic [10] and Boxed Ambients [8] by organizing behavior around places of conversation (which thus generalize sessions) and describing communications relatively to them (local vs. external). As in our case the approach of Conversation Types is optimistic and does not require awkward usage conditions: for instance Conversation Types allow processes to still use a channel after having delegated it, as for contracts. Conversation Types do not ensure progress, which is instead enforced via an auxiliary deduction system that exploits an order relation on channels. In this respect Conversation Types seem more basic than contracts. The authors say that the Conversation Calculus they type can be seen as a π-calculus with labels and patterns, as our calculus is. It will be interesting to check whether/how contracts can type the conversation calculus and deduce an in-depth comparison of the two approaches. We leave this as future work.

A problem that is connected with but orthogonal to the ones studied here is the global specification of choreographies. Contracts are subjective descriptions of (components of) systems, but cannot be used to give a global specification of a choreography to which every acceptable implementation (decomposition) must conform. As a matter of facts, in our theory a closed, well-formed system is typed by either $\mathbf{1}$ or Ω. There exist two main proposals of languages for high-level specification of the structure of conversation within a choreography against which the components of the choreography must be validated. The first proposal, issued from the literature of session types, is based on the definition of "global types" that describe the structure of conversation by listing for each session the sender, receiver and content of each communication [22, 27]. The second proposal is directly embedded into conversation types, since they describe the global structure of the conversation by providing a set of traces in which internal transitions are labeled by the synchronization that generated them. Whether these two approaches fit our setting is matter of future research.

Alternative characterizations of viability and of the subcontract relations, as well as the proof system for $\sqsubseteq$, shed light on important aspects of the theory, yet we had to omit them because of the page limit: they can be found in the online extended version. We are currently extending the completeness proof of the deduction system to an algorithm for deciding $\sqsubseteq$. In this respect, the constraint of working with regular (hence finite-state) contracts plays a crucial role and may prove fundamental in comparing the expressive power of our theory with alternative theories that share common goals. The exact implications of this constraint are currently unclear and subject of in-depth investigations.

Acknowledgments. Work partially supported by the project ANR-08-EMER-004-04. We thank Mariangiola Dezani-Ciancaglini, Hugo Vieira, and Nobuko Yoshida for insightful discussions.

References

1. L. Aceto and M. Hennessy. Termination, deadlock, and divergence. *J. ACM*, 39(1), 1992.
2. G. Ateniese, M. Steiner, and G. Tsudik. Authenticated group key agreement and friends. In *CCS '98: 5th ACM conference on Computer and Communications Security*, 1998.
3. L. Bettini, S. Capecchi, M. Dezani-Ciancaglini, E. Giachino, and B. Venneri. Session and Union Types for Object Oriented Programming. LNCS 5065, 2008.

4. L. Bettini, M. Coppo, L. D'Antoni, M. De Luca, M. Dezani, and N. Yoshida. Global Progress in Dynamically Interleaved Multiparty Sessions. In *CONCUR '08*, LNCS 5201, 2008.
5. M. Boreale and R. De Nicola. Testing equivalence for mobile processes. *Inf. Comput.*, 120(2):279–303, 1995.
6. M. Bravetti and G. Zavattaro. Contract based multi-party service composition. In *FSEN*, LNCS 4767, pages 207–222. Springer, 2007.
7. M. Bravetti and G. Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In *6th Intl. Symp. on Software Composition*, LNCS 4829, 2007.
8. M. Bugliesi, G. Castagna, and S. Crafa. Access control for mobile agents: The Calculus of Boxed Ambients. *ACM TOPLAS*, 26(1):57–124, 2004.
9. M.G. Buscemi and H. Melgratti. Abstract processes in orchestration languages. In *Proc. of ESOP '09*, LNCS 5502, 2009.
10. L. Caires. Spatial-behavioral types for concurrency and resource control in distributed systems. *TCS*, 402(2-3):120–141, 2008.
11. L. Caires and H. Vieira. Conversation types. In *Proc. of ESOP '09*, LNCS 5502, 2009.
12. S. Carpineti, G. Castagna, C. Laneve, and L. Padovani. A formal account of contracts for Web Services. In *WSFM '06*, LNCS 4184. Springer, 2006.
13. G. Castagna, M. Dezani-Ciancaglini, E. Giachino, and L. Padovani. Foundation of session types. Unpublished manuscript (available on line), January 2009.
14. G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for Web Services. *ACM TOPLAS*, 31(5), 2009.
15. M. Coppo, M. Dezani-Ciancaglini, and N. Yoshida. Asynchronous Session Types and Progress for Object-Oriented Languages. In *FMOODS '07*, LNCS 4468. Springer, 2007.
16. B. Courcelle. Fundamental properties of infinite trees. *Theor. Comput. Sci.*, 25:95–169, 1983.
17. R. De Nicola and M. Hennessy. CCS without τ's. In *TAPSOFT/CAAP '87*, LNCS 249, pages 138–152. Springer, 1987.
18. M. Dezani-Ciancaglini, U. de' Liguoro, and N. Yoshida. On Progress for Structured Communications. In *TGC'07*, LNCS 4912, pages 257–275. Springer, 2008.
19. S. Gay and M. Hole. Subtyping for session types in the π-calculus. *Acta Inf.*, 42(2-3), 2005.
20. M. Hennessy. *Algebraic Theory of Processes*. Foundation of Computing. MIT Press, 1988.
21. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP '98*, LNCS 1382. Springer, 1998.
22. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL '08: the 35th annual ACM Symp. on Principles of Programming Languages*, 2008.
23. N. Kobayashi. A type system for lock-free processes. *Inf. Comput.*, 177(2):122–159, 2002.
24. N. Kobayashi. A new type system for deadlock-free processes. In *CONCUR '06*, LNCS 4137. Springer, 2006.
25. C. Laneve and L. Padovani. The *must* preorder revisited – An algebraic theory for web services contracts. In *Proc. of CONCUR '07*, LNCS 4703. Springer, 2007.
26. Hennessy M. A fully abstract denotational semantics for the pi-calculus. *Theor. Comput. Sci.*, 278:53–89(37), 2002.
27. D. Mostrous, N. Yoshida, and K. Honda. Global principal typing in partially commutative asynchronous sessions. In *Proc. of ESOP '09*, LNCS 5502, pages 316–332, 2009.
28. L. Padovani. Contract-directed synthesis of simple orchestrators. In *CONCUR '08: 19th Intl. Conf. on Concurrency Theory*, LNCS 5201, pages 131–146. Springer, 2008.
29. B.C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996.
30. V. Vasconcelos, S. Gay, and A. Ravara. Session types for functional multithreading. In *CONCUR '04*, LNCS 3170, pages 497–511. Springer, 2004.