Foundations of Behavioural Types

Hans Hüttel, Aalborg University
Ivan Lanese, University of Bologna/INRIA
Vasco T. Vasconcelos, University of Lisbon
Luis Caires, New University of Lisbon
Marco Carbone, ITU Copenhagen
Pierre-Malo Deniélou, Royal Holloway, University of London
Dimitris Mostrous, University of Lisbon
Luca Padovani, University of Turin
António Ravara, New University of Lisbon
Emilio Tuosto, University of Leicester
Hugo Torres Vieira, University of Lisbon
Gianluigi Zavattaro, University of Bologna/INRIA

Behavioural type systems, usually associated to concurrent or distributed computations, encompass concepts such as interfaces, communication protocols, contracts, and choreography, in addition to the traditional input/output operations. The behavioural type of a software component specifies its expected patterns of interaction using expressive type languages, so that types can be used to determine automatically whether the component interacts correctly with other components. The last twenty years have witnessed the fast growth of the field; this paper surveys the main accomplishments.

Categories and Subject Descriptors: F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs; D.3.3 [Programming Languages]: Language Constructs and Features; D.2.4 [Software Engineering]: Software/Program Verification

General Terms: Theory, Languages, Verification
Additional Key Words and Phrases: Behavioural types

ACM Reference Format:
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Types make it possible to classify entities of a program and to describe the permissible results of a computation. A type discipline can guarantee that well-typed programs are well-behaved. Traditionally the focus of the work on type systems has been on the outcome of computations, that is, on what the result of a computation should be.

During the 1990's, program semantics, in particular concurrency theory and especially the study of type disciplines for process calculi, has inspired notions of typing that are also able to describe properties associated with the behaviour of programs and in this way also describe how a computation proceeds. This often includes accounting for the notions of causality and choice. Type disciplines that describe these notions directly are often referred to as behavioural types.

There is no hard and fast line of demarcation between behavioural type systems and other type systems. The work on behavioural types arose in the context of type systems that capture properties of computations in process calculi. While these systems do not describe the behavioural information directly as part of the type language, some of them have been instrumental in the development of the behavioural type systems presented in the current paper, e.g., by introducing notions of separation between capabilities for names, of linear usage of names, and the case analysis of the variant types. We refer the interested reader to [Sangiorgi and Walker 2001; Kobayashi 2003].

ACM Computing Surveys, Vol. V, No. N, Article A, Publication date: January YYYY.
This paper provides a survey of most relevant works on the foundations of behavioural types.

The outline of the paper is as follows. Section 2 sketches the main approaches to behavioural types. The sections that follow describe the most important of these approaches in greater detail. Section 3 describes the notion of types for binary sessions. Section 4 describes notions of behavioural types that focus on multiparty interaction. Section 5 describes approaches to subtyping and polymorphism in the setting of behavioural types.

The remaining chapters describe the expressiveness and algorithmic properties of behavioural types. Section 6 outlines what is known about the relationship between behavioural types and logic. Section 7 addresses liveness properties using behavioural types. Section 8 describes how the various approaches to behavioural types interrelate. Finally, Section 9 deals with algorithmic properties related to behavioural types, including decidability results for typing and subtyping.

2. APPROACHES TO BEHAVIOURAL TYPES

This section briefly presents the various known approaches to behavioural types. In the following chapters we restrict our attention to the main trends still active today.

2.1. Types and effects

The first proposal of behavioural types for concurrency was, to our knowledge, made by Nielson and Nielson 1993 in the setting of the concurrent functional language Concurrent ML. The authors develop a type and effect discipline. A type and effect system makes it possible to statically describe the dynamic behaviour of a computation. The distinction is that the type describes what an expression will compute (sets of values), while the effect describes how an expression will compute (the behaviour). In this approach, type judgements for programs P are of the form \( \Gamma \vdash P : T \& B \). Here \( \Gamma \) denotes a type environment recording the types of free variables, \( T \) denotes the type of \( P \) and \( B \) the effect associated with executing \( P \).

In a polymorphic functional language, a type and effect systems can be used to control resource usage, such as memory management. When the programming language includes the notions of communication and concurrency, effects (also called behaviours) are terms of a process algebra and can, just like the programming language itself, be given a labelled transition semantics Nielson et al. 1999. The main property that these type systems guarantee is that whenever a well-typed program performs a communication c, the effect of the program decreases by the effect corresponding to c.

2.2. Regular types

Regular types first appeared in the context of calculi for describing concurrent objects. In this approach types are words taken from regular languages.

Many of the type systems following this approach deal with the notion of non-uniform objects in object-oriented programming. In an object, each of its methods can be enabled or disabled according to the internal state of the object. A simple example is that of a queue (dequeue is disabled if it is empty), another is a finite buffer (write is disabled if the buffer is full).

Active objects are those that may dynamically change behaviour, and a typing discipline for ensuring the absence of message-not-understood errors will need to take this dynamic behaviour into consideration. Moreover, non-uniformity poses particular problems for typing concurrent objects. In the presence of synchronisation constraints, the availability of a service will depend on the internal state of the object.

There are several ways of dealing with this issue. Nierstrasz 1995 uses the traces of menus offered by (active) objects as a notion of behavioural types and proposes a
notion of subtyping, request substitutability, that generalizes the Liskov Substitution Principle [Liskov and Wing 1994]—by [Wegner and Zdonik 1988]. This substitution principle requires that whenever \( S \) is a subtype of \( T \), we can replace an object of type \( T \) with another object of type \( S \); the resulting program will still have the behaviour of the original program. This means that a service can be refined as long as the original promises are still kept. According to the extension relation defined by [Brinksma et al. 1995], request substitutability gives rise to a transition relation which is close to the failures model.

[Colaço et al. 1997; Colaço et al. 1999] propose an actor calculus supporting objects that dynamically change behaviour. The process language is inspired both by the (functional) Object Calculus [Abadi and Cardelli 1996] and by Typed Concurrent Objects (an asynchronous \( \pi \)-calculus with input-guarded labelled sums and output selections) [Vasconcelos 1994], following the actor model [Hewitt et al. 1973]. The authors define a type system that detects “orphan messages”. These are messages that may fail to ever be accepted by any actor, because dynamic changes to the interface of the actor causes the service requested not to be available anymore. Types describe interfaces annotated with multiplicities (that is, how often a method can be invoked), thus excluding dynamic information, and the type system involves complex operations on a lattice of types. The authors give a type inference algorithm for this type system, based on the familiar notion of solving set constraints by resolution.

Finally, [Najm and Nimour 1997; Najm et al. 1999a; Najm et al. 1999b] propose a calculus of objects that can describe interfaces that change dynamically; the calculus is based on the asynchronous \( \pi \)-calculus. The authors propose a type system that handles dynamic method offers in interfaces and guarantees a liveness property, namely that every pending request will eventually be processed. In this case, a type is defined as a set of deterministic guarded parametric equations that describe a transition system (which may be infinite). The type system has notions of type equivalence, compatibility and subtyping defined using this transition system approach; in particular the notion of subtyping is based on the notions of strong simulation and strong bisimulation and is decidable. Finally, the authors are able to define a type inference algorithm for their system.

2.3. Interface automata

Interface automata provide an automaton-based approach to behavioural types. I/O-automata are now used to specify interfaces of components, and a refinement relation is used to compare abstract and concrete interface specifications.

[Lee and Xiong 2004] give a behavioural type system based on interface automata for the Ptolemy II framework (for composing concurrent components) that captures the dynamic interaction in an environment for component-based design. The interaction types and component behaviour are given as interface automata, and type checking is carried out via composition of automata. [Chouali et al. 2010] present a formal approach, based on interface automata and protocol specifications, that allows one to adapt components and eliminate possible behavioural mismatches that occur in interactions. The approach ensures that components can be re-used in diverse situations without their code being affected. [Chouali and Hammad 2011] describe an approach that uses a combination of component models and interface automata to assemble components and to formally verify that they are interoperable.

2.4. Intersection types

An intersection type system introduces a type constructor \( \land \); a program has type \( T_1 \land T_2 \) if it may, possibly in different runs, have properties corresponding to both \( T_1 \) and \( T_2 \).
This can be thought of as a notion of polymorphism but also as describing an *internal choice* between the behaviours corresponding to $T_1$ and to $T_2$.

Intersection types first arose in the setting of typed $\lambda$-calculi [Barendregt et al. 1983] and are closely related to the model theory of $\lambda$-calculus [Barendregt et al. 2013]. Typability in intersection type systems characterizes normalization behaviour of terms in the $\lambda$-calculus (including exact characterization of the strongly normalizing terms [Coppo et al. 1981; Pottinger 1980]). Intersection types have been used to express behavioural abstractions of program behaviour in settings including abstract interpretation [Jensen 1991, Jensen 1992, Coppo and Ferrari 1993], type refinement [Freeman and Pfenning 1991], model checking [Naik and Palsberg 2005, Kobayashi and Ong 2009], and synthesis [Rehof 2013]. As a consequence, the intersection type system can be regarded as a premier example of a behavioural type system.

The dual notion of *union types* corresponds to the notion of *external choice* and has often been introduced alongside intersection types [Pierce 1991, Igarashi and Nagira 2007, Dunfield and Pfenning 2003, Dunfield 2012, Barbanera et al. 1995].

It is interesting to note the cross-fertilization between more traditional type theory and process-oriented behavioural types. A binary session type $(S_1, S_2)$ with endpoint types $S_1$ and $S_2$ can be thought of as a notion of intersection type [Dezani-Ciancaglini et al. 2009]. [Bettini et al. 2008] combine session types and union types. Finally, [Padovani 2010b] shows that a type system with union and intersection types makes it possible to capture the branching and selection constructs of binary session types.

### 2.5. Typestates

Typestates are a notion of behavioural types dating back to [Strom and Yemini 1986]. In this approach, the type of an entity depends on the operations that are permitted for the entity, when at a particular state. Each type has associated with it a set of typestates, partially ordered; operations on entities of the type are correct if the resulting values are of a typestate reachable by a typestate transition (following the order).

Therefore, typestates are akin to finite-state machines, and a language equipped with a static type system based on them can check at compile-time if all possible sequences of operations are valid with respect to a correct use of the application.

Several languages (mainly object-oriented) support typestates. [Sunshine et al. 2011] developed the Plaid programming language, extending an object-oriented programming language, where typestates incorporate, in the traditional notion of class type (the interface, or the collection of method signatures), the representation (the fields) and the behaviour (the actual implementations of methods). Typestates may change over time, and the type system of Plaid makes it possible to track these changes. [Gay et al. 2010] give semantics to a distributed concurrent object-oriented programming language by means of a unified treatment of communication channels and their session types (cf. Section 2.7) together with a notion of typestates that supports non-uniform objects (cf. Section 2.2).

### 2.6. Processes as types

Another approach, originating from work on type and effect systems, is that of considering processes as types. Here, types are processes that are sound abstractions of the behaviour of programs, and an analysis of the type thus becomes an analysis of the behaviour of a process. Since program properties are checked at the level of types, not programs, these properties are often decidable, and therefore this approach can benefit from the advantages of type checking as well as model checking.

[Boudol 1997] describes a dynamic type system for the Blue Calculus, which is a version of the $\pi$-calculus that directly incorporates the $\lambda$-calculus. In this type system,
types are functional types in the sense of the simply typed \( \lambda \)-calculus but now also incorporate a version of recursive Hennessy-Milner logic in which modalities are interpreted as named resources. In this type system, types are inhabited by processes, and the type system is able to express a form of causality in the way names are used within a process. This ensures that messages sent to a name will meet a corresponding offer.

[Puntigam 2001a; Puntigam 2001b; Puntigam and Peter 2001] define a calculus of active objects (a concurrent object calculus also inspired by the actor model) in which process types impose constraints on the ordering of messages. A static type inference system ensures that even when the set of acceptable messages changes dynamically for an object, every message sequence sent to it will eventually be received.

[Ravara and Vasconcelos 2000] developed a behavioural type system for TyCO [Vasconcelos 1994] to ensure the absence of ‘message-never-understood’ errors in non-uniform concurrent objects. The type safety result guarantees that every message has a chance of being received if it requires a method that may become enabled at some point in the future. The type language is ABT [Ravara et al. 2012], a process algebra similar to the Basic Parallel Processes.

[Kobayashi 2000; Kobayashi et al. 2000] has studied type systems for detecting deadlock and livelock in a synchronous \( \pi \)-calculus. In these systems, the type of a channel carries information about both the arity of the channel and its usage. By this is meant information about the admissible sequences of input and output actions, about when the channel is to be used, and if it must be used.

[Igarashi and Kobayashi 2004] describe a so-called Generic Type System. This is a general framework that makes it possible to develop type systems that capture various properties of \( \pi \)-calculus processes. In the Generic Type System, types are processes from the restriction-free fragment of CCS and the type system involves a general sub-type relation and a consistency condition on types. Type systems for concrete properties can now be obtained as instances of the generic system: a specific type system for a given property can be had by defining an instantiation of the general subtyping relation and of the consistency condition. The properties of the particular type system follow from the general properties of the Generic Type System. All one needs to prove is that reductions on types preserve consistency and that consistency on types implies the desired condition on processes. The Generic Type System is able to capture specific type systems for safety properties (including arity mismatch, race-freedom and even deadlock-freedom), but not liveness properties.

More recently, [Caires and Seco 2013] introduced the concept of behavioural separation. This is a general principle for controlling interference in concurrent, higher-order imperative programs (written in languages such as ML or Java). Behavioural separation types combine notions that originate in behavioural type theories and separation logics [Caires 2008]; this notion of type makes it possible to enforce fine-grained interference control disciplines and at the same time preserve compositionality, information hiding, and flexibility. Behavioural separation contracts specify how the values of a program can be used safely by client code, by integrating behavioural operations such as sequential and parallel separation types within a standard linear type theory.

2.7. Session types
One may view a behavioural type as constituting a protocol that specifies the sequence and kinds of messages transmitted on communication channels in a distributed system. For a system to be correct, its channels must follow the protocol. In this approach, types are seen as terms of a simple process language. A term in this language describes one side of a communicating process. There are three main approaches to this, two of which are rooted in the \( \pi \)-calculus.
Session types express the protocol followed by a channel as a type [Honda 1993; Takeuchi et al. 1994; Honda et al. 1998]. Session types are usually associated to (bi-directional) communication channels. A session type describes not just the types of individual messages (which may in general be session types themselves), but also the steps followed by the protocol and thus also the permissible sequences of messages. The parties reading and writing from the channel at different ends see the channel at dual, or complementary, types. One can then use static typechecking to verify that processes using the channel do so in accordance with the protocol. The concept was originally introduced for a dialect of the $\pi$-calculus, but session types have since been used to give type disciplines for other languages, including functional [Pucella and Tov 2008] and object-oriented programming languages [Dezani-Ciancaglini et al. 2009], and even to operating systems [Fähndrich et al. 2006].

Multiparty session types extend binary session types by describing interaction patterns that include more than two participants [Honda et al. 2008]. Multiparty session types come equipped with a projection operator that yields, for each participant, a local type, akin to a binary session type. Individual processes may then be checked against projections. Multiparty session types guarantee the absence of deadlocks, a property that the various independent binary sessions cannot. [Caires and Vieira 2010] introduce conversation types, aiming at describing service-based systems with multiparty interactions for the conversation calculus [Vieira et al. 2008]. Conversation types unify local and global multiparty types, defining a way to distribute parts of the protocol in a compositional way, allowing in particular to address conversations among an unanticipated number of participants.

Contracts pose a strong emphasis on the definition of general theories for checking the conformance of the communication behaviour of processes against abstract descriptions of the expected input-output actions. Such abstract behaviour can be described with simple process algebras [Castagna et al. 2009b] or more generally as labeled transition systems [Bravetti and Zavattaro 2009a]. Moreover, the abstract specification can include only the description of a binary client-service relationship [Carpineti et al. 2006] or a more complex multi-party choreography [Bravetti and Zavattaro 2008b]. In all the cases, theories have been developed to check whether a process is conformant with one of the roles in the abstract description, and usually these are defined in terms of behaviour refinement: the behaviour of the process refines the abstract behaviour preserving some correctness conditions required on the entire system.

The following sections concentrate on session types, conversation types and contracts.

3. BINARY SESSIONS

This section describes behavioural type systems for binary sessions, that is, communications involving exactly two participants.

3.1. Linear and shared types

Binary session types describe communication patterns involving exactly two participants. Multiparty session types, described in Section 4, fundamentally depart from this assumption. A binary session type describes a protocol as seen from the point of view of one of the two participants.

The basic constructs denote the two contributions to a message exchange. One writes $!\text{nat}$ to denote the output of, say, a natural number, and one writes $?\text{nat}$ to denote the input of a natural number. Types may be composed by means of a prefix operator. If $T$ is a type, then $!\text{nat}.T$ is also a type, and denotes an interaction that starts with the output of a natural number, followed by the behaviour prescribed by $T$. The completed protocol, that is, the protocol on which no further interaction is possible, is denoted by
end. Putting all these pieces together one can write a session type

!nat.?bool.end

describing a series of message exchanges, starting with the output of a natural number, followed by the input of a boolean value, followed by termination.

The above type describes an interaction as seen from one of the participant’s point of view. The type for the second participant is the complementary, or the dual, obtained as follows. The dual of output is input, the dual of input is output, and the dual of end is end. In either case, input or output, the types of the values exchanged in messages remain unchanged. In this way, the dual of !nat.?bool.end is ?nat.!bool.end.

A further important construct usually present in session types is choice. Again we have two points of view: that of a participant that offers the menu of options, and that of the participant that selects a particular option. A choice between depositing or withdrawing money at some ATM can be written, from the point of view of the client as $\oplus\{\text{deposit} : T_1, \text{withdraw} : T_2\}$. The ATM, on the other hand, offers a menu, written as $\&\{\text{deposit} : T_3, \text{withdraw} : T_4\}$. Duality also applies to choice. The above two types are dual if $T_1$ is a dual of $T_3$, and $T_2$ is a dual of $T_4$.

So far session types can only offer series of fixed-length interactions. Often the exact number of messages exchanged (and choices performed) cannot be determined in advance. Just think of the type describing a typical session between a client and an ATM: after selecting option withdraw and providing the amount, the client would like to have all his choices available again, so that, e.g., she may then check the remaining balance. Potential infinite behaviour in session types is usually introduced by means of recursion operator: if $T$ is a type and $a$ is a type variable, then $\text{rec } a.T$ and $a$ are also types. Duality for recursive types pose delicate questions; [Bernardi and Hennessy 2013b] discuss various alternatives. We are now in a position to introduce the type $T_{\text{ATM}}$ for our ATM machine, as seen from the point of view of a client. Clients start by providing their user-id in the form of a string; the authentication details are omitted. They are then offered a four-way menu. If withdraw is picked, then clients must provide a nat describing the amount to be withdrawn, to which the ATM then answers with a dispense or overdraft option. In either case, the client is again provided with the four-choice menu.

1string.rec a. $\oplus\{\text{deposit} : !\text{nat}.a$

$\quad\text{withdraw} : !\text{nat}.\&\{\text{dispense} : \text{a}, \text{overdraft} : \text{a}\}$,

$\quad\text{balance} : ?\text{nat}.\text{a},$

$\quad\text{quit} : \text{end}\}$

The types discussed so far describe interactions meant to be run without interference. They are usually called linear. Complementary to these, we need types whose objects may be shared, and which may be used, in particular, to establish new (linear) sessions. When it comes to shared types there are a few variants in the literature. For [Gay and Hole 2005], a shared type $S$ is either a base type, such as nat or bool, or else a type $^\wedge[T]$ describing an object capable of carrying a session of type $T$. Because sessions themselves may carry shared types, recursive types are also usually present at the level of shared types. As an example, the shared name of an ATM is of type $^\wedge[T_{\text{ATM}}]$. [Honda et al. 1995] make it explicit that the type is capable of generating both session ends, by writing instead $\langle T_{\text{ATM}}, T'_{\text{ATM}}\rangle$, where $T'_{\text{ATM}}$ is a dual of $T_{\text{ATM}}$.

[Vasconcelos 2012] eliminated the stratification of types into linear and shared, by classifying each prefix with a lin (linear) or un (unrestricted or shared) qualifier. The type of the shared name of the ATM name becomes $\text{rec } b.un!T_{\text{ATM}}.b$. Eliminating stratification allows for describing channels that start as linear and end as unrestricted (see ACM Computing Surveys, Vol. V, No. N, Article A, Publication date: January YYYY.
examples in [Vasconcelos 2012]). In order to capture within a single type the capabilities of both ends of a channel, [Giunti and Vasconcelos 2013] use pair types \((T_1, T_2)\) where \(T_1\) describes the behaviour of one end and \(T_2\) the behaviour of the other. Contrary to [Honda et al. 1998], types \(T_1\) and \(T_2\) need not be dual of each other.

### 3.2. \(\pi\)-calculus for binary session types

Types need programming languages. Session types have been incorporated in different programming paradigms. Here we focus on the \(\pi\)-calculus, a process algebra on which session type systems are traditionally tested.

Session types require mild variations of the \(\pi\)-calculus as introduced by [Milner et al. 1992]. In the \(\pi\)-calculus, types are assigned to channels. If channel \(x\) is of type \(!T_1.T_2\) and value \(v\) is of type \(T_1\), then one may write \(x!v.P\) to denote a process that writes \(v\) on \(x\) and continues as prescribed by process \(P\). The type of \(x\) in \(P\) is \(T_2\). Conversely, if \(x\) is of type \(?T_1.T_2\), then \(x?y.P\) denotes a process that reads a value from channel \(x\), binds it to \(y\), and continues as \(P\). In \(P\), variable \(y\) is of type \(T_1\) and \(x\) is of type \(T_2\).

Processes are typed against typing contexts, essentially a map from variables to types. Sequents are of the form \(\Gamma \vdash P\) and say that process \(P\) is well typed under context \(\Gamma\). Following to the description above, the typing rules for input and output (of linear values on linear channels) [Honda et al. 1998] should be easy to understand.

\[
\begin{align*}
\Gamma, x: T_2 \vdash P & \quad \Gamma, y: T_1, x: T_2 \vdash x!y.P \quad \Gamma, y: T_1, x: T_2 \vdash x?y.P \\
\end{align*}
\]

In order to deal with choice, [Honda et al. 1998] introduce two new language constructs, called branch and select. If \(x\) is of type \&\{\(l_1: T_1, l_2: T_2\}\}, then \(x \triangleright \{l_1: P_1, l_2: P_2\}\) denotes a process (usually called branching) that offers two options, and behaves as \(P_1\) if option \(l_1\) is selected and as \(P_2\) if option \(l_2\) is selected. Conversely, if \(x\) is of type \(\oplus\{l_1: T_1, l_2: T_2\}\), then \(x \triangleleft l_2. P\) selects the \(l_2\) option on channel \(x\) and proceeds as \(P\).

The fundamental change to the \(\pi\)-calculus often required by session types forces the syntactic distinction of the two ends of a channel. [Gay and Hole 2005] write \(x^+\) and \(x^-\) to speak about the two ends of channel \(x\). In the \(\pi\)-calculus [Milner et al. 1992], channels are created by means of the \(\nu\) constructor, as in \((\nu x: T)P\). Gay and Hole, write \((\nu x: T)P\) and use \(x^+\) of type \(T\) and \(x^-\) of a type dual of \(T\) in process \(P\), as in

\[(\nu x: !\text{nat}. T)(x^+!5.P_1 | x^-?z.P_2)\]

which reduces in one step to \((\nu x: T)(P_1 | P_2[5/z])\), where \(P_2[5/z]\) denotes process \(P_2\) with the free occurrences of variable \(z\) replaced by the value \(5\). An alternative formulation uses (non-annotated) variables to describe the two ends of a channel [Vasconcelos 2012]. In this case, when creating a new channel we explicitly name its two ends using conventional variables, as in \((\nu xy: !\text{nat}. T)(x^+5.P_1 | y?z.P_2)\).

The syntactic distinction between the two ends of channels is required when two situations arise together:

1. Processes may end up with both ends of a channel and use them in sequence, as in \(x!5.x?z.P\), where the first \(x\) denotes one end whereas the second denotes the other end of a same channel, and
2. Types describe only one end of a channel, as in \(!\text{nat}. T\).

[Yoshida and Vasconcelos 2007] and [Giunti and Vasconcelos 2013] further discuss the problem. The type system in [Honda et al. 1998] does not require this distinction because the particular nature of channel passing precludes processes from ending up with both the ends of the channel, thus avoiding condition (1). The same happens in the interpretation of session types in intuitionistic linear logic discussed in Section 6.1.
The system by [Giunti and Vasconcelos 2013] uses types that describe both ends of a channel, as in \((\text{nat}.T_1, \text{nat}.T_2)\), thus steering clear from condition (2).

### 3.3. Type safety

Given a notion of types as presented in Section 3.1 and a language as presented in Section 3.2, one may ask what benefits types will bring, apart from providing an abstract description of the communication patterns of programs. After all, type systems are meant to prevent certain kinds of run-time errors. In the case of binary session types, well-typed programs are exempt from a series of common programming errors, which we can categorize in two classes: **communication errors** and **race errors**. In the first case one finds two processes running in parallel, trying to interact on a given channel, but in a non-compatible way, for example two outputs \(x^+!5 \mid x^-!7\), or a selection and an input \(x^+ \triangleleft \text{quit} \mid x^-?y\). In the second category one finds two process competing for a given resource, e.g., two outputs competing for a same input \(x^+!5 \mid x^+!7 \mid x^-?y\).

### 3.4. Binary contracts

Contracts take an approach different from session types, by using process-algebra like languages or labeled transition systems for describing abstractions of the communication behaviour of programs. Before reviewing the most relevant literature, we present a simple example of a contract-like description of the binary interaction between a client and an ATM in the example reported in Section 3.1. Using the notation of [Bravetti and Zavattaro 2009a], the behaviour of the ATM from the point of view of the server is described as follows

\[
\text{auth}; (\text{deposit}; \text{amount} + \text{withdraw}; \text{amount}; (\text{dispense} + \text{overdraft}) + \text{balance}; \text{amount})^*; \text{quit}
\]

While the prefixes in a binary session type describe the types that are communicated or (in the case of branching and selection) the choices that are possible, the labels in a contract directly describe the actions allowed by the ATM.

The input action \text{auth} identifies the initial authentication data sent by the client to the ATM. After this step, the ATM enters a cycle offering three functionalities: deposit an amount indicated by the client, withdraw or show the current balance. The cycle is terminated by the \text{quit} action. The behaviour of a client interested in asking for the balance and then performing a withdraw action can be represented as follows.

\[
\text{auth}; \text{balance}; \text{amount}; \text{withdraw}; \text{amount}; (\text{dispense} + \text{overdraft}); \text{quit}
\]

Intuitively, the two contracts are compatible in the sense that their combination guarantees the completion of the expressed protocols. Indeed, compatibility corresponds to the notion of communication error freedom from Section 3.3. The papers on contracts that we review below formalize appropriate notions of compatibility between contracts.

The pioneering work [Fournet et al. 2004] introduced the notion of contract as the description of the input-output behaviour of processes. It adopted the process algebra CCS to denote contracts. Its main contribution is the formalization of the notion of stuck-free conformance: a CCS process \(P\) conforms to a contract \(C\) if \(P\) can replace \(C\) in every context preserving stuck freedom (roughly, stuck freedom corresponds to absence of local deadlocks: the exact definition can be found in [Fournet et al. 2004]).

This approach has inspired several subsequent works. [Carpineti et al. 2006] considered a similar language for the description of contracts and processes. They introduced a different notion of conformance, namely, an asymmetric client-service compliance notion: a client and a service are compliant when in every computation the client is guaranteed to reach a successful state. This approach has been subsequently extended by some of the authors in two directions. [Laneve and Padovani 2007] introduce the notion of input and output alphabet associated to a contract, while [Castagna et al. 2007]...
Hüttel et al. 2009b propose dynamic filters that can be associated to services in order to eliminate interactions on non admitted channels. Both approaches aim at relaxing the conformance relation, thus extending the set of processes that can safely replace a given service. Interesting variants of this approach deal with dynamic communication topologies Castagna and Padovani 2009 or with the standard languages for the description and composition of Web Services Laneve and Padovani 2013.

3.5. Variations and extensions of binary sessions

Different session calculi have been proposed in the literature, either by extending the types above with new features or by changing the operational semantics of the underlying languages.

Asynchronous semantics. Gay and Vasconcelos 2010 study a functional language with asynchronous (buffered) semantics. In the context of the π-calculus with sessions, Kouzapas et al. 2013 consider a semantics based on order-preserving asynchronous communication inside each session and asynchronous message arrival for general channels. Honda et al. 2008 develop the notion of multiparty session types on top of an asynchronous π-calculus (see Section 4).

Event-driven programming. Kouzapas et al. 2013 extend session types with non-blocking detection of message arrival (events) and dynamic inspection of session types to model event-driven programming. As a result, they can encode the event selector, a central component of event-driven systems, enabling the development of type-safe event-driven applications. Also, they prove the correctness of the Lauer-Needham duality by defining a systematic transformation from multithreaded to event-driven processes which is type- and semantics-preserving.

Exceptions. Carbone et al. 2008 extended binary sessions with a throw primitive to raise exceptions, and exception handlers for handling them. Interestingly, exceptions that require coordinate handling from both the session participants are considered. Both safety and liveness properties are ensured.

Service-oriented programming. Service-Oriented Computing (SOC) applications are generated by dynamically looking for available services on the network, and composing them so as to obtain the desired functionalities. The different services communicate by exchanging messages over the network (standards such as SOAP are used to ensure interoperability). While current standards (such as WSDL) only check syntactic compatibility of service signatures, session types have been proposed to ensure also behavioral compatibility. Some calculi to model SOC systems have been proposed.

— SCC Boreale et al. 2006 was the first attempt to define a calculus able to directly model SOC systems in a π-calculus style. In particular, SCC features service definition and invocation as primitive operators. When a service is invoked, a private session is created to allow communication between the two processes. Results computed inside the session are propagated to the upper level using a return primitive. This primitive alone is however not enough to model the complex orchestration patterns used to coordinate different client/service pairs. New calculi have been developed to address this issue. A type system based on session types to guarantee deadlock freedom in SCC is presented in Bruni and Mezzina 2008.

— CASPIS Boreale et al. 2008 extends SCC with pipelines, allowing to define flows of data between services, thus improving the modeling of orchestration aspects.

— SSCC Cruz-Filipe et al. 2013, Lanese et al. 2007 also extends SCC to allow for easier orchestration, but using streams. Streams (differently from pipelines) are orthogonal w.r.t. the session hierarchy. Session types ensuring correctness of ses-
session communication are presented by [Lanese et al. 2007]. A simpler type system, ensuring sequentiality of communication, is exploited by [Cruz-Filipe et al. 2008] to enable program transformations to break large sessions into smaller ones. The transformation is not correct for general sessions.

Other approaches to modelling services use multiparty sessions and are presented in the next section.

4. MULTIPARTY SESSIONS

In many cases, it is possible to describe and reason about conversation patterns by means of a composition of binary sessions. However, there are also patterns involving more than two communicating parties for which binary sessions do not suffice.

4.1. Global and local types

Consider by way of example an extension of the ATM example from Section 3, where the ATM establishes a different session with the bank central server reporting any operation that the client chooses. Intuitively, the ATM needs to contact the bank only after the client has made a choice, and never before. Unfortunately, since sessions are binary such a constraint cannot be imposed at type level. For example, a well-typed implementation of the ATM could be a process that, independently from which branch is selected, always reports to the bank that the client has made a deposit.

To address this problem, [Honda et al. 2008] propose a generalisation of binary session types called multiparty session types. Multiparty session types provide for global descriptions of interactive behaviour. Under this paradigm, a software architect prepares a global view of all the message exchanges that take place, instead of separately defining the behaviour of each individual channel end-point (as in binary session types, where we only specify the behaviour of each side of a binary session). The local behaviour of each end-point can be mechanically obtained from the global description by applying a projection operation. A global description is therefore a “formal blueprint” of how a communicating system should behave and it provides a concise specification of how messages flow within the system. This has a major impact on software quality, since a global description will

1. decrease the risk of introducing programming errors;
2. make it easier to detect such errors (both manually and by automatic means), and
3. guarantee the absence of deadlocks.

One can use global descriptions at different levels of abstraction, ranging from abstract descriptions of protocols (multiparty session types) to descriptions of concrete implementations (choreographic programming). As an example, the following is a global type describing a session with three participants—Client, ATM and Bank—where the ATM correctly reports to the bank all the choices made by the client:

\[
\text{Client} \rightarrow \text{ATM} \langle \text{string} \rangle. \text{rec } a.
\]

\[
\begin{align*}
\text{Client} \rightarrow \text{ATM} \langle \text{string} \rangle & \rightarrow \text{ATM} \langle \text{nat} \rangle. \\
& \rightarrow \text{Bank} \left\{ \text{deposit: ATM} \rightarrow \text{Client} \langle \text{nat} \rangle, \text{withdraw: ATM} \rightarrow \text{Client} \langle \text{nat} \rangle, \text{balance: ATM} \rightarrow \text{Client} \langle \text{nat} \rangle, \text{quit: ATM} \rightarrow \text{Bank} \langle \text{nat} \rangle \left\{ \text{dispense: ATM} \rightarrow \text{Client} \langle \text{nat} \rangle, \text{overdraft: ATM} \rightarrow \text{Client} \langle \text{nat} \rangle, \text{balance: ATM} \rightarrow \text{Bank} \langle \text{nat} \rangle, \text{quit: ATM} \rightarrow \text{Bank} \langle \text{nat} \rangle \right\} \right\\
\end{align*}
\]

The multiparty session type above (or simply global type) specifies in which order the implementation of the client, the ATM and the bank have to exchange messages and
the order of requests that are involved. The key operations in a global type are inter-
actions such as

\[
\text{Client} \to \text{ATM}(\text{string})
\]
in which a sender (Client) sends a message of some type (string) to a receiver (ATM), and
choices such as

\[
\text{ATM} \to \text{Client}\{\text{dispense: \ldots, overdraft: \ldots}\}
\]
in which a sender (ATM) asks a receiver (Client) to select a certain branch.

Global types are used for checking programs running in parallel, each implementing
one of the roles specified in the type, e.g., Client, ATM, and Bank. In order to realise that,
a notion of projection from global types to local types is defined. For example, the local
type corresponding to ATM in the interaction above would be:

\[
\text{Client} \ ? \ \text{string}. \ \text{rec} \ a. \ \text{Client} & \left\{
\begin{aligned}
\text{deposit:} & \ \text{Client} \ ? \ \text{nat}. \ \text{Bank} \ \oplus \ \{\text{deposit:} \ a\}, \\
\text{withdraw:} & \ \text{Client} \ ? \ \text{nat}. \\
\text{Bank} \ \oplus \ \{\text{withdraw:} & \ \text{Client} \ \oplus \ \{\text{dispense:} \ a\} \}, \\
\text{balance:} & \ \text{Client} \ ? \ \text{nat}. \ \text{Bank} \ \oplus \ \{\text{balance:} \ a\}, \\
\text{quit:} & \ \text{Bank} \ \oplus \ \{\text{quit:} \ \text{end}\}.
\end{aligned}
\right.
\]

Note how local types are slightly more complex than standard binary session types
since each communication operation is now labeled also with the party this role is
supposed to communicate with (rather than ?nat we now write Client?nat). Multiparty
session types provide the same safety guarantees as binary session types, described in
Section 3.3, but in the more complex context of multiparty sessions.

An important question when dealing with multiparty sessions is that of finding a
suitable language for describing these interactions. [Honda et al. 2008] present a gen-
eralisation of binary sessions to multiparty asynchronous sessions for the \(\pi\)-calculus.
In contrast to the binary case, sessions are now established between multiple pro-
cesses via multiparty synchronisation. Then, private (in-session) communication is
carried out, asynchronously, between session participants. Technically, sessions are
established as follows:

\[
a[2..n](\tilde{s}).P_1 | a[2](\tilde{s}).P_2 | \cdots | a[n](\tilde{s}).P_n \rightarrow (\nu \tilde{s})(P_1 | \cdots | P_n | s_1: \emptyset | \cdots | s_m: \emptyset)
\]

In the above, the term on the left-hand side of the reduction \(\rightarrow\) contains \(n\) processes
running in parallel, each of them willing to establish a session on public channel \(a\).
Note how each participant is labelled with a role name \(1, 2, \ldots, n\). Each label (where,
for technical reasons \(1\) is actually denoted by \(2.\)) corresponds to the unique role that
a participant in the new session has to play. The session is established through a dis-
tributed synchronisation which creates the session (private) channels \(s_1, \ldots, s_m\) and
the corresponding FIFO queues (denoted in the reductum by \(s_1: \emptyset, \ldots, s_m: \emptyset\)). Once
the connection is established, processes \(P_1, \ldots, P_n\) can asynchronously communicate
by using the queues corresponding to one of the channels \(s_1, \ldots, s_m\).

4.2. Conversation types

Conversation types [Caires and Vieira 2010] extend (binary) session types so as to ad-
dress multiparty interaction. Although conversation types were originally introduced
to type terms in the Conversation Calculus [Vieira et al. 2008], the approach carries
over to a more foundational setting, namely to a “mild” extension of the \(\pi\)-calculus in
which communication actions are labelled. Given that a session type characterises the
usage of a single channel by two-parties, it seems natural to consider that a multiparty
extension of a session type characterises the usage of a single channel by multiple par-
ties. To motivate the underlying model, consider the following example where three
(concurrent) threads interact in a channel \textit{bank}: the leftmost thread sends 100, the middle thread sends \texttt{true}, and the rightmost thread sequentially receives two values (omitting the trailing 0, as usual).

\[
\text{bank!}100 \mid \text{bank!}\texttt{true} \mid \text{bank?}x.\text{bank?}y
\]

Looking at the specification one may immediately identify a potential communication problem: two threads are simultaneously trying to send a message, a communication race. As a consequence, the receiving process may actually receive first either value 100 or value true, making it impossible to (statically) characterise how the received values can be used. Now consider that we extend the specification above, by adding labels to communication actions.

\[
\text{bank!}\text{deposit}(100) \mid \text{bank!}\text{letOverdraft}(\texttt{true}) \mid \text{bank?}\text{deposit}(x).\text{bank?}\text{letOverdraft}(y)
\] (1)

Thanks to the labels we may now distinguish two synchronisations, given the order imposed by the receiving process: first a \texttt{deposit} labelled message is exchanged, then a \texttt{letOverdraft} labelled message. Labels thus allow one to recover pairwise linear interactions, even if multiple parties share a single communication medium. Thus the safety properties described in Section 3.3 hold.

Let us now turn to a typing characterisation of the system given in (1). The leftmost process uses channel \textit{bank} to output an integer, which we may characterise with the (session) type !\texttt{Int}.end. Extending the type with the corresponding label we then have the conversation type !\texttt{deposit}(\texttt{Int}).end, and likewise for the process in the middle we have !\texttt{letOverdraft}(\texttt{Bool}).end. On the receiving end, the rightmost process may be characterised by type ?\texttt{deposit}(\texttt{Int}), ?\texttt{letOverdraft}(\texttt{Bool}).end. The sequential exchange of messages \texttt{deposit} and \texttt{letOverdraft} in channel \textit{bank} is captured by conversation type \texttt{letOverdraft}(\texttt{Bool}).end, where each \texttt{tau} captures a message exchange internal to the characterised system.

We may draw a comparison between the previous characterisations and the local and global types of [Honda et al. 2008], respectively, characterising the interface with the external environment (via output ! and input ? descriptions) and the internal message exchanges (via \texttt{tau} descriptions). Differently from [Honda et al. 2008], conversation types mix interface and internal descriptions at the same level in the type language. For the sake of illustration consider the system below, consisting of part of (1).

\[
\text{bank!}\text{letOverdraft}(\texttt{true}) \mid \text{bank?}\text{deposit}(x).\text{bank?}\text{letOverdraft}(y)
\]

Channel \textit{bank} is used according to type ?\texttt{deposit}(\texttt{Int}).\texttt{letOverdraft}(\texttt{Bool}).end, saying that first a \texttt{deposit} message is received after which message \texttt{letOverdraft} is exchanged. Such type is obtained as a behavioural combination of two (local) types, namely !\texttt{deposit}(\texttt{Bool}).end and ?\texttt{letOverdraft}(\texttt{Int}), ?\texttt{letOverdraft}(\texttt{Bool}).end. Notice that the \texttt{tau} combines the (dual) output and input descriptions for message \texttt{deposit}, while the reception of message \texttt{letOverdraft} is left at the interface level, open to synchronise with an output originating from the external environment.

This ability to \textit{merge} behaviours (or symmetrically, the ability to \textit{split} behaviours into smaller pieces), defined in an algebraic way in [Caires and Vieira 2010], allows one to compositionally characterise systems. Furthermore, the ability of characterising subsystems allows one to address configurations in which parties engage dynamically in conversations: the behavioural contribution of a single participant to a conversation may be completely retained and carried out by the participant or the participant may choose at some point to delegate one or more parts of the protocol to other parties (while possibly retaining some other slice of the protocol). In fact, since the underlying model (the labelled \pi-calculus) does not support atomic multiparty synchronisation, the way in which multiparty interaction is modelled is by allowing multiple parties to join
an ongoing conversation (realised by channel name passing). Using the terminology from session types, conversation joining is supported by channel delegation, only now delegation is partial: the delegating party will still have access to the communicated channel.

[Baltazar et al. 2012a] introduced a novel type construct to capture the idea that some behaviours are not necessarily carried out immediately and can actually take place sometime in the future. Type $\Diamond letOverdraft(\text{Bool})$ says that the output of message letOverdraft will happen sometime but not necessarily immediately. The $\Diamond$ type constructor is related to the eventually operator from temporal logic and satisfies the same basic axioms. When composing the (sometime) output of message letOverdraft and the (immediate) output of message deposit we obtain the type $! deposit(\text{Int}).\Diamond ! letOverdraft(\text{Bool})$. End that composed with the dual (sequential) capabilities $? deposit(\text{Int}).? letOverdraft(\text{Bool})$. End yields the overall protocol.

4.3. Multiparty contracts

Contracts for multiparty process composition were initially investigated by [Bravetti and Zavattaro 2007]. The idea is to adopt a choreography language, like WS-CDL [Kavantzas et al. 2005] or its formalization [Busi et al. 2005], to describe the globally observable behaviour of correctly interacting peers. As a simple example, consider the description of the binary interaction between the client (denoted by $C$) and the ATM (denoted by $A$) described in Section 3.4 expressed in the language proposed in [Bravetti and Zavattaro 2007].

\[ auth_{C\rightarrow A}; \text{balance}_{C\rightarrow A}; \text{amount}_{A\rightarrow C}; \]
\[ \text{withdraw}_{C\rightarrow A}; \text{amount}_{C\rightarrow A}; (\text{dispense}_{A\rightarrow C} + \text{overdraft}_{A\rightarrow C}); \text{quit}_{C\rightarrow A} \]

The notation $auth_{C\rightarrow A}$ expresses an interaction on the operation $auth$ having $C$ for sender and $A$ for receiver.

As an example of a multiparty interaction expressed in terms of a multiparty contract, we can describe the behaviour of the client role ($C$), the ATM role ($A$) and the bank role ($B$), discussed in Section 4, at least for the balance and withdraw operations.

\[ auth_{C\rightarrow A}; \left( \text{withdraw}_{C\rightarrow A}; \text{amount}_{C\rightarrow A}; \text{getAmount}_{A\rightarrow B}; \text{provAmount}_{B\rightarrow A}; \right) \]
\[ (\text{dispense}_{A\rightarrow C} + \text{overdraft}_{A\rightarrow C}) + \text{balance}_{C\rightarrow A}; \text{askAmount}_{A\rightarrow B}; \text{repAmount}_{B\rightarrow A}; \text{amount}_{A\rightarrow C})^*; \text{quit}_{C\rightarrow A} \]

A notion of conformance is then introduced as a relation among a local contract $C$, a multiparty contract $H$ and a role $R$, formalising the possibility to implement the choreography $H$ by adopting a peer following the contract $C$ to realize the role $R$. For instance, a peer following the contract

\[ auth; (\text{withdraw}; \text{amount}; (\text{dispense} + \text{overdraft}) + \text{balance}; \text{amount})^*; \text{quit} \]  

(2)

could be used to realize the client role $C$ in the choreography above. The theories for multiparty contracts formalise this notion of conformance. For instance, [Bravetti and Zavattaro 2007] introduce a notion of correctness for choreography implementations based on the following intuition: a system is correct if for every reachable state, it is always possible for the peers to reach a successful state. This notion of correctness assumes fairness, as it is always considered possible to exit from loops if this is necessary to reach a successful state. This notion of correctness allows one to check whether the parallel composition of peers following some given contracts is a good implementation of a choreography. Conformance is then defined as a refinement on contracts that preserves correctness.
A complete theory of contracts based on this approach has been proposed in [Bravetti and Zavattaro 2008b]; this theory has been subsequently extended by considering a stronger notion of correctness according to which output actions cannot wait indefinitely [Bravetti and Zavattaro 2009b], by considering asynchronous instead of synchronous communication [Bravetti and Zavattaro 2008a], and by taking a language independent approach by representing processes as labelled transition systems [Bravetti and Zavattaro 2009a]. A technique to generate contracts for peers conformant to a given choreography based on a notion of projection similar to the one studied for multiparty session types has been studied in [Lanese et al. 2008].

Alternative approaches for managing multiparty contracts have been taken, for instance, by [Bocchi et al. 2010] and [Bartoletti et al. 2012b]. These approaches are described in Section 6.2. An alternative graphical model for the specification of choreographies—collaboration diagrams—was proposed by [Bultan and Fu 2007] where the notion of realisability corresponds to the possibility to correctly implement a given collaboration diagram as a parallel composition of services.

4.4. Variations and extensions of multiparty session types

[Coppo et al. 2014] propose a simplified version of [Honda et al. 2008] where the number of session channels created upon session initiation is no longer arbitrary, but depends on the number of session roles. In particular, session initiation creates a session channel for each ordered pair of roles. [Yoshida et al. 2010] propose another variant that includes primitive recursion in order to model sessions parametric on the number of roles. [Bocchi et al. 2010] describe an approach to adding data to the theory of choreographies, together with assertions on the communicated values and invariants for recursion. This approach is further described in Section 6.2. [Deniélou and Yoshida 2011] give yet another variant of the $\pi$-calculus with multiparty sessions where it is possible for participants to join and leave a session. [Castagna et al. 2012] presents a new, streamlined language of global types that is given a trace-based semantics with features and restrictions that are semantically justified, and gives an extensive comparison with other related specification languages. [Lange and Tuosto 2012] propose type systems that make it possible (under certain conditions) to synthesise a choreography (i.e. a multiparty global type), given a collection of local session types that describe end-point behaviours (that is, local types). [Carbone and Montesi 2013] propose a choreography language with multiparty asynchronous sessions where global programs can now be mapped into the $\pi$-calculus with multiparty sessions. Finally, [Montesi and Yoshida 2013] extend [Carbone and Montesi 2013] to a calculus that supports compositionality of choreographies. The key approach consists in adding $\pi$-calculus sends and receives to the choreography language. This goes in the direction of conversation types, described in Section 4.2.

[Carbone 2009] extends the work on exceptions for binary session types (Section 4.5) to deal with multiparty interactions. The paper shows by way of an example how to project a choreography with exceptions to derive the description of single endpoints. This work has been refined by [Capecchi et al. 2010], where they provide for asynchronous exceptions that can be thrown to any subset of the participants of a multiparty session. Operators dealing with exceptions are also available in the Conversation Calculus [Vieira et al. 2008].

Multiparty session types are also used for typing MPI (Message-Passing Interface) programs. [Honda et al. 2012] extend the work on parametric multiparty session types to describe the interactions within high-performance computing (HPC) programs. This work includes primitives for expressing collective operations idiomatic in HPC programs, such as scatter, for distributing an array amongst the participants, or reduce, for computing an operation depending on values contributed by a group of participants,
as well as collective choices and loops. Traditionally, the branch and select primitives are dual and involve a participant that offers a menu of choices, from which the other chooses one. In HPC programs the idiom is different and participants choose a (same) path based only on local information gathered from previous interactions. No specific communication is needed for selecting a branch.

Building on [Honda et al. 2012], [Marques et al. 2013] introduce a dependent functional type constructor and a notion of refinement types on protocols. This way, protocols can be parametric, for instance, on the size of the problem, and restrictions can be imposed on the exchanged data. As an example, Πp : Πsize: {n: nat|n%p = 0}.scatter(0,MPI.FLOAT, size/p) denotes a protocol parametric on the number of participants (p) and on the size of the problem (size) that scatters a float array in chunks of size/p amongst its participants. For that to succeed, the size of the problem must be a multiple of the number of participants.

[Ng and Yoshida 2014] define Pabble, a protocol description language with dependent types. The language can describe an overall interaction topology designed for a variable number of participants arranged in multiple dimensions. These parameterised protocols in turn automatically generate local protocols for type checking parameterised MPI programs for communication safety and deadlock freedom. The theory underlying Pabble guarantees the termination of endpoint projection and of type checking algorithms.

5. EXTENSIONS TO TYPE THEORIES

This section introduces further extensions to the theory of behavioural types, namely subtyping and polymorphism for session types, and contract refinement.

5.1. Subtyping for session types

The first formulation of subtyping for binary session types is in [Gay and Hole 1999; Gay and Hole 2005]. The idea is to define the subtype relation on binary session types coinductively using a definition reminiscent of that of the simulation preorder for transition systems. For non-recursive session types an inductive definition in the form of inference rules suffices. In this case, the inference rules for input and output are as follows.

\[
\begin{align*}
T \leq U & \quad V \leq W \\
\Rightarrow T.V \leq U.W & \quad U \leq V \quad V \leq W \\
T.V \leq U.W & \quad U \leq T.
\end{align*}
\]

Subtyping is given the usual meaning, namely that T_1 ≤ T_2 indicates that any value of type T_1 can be safely used in a context in which a value of type T_2 is expected. Now, the context uses not T_1 but the dual of T_1, hence the contravariant/covariant inversion with respect to the lambda-calculus. In summary: input operations (?, &) are covariant, and output operations (!, ⊕) are contravariant. Continuations are always covariant.

The rules for subtyping allow a process to receive a supertype of the specified input type, which means that it can handle more general input types, and dually to actually send a subtype of the specified type (which amounts to sending a more refined value). Combining the usual subsumption rule with the typing rules for input/output of Section 3.2 one obtains the following rules.

\[
\begin{align*}
\Gamma, x: T_2 \vdash P & \quad T_1 \leq T_3 \\
\Gamma, y: T_3, x: !T_1.T_2 \vdash x'y.P & \quad \Gamma, y: T_1, x: T_3 \vdash P & \quad T_3 \leq T_1 \\
\Gamma, x: ?T_1.T_2 \vdash x'y.P & \quad \Gamma, x: ?T_1.T_2 \vdash x'y.P
\end{align*}
\]

Behavioural techniques for subtyping, based on web contracts for binary sessions (see Section 5.3), are developed in [Barbanera and de’Liguoro 2010]. This approach is aiming at a more semantic characterisation of the notion of sub-contract in a language of session behaviours, which can be understood as behavioural types expressed in a
The work on fair subtyping in [Padovani 2011] is a promising way to extend the notion of subtyping from dyadic to (higher-order) multiparty sessions and also follows an approach based on contracts. In this case, too, the approach is inspired by behavioural techniques for processes (fair testing pre-order) similar to those adopted for multiparty contracts and discussed in Section 5.3. The result is a powerful methodology for multiparty subtyping. In particular, a challenging and interesting aspect of this work is the treatment of subtyping for (potentially) infinite sessions in conjunction with a fairness guarantee. Simply put, fairness here means that liveness and termination is preserved by subtyping. A syntactic axiomatisation and algorithms to decide fair subtyping are obtained.

In the quest for more flexible compositions of processes that retain the required safety properties, another kind of subtyping has been proposed in [Mostrous and Yoshida 2009] for binary sessions, and in [Mostrous et al. 2009] for multi-party sessions. This is based on the re-ordering of communications within a session, rather than on the possibility of sending and receiving different types. For such reorderings to be meaningful, communications need to be buffered, which means that input is non-blocking (or asynchronous). Because of this, it becomes possible to send values (and choice labels) in advance of inputs (or branchings), which opens significant possibilities for optimisation. A typical situation is when a type records that a value is to be sent after one or more input actions, and the implementation does not introduce a causal dependency between the input and the subsequent outputs: this form of subtyping allows a process to send such outputs in advance, providing for more efficient communication. The intuitive idea can be understood by a simple example. A process $k?x.k'?lb.P$ that first receives on channel $k$ and then sends on $k'$, is assigned a supertype compared to $k'lb.k?x.P$ which performs the output in advance.

These ideas are developed to also deal with recursive types and algorithmic solutions to subtyping and type-checking. Several practical applications of such optimisations can be found in [Mostrous and Yoshida 2009] [Mostrous et al. 2009], including communication load optimisations for mobile code and applications to high-performance algorithms such as double-buffering.

### 5.2. Polymorphism for session types

The first study of polymorphic sessions and specifically bounded polymorphism is in [Gay 2008]. This work combines subtyping and polymorphism in the style of $F$-calculus. In particular, the usual branching and selection session types are extended with a type payload that specifies also the bounds, leading to types of the below shape, where type variables $X_i$ are bounded by types $T_i$ and may appear in the $U_i$.

$$\&\{l_i(X_i \leq T_i) : U_i\}_{i \in I} \quad \oplus \{l_i(X_i \leq T_i) : U_i\}_{i \in I}$$

The above type is assigned to terms of the shape $x \mapdot \{l_i(X_i \leq T_i) : P_i\}_{i \in I}$ for branching, and to $x \mapleft l(B).P$ for selection. As can be seen, type instantiation is “piggybacked” into selection and branching. This provides for a simpler language, avoiding additional constructs.

The above work is adapted to an object-oriented setting in [Dezani-Ciancaglini et al. 2006]. A notable aspect of this work is that choice (selection and branching) is not based on labels but rather it is guided by the class of a communicated object, which is arguably more integrated to the object paradigm. This makes bounded polymorphism more challenging (because subtyping can introduce ambiguity in the choice of a type-driven branch) and leads to a non-trivial extension of [Gay 2008], [Goto et al. 2013].
defined a polymorphic system for multiparty sessions in the style of contracts, with the distinguishing feature that type instantiation can affect multiple participants. We should also mention that logical interpretations of sessions, detailed in the next section, introduce polymorphism with universal quantification as input and existential quantification as output of a type.

5.3. Refinement for contracts

A notion corresponding to subtyping appears in the study of contracts, namely that of refinement. Using an adapted testing-based equivalence, [Castagna and Padovani 2009] provide a semantic account of how contracts can be related, in terms of the final outcome (deadlock, success or indefinite progress) of every sub-component involved in the contracts.

As in [Castagna et al. 2009b], the refinement relation defined on contracts á la Padovani 2010a allows for safe replacement of services. The refinement relation coincides with the well-known must testing preorder [De Nicola and Hennessy 1984]; this has been remarked in [Castagna et al. 2009b; Padovani 2010a], and proved in [Bernardi 2013], which also introduces refinements for clients. [Bravetti and Zavattaro 2008b] considered the impact of fairness on contract refinement, showing a reduction of the notion of contract refinement to should testing [Rensink and Vogler 2007] in place of must testing. Fairness is useful in case of infinite behaviour in which it is necessary to assume the possibility to exit from loops in order to guarantee completion. For instance, fairness needs to be considered to prove that the contract

\[
\text{auth}; (\text{withdraw}; \text{amount})^*; \text{quit}
\]

actually refines the client behaviour of contract (2) in Section 4.3. In fact, a peer following this restricted behaviour (no balance requests could be issued) can be used to implement the choreography presented in Section 4.3 because correctness continues to be guaranteed.

[Barbanera and de’Liguoro 2010] take inspiration from the work on refinement for contracts and use it to give a new account of the behavioural semantics of session types, using the notions of compliance and sub-behaviour from the work on contracts. [Bernardi and Hennessy 2013a] show that the refinement relation for servers equals the must testing preorder only if contracts are finite-state, and that the refinements for clients coincide only in languages as restricted as the finite part of session behaviours of [Barbanera and de’Liguoro 2010].

6. LOGICS

Logic and types may interact in different ways. This section introduces a linear logic interpretation of session types, and different works on the logical refinement of behavioural types.

6.1. Linear logic foundations of session types

Linearity is an important and recurring theme in concurrency and, in particular, in (behavioural) type systems for process calculi; already [Honda 1993] mentions linear logic as a source of inspiration for some aspects of session types. [Caires and Pfenning 2010] introduce a Curry-Howard style interpretation of binary session types in intuitionistic linear logic that exposes a deep correspondence between linear logic propositions and session types. The correspondence faithfully interprets the communication discipline of session-typed processes as reductions in logical derivations and conversely, as for the Curry-Howard isomorphism.
The basic correspondence between linear logic propositions and session types as described in Section 3.1 is as follows.

<table>
<thead>
<tr>
<th>Proposition</th>
<th>Session type</th>
</tr>
</thead>
<tbody>
<tr>
<td>T ⇸ S</td>
<td>T?:S</td>
</tr>
<tr>
<td>T ⊗ S</td>
<td>T!:S</td>
</tr>
<tr>
<td>S &amp; T</td>
<td>S &amp; T</td>
</tr>
<tr>
<td>S ⊕ T</td>
<td>S ⊕ T</td>
</tr>
<tr>
<td>1</td>
<td>end</td>
</tr>
<tr>
<td>!S</td>
<td>^S</td>
</tr>
</tbody>
</table>

In traditional functional interpretations of (intuitionistic) linear logic, an object of type $A ⇸ B$ is a linear function that, when given an argument of type $A$, returns a result whose type is $B$ [Girard and Lafont 1987]. In the interpretation, an object of type $x:A ⇸ B$ implements on channel $x$ a session that first receives on $x$ a session (channel) of type $A$, and afterwards behaves as $B$. Here $B$ specifies a continuation session behaviour on $x$ that somehow relies on the input session. These basic ideas can be explained by looking to the typing rules. Under the intuitionistic system processes are typed using judgements of the form

$$\Gamma; \Delta \vdash P :: z:A$$

Here $\Gamma$ and $\Delta$ are typing contexts: $\Gamma$ declares the shared channels (subject to contraction and weakening) while $\Delta$ declares the session channels (subject to the strict linear discipline). The $z:A$ on the right is a singleton typing context, declaring exactly a distinguished session. The judgement above may be naturally read as: process $P$ when composed with shared servers complying to $\Gamma$ and (open) sessions complying to $\Delta$ will safely provide a session of type $A$ at channel $z$. The rules for output and input are as follows.

$$\Gamma; \Delta \vdash P :: y:A$$  \hspace{1cm}  $$\Gamma' \vdash Q :: x:B$$

$$\Gamma; \Delta, \Delta' \vdash (\nu x) x!y.(P \mid Q) :: x:A \otimes B$$

Note that the continuation process in the output case mentions two sub-processes $P$ and $Q$, where $Q$ is the session continuation process (on $x$) while $P$ is the process that implements the session channel output in the communication (on $y$). This formulation subsumes the usual rule for output, given that $P$ can just act as a forwarder process (implementing the identity or copy-cat session), so that the rule focus on bound output, as in the internal mobility discipline introduced by [Sangiorgi 1996].

Process composition is typed by the cut-rule, which combines parallel composition and channel restriction.

$$\Gamma; \Delta \vdash P :: x:A$$  \hspace{1cm}  $$\Gamma; \Delta' \vdash Q :: T$$

$$\Gamma; \Delta, \Delta' \vdash (\nu x)(P \mid Q) :: T$$

It is useful to consider a simple example. We describe a client process that wishes to deposit money to a bank account via an ATM machine. The client does so by sending to the ATM his authentication information, after which it may then send the amount she wishes to deposit. The ATM will then send back a receipt of the operation. From the point of view of the client, the session protocol followed by the ATM can be described by the following type.

$${\text{ATMProto}} \triangleq \text{auth} \to \text{amount} \to (\text{receipt} \otimes 1)$$

Here auth, amount and receipt are types that represent shareable values of basic data types (e.g. strings and integers). If we assume that $s$ is the session channel along which
the client and the ATM interact, the following process implements the client:

$$\text{BCIntBody}_s \triangleq \text{slid}.s!n.s?r.0$$

The process above specifies a client that first sends its authentication information $id$, then the amount $n$ to be deposited and finally receives the appropriate receipt. The following judgement is derivable.

$$\cdot; s:ATMProto \vdash \text{BCIntBody}_s :: -:1$$

The ATM code is as follows:

$$\text{ATMBody}_s \triangleq s?auth.s?amt.s!rc.0$$

By composing the two processes with the cut rule, we obtain the following:

$$\cdot; \cdot \vdash (\nu s)(\text{ATMBody}_s \mid \text{BCIntBody}_s) :: -:1$$

It is also possible to develop the interpretation on top of a classical linear logic formulation [Wadler 2012; Caires et al. 2014]. The intuitionistic formulation seems particularly intuitive, notwithstanding the non-standard look of typing rules, at least when compared with traditional session type systems.

The basic interpretation can be developed in many ways, and applied in several interesting settings. [Toninho et al. 2011] enrich the basic type system based on pure linear logic with dependent types and modalities for affirmation and show how the resulting framework can express proof-carrying code certified with digital signatures in a logically motivated way. [Toninho et al. 2012] also introduce typed encodings of the simply typed lambda calculus into session typed $\pi$-calculus motivated by the linear logic interpretation. Interestingly, one of such encodings corresponds to parallel evaluation (futures). [DeYoung et al. 2012] show how a slight modification of the logical interpretation is enough to represent session typed processes with asynchronous (or buffered) communication.

[Wadler 2012] establishes a connection between the presentation of session types of [Gay and Vasconcelos 2010] and linear logic, and shows how a simple modification yields a process calculus free from deadlock; the deadlock-freedom is a consequence of the correspondence with linear logic. [Caires et al. 2013] develop a complete theory of polymorphic session types based on second order linear logic, which for the first time dissects the notion of behavioural polymorphism. Key technical results include session fidelity and global progress, and remarkably also relational parametricity, that is useful for reasoning about information hiding (in terms of hiding of local protocols). In [Toninho et al. 2013] a monadic integration of a functional language and a process language with session types is studied, allowing to express general higher-order session typed processes.

### 6.2. Logically refined session types

In the setting of binary session types, [Baltazar et al. 2012b] develop a notion of refined session types using the multiplicative linear logic as the language of refinements. The process language extends the $\pi$-calculus with $\text{assume}$ and $\text{assert}$ commands that guide the refinements, allowing for fine-grained specifications of communication protocols in which refinement formulae are seen as logical resources rather than persistent truths. This work can be seen as a generalisation of the work on type and effect systems for correspondence assertions of [Gordon and Jeffrey 2003; Bonelli et al. 2005].

In the setting of multiparty sessions, [Bocchi et al. 2010] blend the theory of choreographies with a design-by-contract approach. In particular, this approach introduces global and local types where data is explicitly added to interactions and used to specify in a suitable logic assertions on the communicated values and invariants in recursions.
The assertions are written in a classical first-order logic. The following shows an example of a global type with assertions.

\[
A \rightarrow B: \{ x: \text{int} \mid x > 0 \}
\]

\[
B \rightarrow C: \{
\begin{align*}
&\{ x \geq 5 \} \text{ge} : C \rightarrow B: \{ y: \text{int} \mid y \% 2 = 0 \}, \\
&\{ x < 5 \} \text{lt} : B \rightarrow C: \{ z: \text{bool} \mid z \iff x = 4 \}
\end{align*}
\}
\]

The three participants A, B, and C follow the protocol described by the interactions in the global type, but, unlike other approaches, interactions also establish constraints that must hold for the data that are communicated. First, A sends B a value \( x \) that must be strictly positive. Participants B and C then engage in a choice operation, governed by labels \( \text{ge} \) and \( \text{lt} \); the actual choice depends on \( x \) being greater or equal to 5. Finally, if choice \( \text{ge} \) is selected (by B), participant C sends an even number back to B, otherwise it receives from B the result of the evaluation of \( x = 4 \).

Global assertions introduce two questions in the definition of projection:

— global types cannot be projected when one of the senders is not able to fulfil its obligations because of ‘lack of information’, and
— the choices that a participant makes should not ruin later choices made by other participants.

The first of these issues can be dealt with by restricting our attention to history sensitive global types. These are types such that, for each interaction, the sending participant knows all the (free) variables found in the assertion associated with the interaction. The second issue will not occur for temporally satisfiable types. These are types such that for each possible set of values that satisfy an assertion \( \phi \) and for all assertions \( \psi \) that occur later, there exists a set of values that satisfy \( \psi \). Decidability of the assertion logic enables certain positive results. Firstly, history-sensitivity and temporal satisfiability will be decidable and preserved by the projection operation. Secondly, it is possible to validate annotated processes ([Bocchi et al. 2010] use a version of the \( \pi \)-calculus with assertions) against local assertions. Finally, well-typed annotated processes are error-free.

[Bartoletti et al. 2011] [Bartoletti et al. 2012a] use contracts at run-time to allow participants to interact. There, a participant declares its contract independently of the others and then advertises it; compatible advertised contracts can then be stipulated to form a multiparty agreement. This agreement establishes a session within which the participants of the stipulated contracts interact by performing the actions dictated by the agreement. [Bartoletti et al. 2012b] study the computational aspects of the framework in [Bartoletti et al. 2012a]. A type system for ensuring honesty has been given in [Bartoletti et al. 2013], while [Lange and Scalas 2013] give a contract model based on multi-party session types for the framework in [Bartoletti et al. 2013]. A methodology for designing and composing services such that security policies are enforced locally is given by [Bartoletti et al. 2008]. Safety properties are specified in contracts and a call-by-contract mechanism enforces them at composition time.

7. CLASSES OF BEHAVIOURAL PROPERTIES

All type systems aim to capture a specific notion of safety for programs within a particular language. For instance, the type systems in Section 6.1 guarantee progress by construction. However, the study of a particular property is sometimes the main motivation for understanding particular behavioural features.
The study of program properties usually distinguishes between safety and liveness properties. A safety property expresses that an undesirable program event will never happen during a program execution (or, equivalently, the invariant property of the undesirable event always being absent), whereas a liveness property describes that a desirable program event will occur eventually during the execution of the program.

Type systems are traditionally been well-suited for expressing safety properties; a particular challenge has been how to use behavioural types also to express liveness properties.

7.1. Channel activeness/responsiveness

In communication-centred applications such as web services or distributed protocols, it is important that every request from a client gets handled correctly by a server. Seen from the point of view of client, it is important that every valid request gets handled eventually by the server and moreover, that the client eventually obtains an answer. From the server’s point of view, it is important that whenever a request is received, the client will respect the communication protocol.

This notion has been dealt with in the literature using behavioural types. [Acciai and Boreale 2008] define the usage of a channel \( r \) to be responsive if a communication on \( r \) is guaranteed to happen eventually. [Gamboni and Ravara 2010] call this property activeness and instead define responsiveness as an additional property. According to them, a channel endpoint \( c \) is active in the process \( P \) if \( P \) is guaranteed to eventually perform an input on \( c \). The endpoint is then said to be responsive if, every time the process receives (respectively, sends) a message on that channel, it is guaranteed to be active and responsive on the message parameters in the terms specified by a channel type.

[Acciai and Boreale 2008] define a type system for guaranteeing responsiveness; the system uses a combination of techniques for deadlock and livelock avoidance together with ones used for describing linearity and receptiveness. The setting is a monadic synchronous \( \pi \)-calculus. The idea of the type system is to build a dependency graph whose vertices are responsive names of a processes. In this graph, there is an edge from name \( a \) to name \( b \) if an output action involving \( a \) is dependent upon an input action on the name \( b \). The type system then checks if the dependency graph is acyclic.

[Gamboni and Ravara 2010] work with the full synchronous polyadic \( \pi \)-calculus; in this case, the type system uses a notion of process types that specialise channel types to represent the interface between a process and its environment. The type algebra covers spatial, logical, and dynamical aspects of process types [Gamboni 2010], and the causal relations between channel usages are captured by behavioural statements embedded in process and channel types. These express the usage of channel endpoints between a process and its environment.

7.2. Capturing properties using spatial types

To achieve a proof system for weak liveness properties such as race freedom, unique receptiveness and deadlock freedom, [Acciai and Boreale 2010] describe a type system for the \( \pi \)-calculus that uses notions from spatial logic as well as a notion of behaviour. Names bound by restriction are typed with formulae from a “shallow” spatial logic (talking only about the next possible action), and processes are typed with terms from a CCS-like process calculus.

The type system allows model checking of spatial formulae. The importance of spatial logic in this setting is that the logical formulae impose constraints on the permissible spatial structure of processes; the structure of a \( \pi \)-calculus process and its type will be essentially the same. The class of properties that can be captured using this approach includes both safety properties and weak liveness properties.
7.3. Termination and deadlock freedom

The notions of termination and deadlock freedom are central in the theory of concurrent processes: non-termination is sometimes desirable—for instance we would not want an operating system to terminate—and in other settings termination must be ensured—requests in service-oriented applications should clearly be fulfilled.

[Yoshida et al. 2004; Berger et al. 2005] use the π-calculus to encode the simply typed λ-calculus. The goal is to show strong normalization for this calculus by means of the combination of a π-calculus type system that will provide a sound characterization of strong normalization and a typed version of Milner’s encoding of the λ-calculus. In this work, type judgments are of the form \( \Gamma \vdash P \triangleright A \), where \( A \) is a so-called action type. An action type should be thought of as a finite directed graph whose vertices are names annotated with an input/output polarity and whose edges describe the causal input/output dependencies between names. The underlying idea of the type system is to ensure strong normalization by ensuring that action types do not have cyclic dependencies between inputs and outputs and that inputs and outputs alternate.

[Kobayashi 1998] uses a similar notion of causality in the form of tag orderings and graph types to prove deadlock-freedom. [Kobayashi 2002] uses behavioural types very similar to session types to reason about global properties of systems, in particular lock-freedom. Termination of processes has also been tackled using conventional type systems; see e.g., [Sangiorgi 2006; Demangeon et al. 2010]. Type systems of this kind have also been employed to ensure deadlock-freedom and lock-freedom (the latter, intuitively, is the property that certain communications will eventually happen), in a series of papers by Kobayashi and coauthors, e.g., [Kobayashi 1998; Kobayashi 2005]. Different type systems may also be combined; e.g., the most powerful system for lock-freedom [Kobayashi and Sangiorgi 2010] combines those for deadlock-freedom and termination.

Another relevant application area for these types has been security; see e.g., [Haack and Jeffrey 2005] system for secrecy and authenticity.

7.4. Progress for session type systems

An important kind of liveness property is that of progress for sessions: throughout a session, every process involved will never get stuck waiting for a message that is never sent, and every message sent will eventually be received.

Session type systems can assure a local progress property within a single session, but they fall short in assuring progress when several (possibly multiparty) sessions are interleaved with each other. This follows from the fact that each session is typed in isolation, and the session type associated with a session endpoint is usually unrelated with the session types of session endpoints that are interleaved with it. More refined type systems that assure the progress property in presence of interleaved sessions are given by [Dezani-Ciancaglini et al. 2007] for synchronous binary sessions and by [Coppo et al. 2014] for asynchronous multiparty sessions. The basic idea of these type systems is to keep track of the dependencies between different sessions: a dependency \( a \prec b \) indicates that there is an input action performed on a session opened on the service name \( a \) that blocks some other action performed on sessions opened on the service name \( b \). Progress is guaranteed provided that \( \prec \) is asymmetric. This dependency-based mechanism is quite conservative and there exist practically relevant session patterns that yield circular dependencies but have progress. In particular, nested sessions, whereby all the input actions pertaining the session are completely nested within the actions of other sessions, have progress even if involved in circular dependencies. For this reason, the type system by [Coppo et al. 2014] discriminate services whose sessions are nested and tolerate circular dependencies on them without compromising progress. Identifying session dependencies and properly classifying services requires a fair amount of
type information associated with processes. [Coppo et al. 2013] provide an inference algorithm for the type system.

All the above type systems that capture progress use whole sessions as units for determining dependencies between services. This means that circular dependencies are introduced by interleaving of sessions that block each other on input actions at different stages of their evolution; such circular dependencies render many processes with progress ill typed. Another consequence is that these type systems pose very restrictive constraints on session delegation. To overcome these limits, [Padovani 2013b] and [Vieira and Vasconcelos 2013] propose more refined type systems where dependency information concerns the single actions described in session types, rather than whole sessions. Both works take inspiration from [Kobayashi 2002] type system for lock free processes. Finally, the systems described in Section 6.1 also ensures progress, while that in Section 7.2 ensures progress on the client side.

8. RELATING APPROACHES

Given the quantity and variety of approaches to behavioural types highlighted by the previous sections, an obvious question to ask is how the various notions of types are related. This includes the study of how approaches to behavioural types relate to conventional types and to communication automata.

The relation between different approaches to behavioural types has been studied by a number of authors. [Bernardi and Hennessy 2012] use (a subset of) contracts in [Castagna et al. 2009b] to define a fully abstract model of session types ordered by their sub-typing relation. [Bernardi 2013] shows that the same model can be defined in terms of must testing refinements for services and clients and extends the model to higher-order session types. While session types can be embedded in contracts, the existence of the converse embedding is still an open question.

Another issue that has been studied is whether behavioural types can be captured using conventional type systems. [Padovani 2010b] establishes a connection between the choice operators used in binary session types and intersection and union types in conventional type systems. [Demangeon and Honda 2011] show how to relate binary session types in a \( \pi \)-calculus with standard type systems for the same \( \pi \)-calculus. [Dardha et al. 2012] present a translation of session types into a simply typed polyadic \( \pi \)-calculus and show that this translation is fully abstract. The translation also allows the authors to relate [Gay and Hole 2005] subtyping for binary session types to subtyping in the \( \pi \)-calculus and to use notions of bounded polymorphism for ordinary type systems in a \( \pi \)-calculus setting to establish notions of polymorphism for binary session types. [Hüttel 2011] proposes a general type system for \( \psi \)-calculus that also makes it possible to obtain type/effect systems as instances, including a version of [Gordon and Jeffrey 2003] system for correspondence types. [Hüttel 2013] proposes a similar approach to provide a general type system for a class of resource-aware type systems including both conventional type systems for linear names [Kobayashi et al. 1999] and the action types of [Berger et al. 2005].

Communicating automata are finite state machines that communicate by exchanging messages via half-duplex channels (i.e. channels that provide communication in both directions, but only in one direction at a time). In [Gouda et al. 1984] a subclass of communicating automata composed by just 2 machines is shown to ensure deadlock-freedom and orphan message-freedom. The first results about equivalence between behavioural types and communicating automata come from [Villard 2011], where systems of communicating automata are used as contract specifications. He proved that the two-machine subclass of communicating automata characterises exactly binary session type behaviours. [Deniélou and Yoshida 2013] explore this connection further in the multiparty case. Since a generalization of the notion of half-duplex does not
work, they propose instead the definition of a multiparty compatibility property which allows for a sound and complete characterisation of the class of communicating automata that can be expressed by the language of multiparty session types [Honda et al. 2008].

9. ALGORITHMS

A central concern for type systems are the problems of type checking and type inference. The former asks whether $\Gamma \vdash P$ is decidable, given a typing context $\Gamma$ and a process $P$; the latter asks whether one can find a $\Gamma$ for given $P$ such that $\Gamma \vdash P$ holds.

Type checking is decidable for most systems here presented, even if the topic is not explicitly mentioned in the papers. One of the first works to explicit describe a type checking algorithm is [Bonelli et al. 2005]. [Giunti 2011] proposes a type checking algorithm for a version of session types for the $\pi$-calculus similar to that of [Giunti and Vasconcelos 2013].

There is as yet little work on type inference for behavioural types. [Mezzina 2008] presents an algorithm for type inference for a service calculus obtained as a simplification of SCC [Boreale et al. 2006]. [Tasistro et al. 2012] describe a polymorphic type system for binary session types without recursion or branching/selection and provide a type inference algorithm for this system. [Imai et al. 2010] describe a strategy for type inference in a binary session type system in a version of the $\pi$-calculus with branching and selection. The underlying idea is to develop a type-safe representation in Haskell of session types and to use this together with Haskell type inference.

For type systems incorporating a notion of subtyping, type checking relies on subtyping being decidable. Therefore algorithms for deciding the subtype relation are particularly relevant. In the setting of binary session types, the first work in this direction is [Gay and Hole 2005], that defines subtyping for binary session types (see Section 5.1). In this paper, algorithms for deciding subtyping and for performing type checking are proposed.

In the approach that uses processes as types, using a CCS-like type language leads to undecidability issues related to model checking and equivalence and preorder checking. This has been studied by [Hüttel et al. 2009] who show that all preorders are undecidable even for a class of CCS processes with recursive definitions and parallel composition only, i.e. without restriction or communication. For the type systems using spatial logic studied by [Acciai and Boreale 2010], the related model checking problem is undecidable even for a class of processes equivalent to Petri nets [Acciai et al. 2010].

In the setting of contract refinement, the decidability of service refinement [Padovani 2010a] follows from the decidability of the must testing preorder on finite-state processes [Cleaveland and Hennessy 1993]. The addition of the fairness assumption in the context of contracts was considered by [Bravetti and Zavattaro 2008b], where a reduction of the proposed contract refinement relation to should testing (instead of must testing) is presented; an algorithm for checking conformance in this case is obtained by composing the presented reduction to the algorithm for checking should testing [Rensink and Vogler 2007].

In the setting of multiparty sessions, [Padovani 2011] considers fair subtyping and presents a number of algorithms. First, an algorithm for deciding whether a type is viable: only viable types can occur as types of correct sessions. Second, algorithms for reducing a type to normal form and for deciding subtyping. The results are extended in [Padovani 2013a] to deal with open session types and a coarser equivalence relation, and an algorithm for deciding open fair subtyping in time $O(n^4)$ is presented.

Designing well-formed choreographies is not easy. Both the synchronization part and the data part should satisfy some conditions. For the synchronization part, [Lanese et al. 2013] propose an algorithm to enforce the conditions described in [Lanese et al. 2013].
For the data part, instead, [Bocchi et al. 2012] proposed three different algorithms for transforming inconsistent constraints on the communicated data as defined in [Bocchi et al. 2010] into consistent ones. The paper also discusses their suitability, and sketches a methodology based on the proposed algorithms.

ACKNOWLEDGMENTS

The authors would like to thank the contributions of Jakob Rehof and Bernardo Toninho. This work was supported by FCT through LaSIGE Strategic Project, ref. PEst–OE/EEI/UI0408/2014 and by the COST Action IC1201 BETTY.

REFERENCES


Foundations of Behavioural Types


Received January XXXX; revised January YYYY; accepted January ZZZZ