Session Coalgebras

Basold, Henning ; Keizer, Alex C.; Pérez, Jorge A.

Published in:
Acm transactions on programming languages and systems

DOI:
10.1145/3527633

Publication date:
2022

Citation for published version (APA):
Basold, H., Keizer, A. C., & Pérez, J. A. (2022). Session Coalgebras: A Coalgebraic View on Regular and Context-Free Session Types. *Acm transactions on programming languages and systems*, *44*(3). https://doi.org/10.1145/3527633

Session Coalgebras: A Coalgebraic View on Regular and Context-free Session Types

ALEX C. KEIZER, Master of Logic, ILLC, University of Amsterdam, The Netherlands,
HENNING BASOLD, LIACS–Leiden University, The Netherlands,
JORGE A. PÉREZ, University of Groningen, The Netherlands,

Compositional methods are central to the verification of software systems. For concurrent and communicating systems, compositional techniques based on *behavioural type systems* have received much attention. By abstracting communication protocols as types, these type systems can statically check that channels in a program interact following a certain protocol—whether messages are exchanged in the intended order.

In this article, we put on our coalgebraic spectacles to investigate *session types*, a widely studied class of behavioural type systems. We provide a syntax-free description of session-based concurrency as states of coalgebras. As a result, we rediscover type equivalence, duality, and subtyping relations in terms of canonical coinductive presentations. In turn, this coinductive presentation enables us to derive a decidable type system with subtyping for the π-calculus, in which the states of a coalgebra will serve as channel protocols. Going full circle, we exhibit a coalgebra structure on an existing session type system, and show that the relations and type system resulting from our coalgebraic perspective coincide with existing ones. We further apply to session coalgebras the coalgebraic approach to regular languages via the so-called rational fixed point, inspired by the trinity of automata, regular languages, and regular expressions with session coalgebras, rational fixed point, and session types, respectively. We establish a suitable restriction on session coalgebras that determines a similar trinity, and reveals the mismatch between usual session types and our syntax-free coalgebraic approach. Furthermore, we extend our coalgebraic approach to account for *context-free* session types, by equipping session coalgebras with a stack.

CCS Concepts: • **Theory of computation → Type theory**; **Process calculi**; Formal languages and automata theory;

Additional Key Words and Phrases: Session types, coalgebra, process calculi, coinduction

ACM Reference format:
Alex C. Keizer, Henning Basold, and Jorge A. Pérez. 2022. Session Coalgebras: A Coalgebraic View on Regular and Context-free Session Types. *ACM Trans. Program. Lang. Syst.* 44, 3, Article 18 (July 2022), 45 pages.
https://doi.org/10.1145/3527633

Jorge Pérez has been partially supported by the Dutch Research Council (NWO) under Project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software).

Authors' addresses: A. C. Keizer, Master of Logic, ILLC, University of Amsterdam, P.O. Box 94242, 1090 GE AMSTERDAM, The Netherlands; email: alex@keizer.dev; H. Basold, LIACS–Leiden University, P.O. Box 9512, 2300 RA Leiden, The Netherlands; email: h.basold@liacs.leidenuniv.nl; J. A. Pérez, Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence, University of Groningen, Nijenborgh 9, 9747 AG Groningen, The Netherlands; email: j.a.perez@rug.nl.

0164-0925/2022/07-ART18
https://doi.org/10.1145/3527633

1 INTRODUCTION

Communication protocols enable interactions between humans and computers alike, yet different scientific communities rely on different descriptions of protocols: one community may use textual descriptions, another uses diagrams, and yet another may use types. There is then a mismatch, which is fruitful and hindering at the same time. Fruitful, because different views on protocols lead to different insights and technologies. But hindering, because exactly those insights and technologies cannot be easily exchanged. With this article, we will provide a view of protocols that opens up new links between communities and that, at the same time, contributes new insights into the nature of communication protocols.

What would such a view of communication protocols be? Software systems typically consist of concurrent, interacting processes that pass messages over channels. Protocols are then a description of the possible exchanges on channels, without ever referring to the exact structure of the processes that use the channels. Since we may, for example, expect to get an answer only after sending a question, it is clear that such exchanges have to happen in an appropriate order. Therefore, protocols have to be a *state-based abstraction of communication behaviour* on channels. Because *coalgebras* provide an abstraction of general state-based behaviour, our proposed view of communication protocols becomes: model the states of a protocol as states of a coalgebra and let the coalgebra govern the exchanges that may happen at each state of the protocol.

The above view of protocols allows us to model protocols as coalgebras. However, protocols are usually not studied for the sake of their description but to achieve certain goals: ensuring correct composition of processes, comparing communication behaviour, or refining and abstracting protocols. *Session types* [29, 30] are an approach to communication correctness for processes that pass messages along channels. The idea is simple: describe a protocol as a syntactic object (a type), and use a type system to statically verify that processes adhere to the protocol. This syntactic approach allows the automatic and efficient verification of many correctness properties. However, the syntactic approach depends on choosing *one particular representation of protocols* and *one particular representation of processes*. We show in this article that our coalgebraic view of protocols can guarantee correct process composition, and allows us to reason about key notions in the world of session types, *type equivalence*, *duality* and *subtyping*, while being completely independent of protocol and process representations.

Our coalgebraic view is best understood by following the distillation process of ideas on a concrete session type system by Vasconcelos [61]. Consider the session type $S = \texttt{?int}.\ \texttt{!bool}.\ \mathsf{end}$, which specifies the protocol on one endpoint of a channel that receives an integer, then outputs a Boolean, and finally terminates the interaction. Note that the protocol S specifies three different states: an input state, an output state, and a final state. Moreover, we note that S specifies only how the channel is seen from one endpoint; the other endpoint needs to use the channel with the *dual* protocol $\texttt{!int}.\ \texttt{?bool}.\ \mathsf{end}$. Thus, session type systems ensure that the states of S are enabled only in the specified order and that the two channel endpoints implement dual protocols.

A state-based reading of session types is intuitive and is already present in programming concepts such as typestates [22, 55, 56], theories of behavioural contracts [9, 11, 12, 19], and connections between session types and communicating automata [16, 40]. The novelty and insight of the coalgebraic view is that (1) it describes the state-based behaviour of protocols underlying session types, supporting unrestricted types and delegation, without adhering to any specific syntax or target programming model; (2) it offers a general framework in which key notions such as type equivalence, duality, and subtyping arise as instances of well-known coinductive constructions; and (3) it allows us to derive type systems for specific process languages, like the π-calculus.

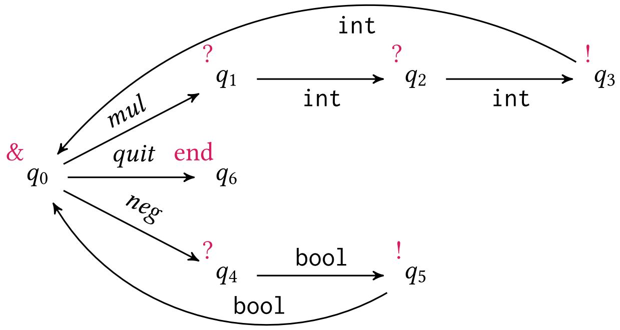


Fig. 1. Protocol of the mathematical server as a session coalgebra.

Session Coalgebras at Work. How does this coalgebraic view of protocols work for session types? Consider a "mathematical server" that offers three operations to clients: integer multiplication, Boolean negation, and quitting. The following session type T specifies a protocol to communicate with this server:

$$T = \mu X.\ \&\begin{cases} \mathit{mul}: & ?\mathtt{int}.\ ?\mathtt{int}.\ !\mathtt{int}.\ X \\ \mathit{neg}: & ?\mathtt{bool}.\ !\mathtt{bool}.\ X \\ \mathit{quit}: & \mathrm{end} \end{cases}$$

T is a recursive protocol, as indicated by "$\mu X.$," which can be repeated. A client can choose, as indicated by &, between the three operations (*mul*, *neg*, and *quit*) and the protocol then continues with the corresponding actions. For instance, after choosing *mul*, the server requests two integers and, once received, promises to send an integer over the channel. We can see states of the protocol T emerging, and it remains to provide a coalgebraic view on the actions of the protocol to obtain what we will call *session coalgebras*.

Figure 1 depicts a session coalgebra that describes protocol T. It consists of states $q_0, \ldots, q_6$, each representing a different state of T, and transitions between these states to model the evolution of T. Meaning is given to the different states and transitions through the labels on the states and transitions. The state *labels*, written in red at top-left of the state name, indicate the branching type of that state. Depending on the branching type, the labels of the transitions bear different meanings. For instance, q_0 is labelled with "&," which indicates that this state initiates an external choice. The labels on the three outgoing transitions for q_0 (*mul*, *neg*, *quit*) correspond then to the possible kinds of message for selecting one of the branches. Continuing, states $q_1, \ldots, q_5$ are labelled with a request for data (label ?) or the sending of data (label !), and the outgoing transition labels indicate the type of the exchanged values (e.g., `bool`). Finally, state q_6 decrees the end of the protocol (label end). Note that the cyclic character of T occurs as transitions back to q_0; there is no need for an explicit operator to capture recursion.

A session coalgebra models the view on one channel endpoint, but to correctly execute a protocol we also need to consider the *dual* session coalgebra that models the other endpoint's view. In our example, the dual of Figure 1 is given by the diagram in Figure 2, which concerns states $s_0, \ldots, s_6$. More precisely, the *states* q_i and s_i are pairwise dual in the following sense. The external choice of q_0 becomes an *internal choice* for s_0, expressed through the label $\oplus$, with exactly the same labels on the transitions leaving s_0. This means that whenever the server's protocol is in state q_0 and the client's protocol in state s_0, then the client can choose to send one of the three signals to the server, thereby forcing the server protocol to advance to the corresponding state. All other states turn from sending states into receiving states and vice versa. We will see that this *duality*

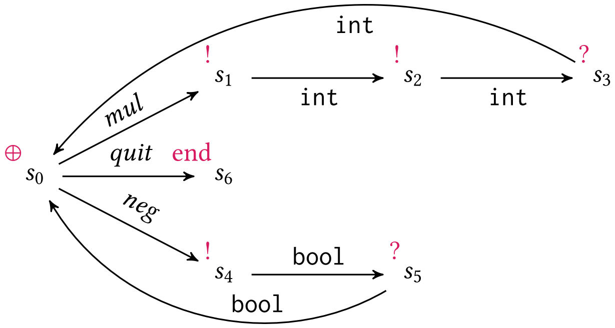


Fig. 2. Session coalgebra for the client view protocol the of mathematical server.

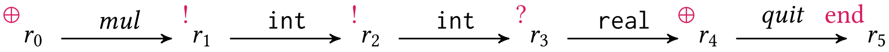


Fig. 3. Session coalgebra that uses only part of a mathematical server.

relation between states of session coalgebras has a natural coinductive description that can be obtained with the same techniques as bisimilarity. The duality relation for T will give us then the full picture of the intended protocol.

Suppose a client who would only want to use multiplication once but could also handle real numbers as inputs. Such a client had to follow the protocol given by the session coalgebra in Figure 3, with states $r_0, \ldots, r_5$.

In theories of session types, the protocol of Figure 2 is a *subtype* of the protocol in Figure 3 (cf. References [23, 24]). Concretely, this new client can also follow the subtype protocol, and can thus communicate with a server following the protocol of Figure 1. For session coalgebras, we recover the same notion of subtyping by using specific *simulation* relations that allow us to prove that the behaviour of r_0 can be simulated by s_0. Together, simulations and duality provide the foundation of typical session type systems.

We have used thus far session types and coalgebras for protocols with exchanges of simple data values. In contrast, rich session type systems can regulate *session delegation*, the dynamic allocation and exchange of channels by processes. Imagine a process that creates a channel, which should adhere to some protocol T. From an abstract perspective, the process holds both endpoints of the new channel, and has to send one of them to the process it wishes to communicate with. To ensure statically that the receiving process respects the protocol of this new channel, we need to announce this communication as a transmission of the session type T (via an existing channel) and use T to verify the receiving process. Session delegation adds expressiveness and flexibility, but it may cause problems in the characterisation of a correct notion of duality [25]. Remarkably, our coalgebraic view of session types makes this characterisation completely natural.

As an example, consider the type $T = \mu X.\ ?X.\ X$, which models a channel endpoint that infinitely often receives channel ends of its own type T. To obtain the dual of T, we may naïvely try to replace the receive with a send, which results in the type $\mu X.\ !X.\ X$. The problem is that the two channel endpoints would not agree on the type they are sending or receiving, as any dual type of T needs to send messages of type T. Thus, the correct dual of T would be the type $U = \mu X.\ !T.\ X$. Both T and U specify the transmission of non-basic types, either the recursion variable X or T, in contrast to the mathematical server that merely stipulated the transmission of basic data values (integers or Booleans).

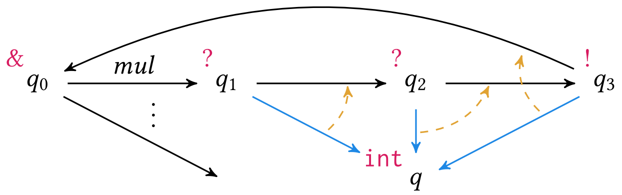


Fig. 4. Protocol of mathematical server as session coalgebra.

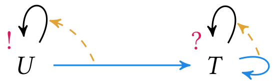


Fig. 5. Session coalgebra for a recursive type T and its dual U.

In our session coalgebras for the mathematical server it sufficed to have simple data types and branching labels on transitions. However, to represent T and U, we will need another mechanism to express session delegation. We observe that a transmission in session types consists of the transmitted data and the session type that the protocol must continue with afterward. Thus, a transition out of a transmitting state in a session coalgebra encompasses both a *data transition* and a *continuation transition*. In diagrams of session coalgebras, we indicate the data transition by a coloured arrow $\longrightarrow$ and an arrow ⇢ connecting the data to the continuation transition. Using the combined transitions, Figure 4 redraws the multiplication part of the mathematical server in Figure 1.

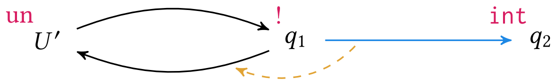
This way, the transition $q_1 \xrightarrow{\texttt{int}} q_2$ has been replaced by *both* a data transition into a new state q *and* a continuation transition into q_2. Moreover, q has been declared as a *data state* that expects an integer to be exchanged (label `int`).

Having added these transitions to our toolbox, we can present the two types T and U as session coalgebras. The diagram in Figure 5 shows such a session coalgebra, in which we name the states suggestively T and U.

Using this presentation as session coalgebras, it is now straightforward to *coinductively* prove that the states T and U are dual: (1) the states have opposite actions; (2) their data transitions point to equal types; and (3) their continuations are dual by coinduction. Clearly, the last step needs some justification but it will turn out that we can appeal to a standard definition of coinduction in terms of greatest fixed points. This demonstrates that our coalgebraic view on session types makes the definition of duality truly natural and straightforward.

Up to here, we have discussed session types and coalgebras that are *linear*, i.e., they enforce that protocols complete exactly once. In many situations, one also needs *unrestricted* types, which enable sharing of channels between processes that access these channels concurrently. This is the case of a process that offers a service for other processes, for instance a web server. Session delegation allows us to create dynamically channels and check their protocols, but the shared channel for *initiating a session* [24] has to offer its protocol to an arbitrary number of clients. Unrestricted types enable us to specify these kind of service offers.

As an example, consider a process that provides a channel for communicating integers to anyone asking, like a town hall official handing out citizen numbers. The session type $U' = \mu X.\ \texttt{un}\,!\texttt{int}.\ X$ represents the corresponding protocol, where "un" qualifies the type $!\texttt{int}.\ X$ as unrestricted. This allows the process holding the end of a channel with type U' to transmit an integer to any process that is connected to the shared channel, without any restriction on their number. Using this, it is surprisingly simple to express U' in our coalgebraic view: We introduce a new state label "un" (unrestricted). The idea is that states reachable from an un-labelled state can be used arbitrarily as protocols across different processes connecting to a channel that follows the protocol given by those states. The following diagram shows a session coalgebra with a state that corresponds to U':



Contributions. In this article, we make the following contributions.

(1) We introduce the notion of *session coalgebra*, which justifies the state-based behaviour of session types from a coalgebraic perspective. Using coalgebra as a unifying framework for session types has two advantages:
   - session coalgebras can be defined and studied independently from specific syntactic formulations, while keeping the operational behaviour of session types; and
   - we can uncover the innate *coinductive* nature of key notions in session types, such as duality, subtyping, and type equivalence through standard coalgebraic techniques.

(2) To enable the verification of processes against protocols described by session coalgebras, we also contribute a *type system* for π-calculus processes, in which channel types are given by states of an *arbitrary* session coalgebra. Exploiting our coalgebraic perspective, our type system supports subtyping by combining elements from type systems by Gay and Hole [24] and by Vasconcelos [61].

(3) We provide a *type checking algorithm* for that system, provided that the underlying session coalgebra fulfils two intuitive conditions. In doing so, we show how a specific type syntax can be equipped with a session coalgebra structure and how the two decidability conditions are reflected in the type system.

(4) Finally, we show how the coalgebraic approach naturally extends to the more expressive *context-free* session types by Thiemann et al. [58], by equipping session coalgebras with a stack, which allows the finitary presentation of session types with sequential composition.

Related Work. To our knowledge, a coalgebraic justification to session types is novel, although specific state-based description of protocols have been considered before [9, 11, 12, 15, 16, 19, 22, 40, 55, 56]. In particular, although communicating automata can also provide syntax-independent characterisations of session types (see, e.g., References [16, 17]), such characterisations do not support delegation, an expressive feature that is cleanly justified in our coalgebraic approach.

Coinduction already has been exploited in the definition of type equivalence [58], subtyping [23, 24] and, especially, duality for systems with recursive types [7, 25, 39]. Unlike ours, these previous definitions are language-dependent, as they are tailored to specific process languages and/or syntactic variants of the type discipline. Session coalgebras enable thus the generalisation of insights and technologies from specific languages to any protocol specification that fits under the umbrella of state-based sessions.

We show how a specific type syntax can be equipped with an appropriate coalgebraic structure. This is in contrast to starting with a specific type syntax and then employing category theoretical ideas, as done in Reference [59], where coinductive session types are encoded in a session type system with parametric polymorphism [10].

Organisation. Throughout the article, we will turn the ideas sketched above into a coalgebraic framework. We introduce in Section 2 a concrete session type syntax that we will use as illustration of our framework. In Section 3, we will define session coalgebras as coalgebras for an appropriate functor and show that the type system from Section 2 can be equipped with a coalgebraic structure. The promised coinductive view on type equivalence, duality, subtyping, and so on, will

$$\begin{array}{rcl@{\qquad\qquad}rcl} p & ::= & ?T.\,S & S & ::= & \mathsf{end} \\ & | & !T.\,S & & | & q\,p \\ & | & \&\{l_i : S_i\}_{i\in I} & & | & X \in \mathsf{Var} \\ & | & \oplus\{l_i : S_i\}_{i\in I} & & | & \mu X.\,S \\ \\ q & ::= & \mathsf{lin} \mid \mathsf{un} & T & ::= & S \mid d \in D \end{array}$$

Fig. 6. Session types over sets of basic data types D and of variables Var.

be provided in Section 4. Moreover, we will show that these notions are decidable under certain conditions that hold for any reasonable session type syntax, including the one from Section 2. Up to that point, the session coalgebras only had intrinsic meaning and were not associated to any process representation. Section 5 sets forth a type system for π-calculus, in which channels are assigned states of a session coalgebra as types. The resulting type system features subtyping and algorithmic type checking, presented in Section 6. In Section 7, we illustrate how to expand the coalgebraic presentation to account for context-free session types. Some final thoughts are gathered in Section 8.

This article is an extended version of the conference paper [34], which goes into more detail regarding proofs and collects additional new material on context-free session types.

2 SESSION TYPES

To motivate the development of session coalgebras, we recall in this section the concrete syntax of an existing session type system by Vasconcelos [61]. After building up our intuition, we introduce session coalgebras in Section 3 to show they can represent this concrete type system.

The types of the system that we will be using are generated by the grammar in Figure 6, relative to a set of basic data types D and a countable set of type variables Var. This grammar has four syntactic categories: pretypes p, qualifiers q, session types S, and types T, which contains both session and basic types. A pretype p is simply a communication action: send (!), receive (?), external choice (&), and internal choice ($\oplus$) indexed by a finite sets I of labels, followed by one or multiple session types. The simplest types are basic data types in D and the completed, or terminated, protocol represented by end. A pretype and qualifier also form a session type, written as $q\,p$. The "lin" qualifier enforces that the communication action p has to be carried out exactly once, while the "un" qualifier allows arbitrary use of p. Finally, we can form recursive session types with the fixed point operator μ and the use of type variables. We use the usual notion of α-equivalence, (capture-avoiding) substitution, and free and bound type variables for session types.

The grammar allows arbitrary recursive types. We let Type be the set of all T in which recursive types are *contractive* and *closed*, which means that they contain no substrings of the form $\mu X_1.\mu X_2 \dots \mu X_n.X_1$ and no free type variables. In a slight abuse of notation, we will use T to range over elements of Type from here on.

To lighten up notation, we will usually omit the qualifier lin and assume every type to finalise with end. With these conventions, we write, e.g., `?int.` instead of lin `?int.` end and un `?int.` for a single unrestricted receive.

We assume there is some decidable subtyping preorder $\leq_D$ over the basic types. A type is a subtype of another if the subtype can be used anywhere where the supertype was accepted. In examples, we use the basic types `int`, `real`, and `bool`, and we assume that `int` is a subtype of `real`, as usual.

An important notion is the *unfolding* of a session type, which we define next:

Definition 2.1 (Unfolding). The unfolding of a recursive type $\mu X.S$ is defined recursively

$$\mathit{unfold}(\mu X.S) = \mathit{unfold}(S[\mu X.S/X])$$

For all other T in Type, *unfold* is the identity: $\mathit{unfold}(T) = T$.

Because we assume that types are contractive, $\mathit{unfold}(T)$ terminates for all T. Also, because all types are required to be closed, $\mathit{unfold}(T)$ can never be a variable X. Any such variable would have to be bound somewhere before use, meaning it would have been substituted. Furthermore, unfolding a closed type always yields another closed type, as each removed binder always causes a substitution of the bound variable.

3 SESSION COALGEBRA

Here, we will discuss *session coalgebras*, the main contribution of this article. The idea is that session coalgebras will be coalgebras for a specific functor F, which will capture the state labels and the various kinds of transitions that we discussed in Section 1. An important feature of coalgebras in general, and session coalgebras in particular, is that the states can be given by an arbitrary set. We will leverage this to define a session coalgebra on the set of types Type introduced in Section 2.

3.1 Preliminaries

Before coming to the definition, let us briefly recall some notions of category theory [4, 8, 47]. We will not require a lot of category theoretical terminology; in fact, we will only use the category **Set** of sets and functions. Moreover, we will be dealing with *functors* $F\colon \mathbf{Set} \to \mathbf{Set}$ on the category **Set**. Such a functor allows us to map a set X to a set $F(X)$, and functions $f : X \to Y$ to functions $F(f) : F(X) \to F(Y)$. A functor must preserve identity and compositions, that is, F maps the identity function $\mathrm{id}_X\colon X \to X$ on X to the identity on $F(X)$: $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$; and, given functions $f\colon X \to Y$ and $g\colon Y \to Z$, we must have $F(g \circ f) = F(g) \circ F(f)$.

A central notion is that of the coalgebras [33, 49, 50] for a functor F. A *coalgebra* is given by a pair (X, c) of a set X and a function $c\colon X \to F(X)$. For simplicity, we often leave out X and refer to c as the coalgebra. The general idea is that the set X is the set of *states* and that c assigns to every state its one-step behaviour. In the case of session coalgebras this will be the state labels and outgoing transitions. Given two coalgebras $c\colon X \to F(X)$ and $d\colon Y \to F(Y)$, we say that $h\colon X \to Y$ is a *homomorphism*, if $d \circ h = F(h) \circ c$. Coalgebras and their homomorphisms form a category $\mathbf{CoAlg}(F)$, with the same identity maps and composition as in **Set**.

3.2 Session Coalgebras

Coming to the concrete case of session coalgebras, we now construct a functor that allows us to capture the state labels and the different kinds of transitions. Keeping in mind that states of a session coalgebra correspond to states of a protocol, we need to be able to label the states with enabled *operations*. Depending on their operations, states may also have *polarities*.

Definition 3.1 (Operations and Polarities). We let O be the set of all operations $O = \{\mathrm{com}, \mathrm{branch}, \mathrm{end}, \mathrm{bsc}, \mathrm{un}\}$ and P the set of polarities $P = \{\mathrm{in}, \mathrm{out}\}$.

The operation of a state describes the action it represents: “com” marks the transmission (sending or receiving) of a value; “branch” marks an (internal or external) choice; “end” marks the completed protocol; “bsc” marks a basic data type; and “un” marks an unrestricted type. States that transmit data (labelled with “com”) or allow for choice (labelled with “branch”) also have a polarity, which can be either “in” (a receiving action or external choice) or “out” (a sending action or internal choice).

Note that pairs in $\{\text{com}, \text{branch}\} \times P$ correspond directly to the actions that can appear in session types: ? = (com, in), ! = (com, out), & = (branch, in), and $\oplus$ = (branch, out). We will be using these markers to abbreviate the pairs.

Now that we have the possible operations of a protocol, we need to define the transitions that may follow each operation. Recall that the transition at a choice state has to be labelled with messages that resolve that choice. We therefore assume to be given a set $\mathbb{L}$ of possible choice labels. The variable l will be used to refer to an element of $\mathbb{L}$. Then, $\mathcal{P}^{+}_{<\aleph_0}(\mathbb{L})$ is the set of all finite, non-empty, subsets of $\mathbb{L}$. Variables $L, L_1, L_2, \ldots$ refer to these finite, non-empty subsets of $\mathbb{L}$.

Our goal is to define a *polynomial functor* [21] that captures the states labels and transitions. This requires some further formal language. First, given sets X and Y, we denote by X^Y the set of all (total) functions from Y to X. Second, given a family of sets $\{X_i\}_{i \in I}$ indexed by some set I, their *coproduct* is the set $\coprod_{i \in I} X_i = \{(i, x) \mid i \in I, x \in X_i\}$.

We are now ready to define session coalgebras:

Definition 3.2 (Session Coalgebras). Let $*$ and $\mathcal{d}$ be some fixed distinct elements, and let A and B_a be sets defined as follows, where $a \in A$. Recall that D is the set of all basic data types:

$$\begin{array}{rlrl} A = & \{\text{com}\} \times P & B_{\text{com},p} &= \{*, \mathcal{d}\} \\ & \cup\ \{\text{branch}\} \times P \times \mathcal{P}^{+}_{<\aleph_0}(\mathbb{L}) & B_{\text{branch},p,L} &= L \\ & \cup\ \{\text{end}\} & B_{\text{end}} &= \emptyset \\ & \cup\ \{\text{bsc}\} \times D & B_{\text{bsc},d} &= \emptyset \\ & \cup\ \{\text{un}\} & B_{\text{un}} &= \{*\} \end{array}$$

The polynomial functor $F : \mathbf{Set} \longrightarrow \mathbf{Set}$ is defined by

$$F(X) = \coprod_{a \in A} X^{B_a},$$

$$F(f)(a, g) = (a, f \circ g).$$

A coalgebra (X, c) for the functor F is called a *session coalgebra*.

Let us unfold this definition. Given a session coalgebra $c : X \longrightarrow F(X)$ and a state $x \in X$, we find in $c(x) \in F(X)$ the information of x encoded as a tuple (a, f) with $a \in A$ and $f : B_a \longrightarrow X$. From a, we get directly the operation, and the polarity for com states, the type of values communicated for bsc states or the message labels of branch states. The function f encodes the transitions out of x. The domain of f is exactly the set of labels that have a transition, and it is dependent on the kind of state declared by a. If f has a domain of cardinality two or more, then one can think of the state x "splitting" into states that will be visited one after another. This idea corresponds to the intuition expressed in the diagrams of session coalgebras above, and has been previously used to model processes with binary "splitting" and communication [35, 52]. Here, we are not so much interested in the coalgebraic semantics [27] of session types; rather, we aim to define a coalgebraic view on them that allows the derivation of everything a type system for processes needs.

It is convenient to partition the domain of the transition map f into data and continuations. Notice how only com states have data transitions, for other states, all transitions are continuations. As usual, we write $\text{dom}(f)$ for the domain of f.

Definition 3.3. Suppose $c(x) = (\text{com}, p, f)$, then the *data domain* of f is $dom_D(f) = \{\mathcal{d}\}$ and the *continuation domain* is $dom_C(f) = \{*\}$. In all other cases, $dom_D(f) = \emptyset$ and $dom_C(f) = dom(f)$.

We will have to analyse subsets of coalgebras that are closed under transitions. Given a session coalgebra $c : X \longrightarrow F(X)$, we say that $d : Y \longrightarrow F(Y)$ with $Y \subseteq X$ is a *subcoalgebra* of c if the inclusion map $Y \to X$ is a coalgebra homomorphism. If F weakly preserves pullbacks, then $c(Y) \subseteq F(Y)$ and thus d is the restriction of c to Y [49, Theorem 6.3]. As all functors that we consider are weakly pullback preserving, we will also refer to Y as subcoalgebra. The subcoalgebra *generated by* $x \in X$ in c, denoted by $\langle x \rangle_c$, is the least subset of X that contains x and is a subcoalgebra of c [49]. Intuitively, it is the set that includes x and all states that are reachable from x.

3.3 An Alternative Presentation of Session Coalgebras

Session coalgebras (X, c) are rather complex. We show how to build up c as the combination of two simpler functions, denoted σ and δ, so that $c(x) = (\sigma(x), \delta(x))$ with $\sigma \colon X \to A$ and $\delta(x) \colon B_{\sigma(x)} \to X$. Observe that every state gets an operation in O assigned, thus we may assume that there is a map $\mathrm{op} \colon X \to O$. Depending on the operation given by $\mathrm{op}(x)$, the label on x will then have different other ingredients that are captured in the following proposition.

To formulate the proposition, we need some notation. Suppose $f \colon X \to I$ is a map and $i \in I$. We define the *fibre* X_i^f of f over i to be $X_i^f = \{x \in X \mid f(x) = i\}$. Moreover, we let the *pairing of functions* f and g be $\langle f, g\rangle(x) = (f(x), g(x))$.

Proposition 3.4. *A session coalgebra* (X, c) *can equivalently be expressed by providing the following maps:*

$$\begin{array}{lll} op & : X \to O & \textit{maps each state to an operation} \\ pol & : X^{op}_{\mathrm{com}} + X^{op}_{\mathrm{branch}} \to P & \textit{maps}\ \mathrm{com}\ \textit{and}\ \mathrm{branch}\ \textit{states to a polarity} \\ la & : X^{op}_{\mathrm{branch}} \to \mathcal{P}^{+}_{<\aleph_0}(\mathbb{L}) & \textit{maps}\ \mathrm{branch}\ \textit{states to a set of labels} \\ da & : X^{op}_{\mathrm{bsc}} \to D & \textit{maps}\ \mathrm{bsc}\ \textit{states to their basic type} \\ \delta_a & : X^{\sigma}_{a} \to X^{B_a} & \textit{maps each state to a transition function,} \end{array}$$

where

$$\sigma(x) = \begin{cases} \langle op, pol\rangle(x) & \textit{if } op(x) = \mathrm{com}, \\ \langle op, pol, la\rangle(x) & \textit{if } op(x) = \mathrm{branch}, \\ \langle op, da\rangle(x) & \textit{if } op(x) = \mathrm{bsc}, \\ op(x) & \textit{if } op(x) = \mathrm{end} \textit{ or } op(x) = \mathrm{un}. \end{cases}$$

We specified δ_a as a family of transition functions so we can express the necessary constraints in terms of the codomains. We can define a single global transition function as $\delta(x) = \delta_{\sigma(x)}(x)$. This is how the coalgebra finally becomes $c(x) = (\sigma(x), \delta(x))$. As long as the provided functions map the appropriate elements of X to the specified codomains, this derived function will conform to $c : X \to F(X)$.

The procedure also works backward: given any session coalgebra, we can derive functions $\mathrm{op}(x)$, $\mathrm{pol}(x)$, and so on, from $c(x)$. We will often use $\mathrm{op}(x)$, $\sigma(x)$, and $\delta(x)$ to refer to those specific parts of an arbitrary session coalgebra.

3.4 Coalgebra of Session Types

In Section 1, we informally explained how session types can be represented as states of a session coalgebra. We will now justify this claim by showing that session types are, in fact, states of a specific session coalgebra $(\mathrm{Type}, c_{\mathrm{Type}})$.

We define the functions op, pol, δ, and la (see Proposition 3.4) on Type. Using Proposition 3.4, we can then derive $c_{\text{Type}} : \text{Type} \to F(\text{Type})$. Let us begin with the linear types:

T	$c_{\text{Type}}(T)$			
	op(T)	pol(T)	$\delta(T)$	la(T)
$\text{lin}\,?T.\ T'$	com	*in*	$\delta(T)(*) = T'$ $\delta(T)(\mathcal{d}) = T$	
$\text{lin}\,!T.\ T'$		*out*		
$\text{lin}\&\{l_i : T_i\}_{i \in I}$	branch	*in*	$\delta(T)(l_i) = T_i$	$\{l_i \mid i \in I\}$
$\text{lin}\oplus\{l_i : T_i\}_{i \in I}$		*out*		

Under this definition, la(T) is finite, by virtue of an expression being a finite string. The completed protocol end and basic types d are straightforward: $c(\text{end}) = (\text{end}, f_\emptyset)$ and $c(d) = (\text{bsc}, d, f_\emptyset)$ for any $d \in D$, where $f_\emptyset \colon \emptyset \to X$ is the empty function. Recursive types are handled according to their unfolding, $c(\mu X.\ T) = c(\mathit{unfold}(\mu X.\ T))$. Recall that contractivity ensures that *unfold* always terminates. As our types are closed, all recursion variables are substituted during the unfolding of their binder. Consequently, we do not need to define c on these variables. Also note that this definition results in an *equi-recursive* interpretation of recursive types, i.e., a recursive type and its unfolding are considered as equal.

Session types can also be unrestricted, and consist of a pretype p with a qualifier un. Session coalgebras have un states to mark unrestricted types; the continuation describes what the actual interaction is. Thus, we define $\text{op}(\text{un}\ p) = \text{un}$ and $\delta(\text{un}\ p)(*) = \text{lin}\ p$.

Remark 1 (Alternative Syntaxes and their Functors). The unrestricted session types that we have adopted are fairly standard, but they are not the only ones in the literature. Most notably, Gay and Hole [24] defined a type $\rceil[T_1, \ldots, T_n]$ that allows infinite reading *and* writing. To allow for such behaviour in session coalgebra, we can change B_{un} to a set of two elements, such a $\{*_1, *_2\}$. Like internal choice, the two transitions describe an option of which behaviour to follow, but without sending synchronisation signals. One transition could go to a read, and the other to a write, both recursively continuing as the original type $\rceil[T_1, \ldots, T_n]$.

It is possible, although not entirely trivial, to change the further definitions appropriately and get a decidable type checking algorithm encompassing both the syntax presented in this work, and Gay and Hole's syntax. We choose not to, so to keep the presentation simpler.

4 TYPE EQUIVALENCE, DUALITY, AND SUBTYPING

Up to here, we have represented session types as session coalgebras, but we have not yet given a precise semantics to them. As a first step, we will define three relations on states: *bisimulation*, *duality*, and *simulation*. Bisimulation is also called behavioural equivalence for types; we will show that bisimilar types are indeed equivalent. Duality specifies complementary types: it tells us which types can form a correct interaction. Simulation will provide a notion of subtyping: it tells us when a type can be used where another type was expected. Besides relations on session coalgebras, we also introduce the *parallelizability* of states that allows us to rule out certain troubling unrestricted types. Finally, we will obtain conditions on coalgebras to ensure the decidability of the three relations and therefore the type system that we derive in Section 5.

In the following, we will denote by Rel_X the poset $\mathcal{P}(X \times X)$ of all relations on X ordered by inclusion. Recall that a post-fixpoint of a monotone map $g \colon \text{Rel}_X \to \text{Rel}_X$ is a relation $R \in \text{Rel}_X$

with $R \subseteq g(R)$. Note that Rel_X is a complete lattice and that therefore any monotone map has a greatest post-fixpoint, which is also the greatest fixpoint, by the Knaster-Tarski Theorem [57].

We will define bisimulation, simulation, and duality as the greatest (post-)fixpoint of monotone functions, which we will therefore call *coinductive definitions*. This definition turns out to be intuitively what we would expect and the interaction of infinite behaviour with other type features is automatically correct. The coinductive definitions also give us immediately proof techniques for equivalence, duality, and subtyping: to show that two states are, say, dual we only have to establish a relation that relates the two states and show that it is a post-fixpoint. This technique can then be enhanced in various ways [48], and we will show that it is decidable for reasonable session coalgebras.

4.1 Bisimulation

Two states of a coalgebra are said to be *bisimilar* if they exhibit equivalent behaviour. We abstract away from the precise structure of a coalgebra and only consider its observable behaviour. Two states of a session coalgebra are bisimilar if their labels are equal and if the states at the end of matching transitions are again bisimilar. There is one exception to the equality of labels: basic types can be related via their pre-order, which does not have to coincide with equality.

Definition 4.1. Given some coalgebra (X, c), we define $c^* : \mathrm{Rel}_{F(X)} \to \mathrm{Rel}_X$, the binary preimage of c, as follows:

$$c^*(R) = \{(x, y) \mid (c(x), c(y)) \in R\}\,.$$

Definition 4.2. We define the function $f_\sim : \mathrm{Rel}_X \to \mathrm{Rel}_{F(X)}$ as

$$\begin{aligned} f_\sim(R) = {} & \{((a, f), (a, f')) \mid (\forall \alpha \in \mathit{dom}(f)) \quad f(\alpha)\, R\, f'(\alpha)\} \\ & \cup \{((\mathrm{bsc}, d, f_\emptyset), (\mathrm{bsc}, d', f_\emptyset)) \mid d \leq_D d' \wedge d' \leq_D d\}, \end{aligned}$$

where $f_\emptyset \colon \emptyset \to X$ is the empty function.

It can be easily checked that c^* and $f_\sim$ are both monotone maps and thus also their composition. Thus, the greatest fixpoint in the following definition exists.

Definition 4.3 (Bisimilarity). Let $g_\sim = c^* \circ f_\sim$. A relation R is called a bisimulation if it is a post-fixpoint of $g_\sim$. We call the greatest fixpoint of $g_\sim$ *bisimilarity* and denote it by $\sim$.

It should be noted that $f_\sim$ is not the canonical relation lifting typically considered to construct bisimilarity as coinductive relation [20, 28, 54]. The reason is simply that we only assume the data types D to form a preorder, and thus we need to check the order in both directions. The map $f_\sim$ would be the canonical relation lifting if we turned D into a poset.

4.2 Duality

Duality describes exactly opposite types in terms of their polarity. That is, the dual of input is output and the dual of output is input: $\overline{in} = out$ and $\overline{out} = in$. We can extend this to tuples a in A, see Definition 3.2, with the exception of basic types, because they do not describe channels:

$$\begin{aligned} \overline{(\mathrm{com}, p)} &= (\mathrm{com}, \overline{p}) & \overline{(\mathrm{end})} &= (\mathrm{end}) \\ \overline{(\mathrm{branch}, p, L)} &= (\mathrm{branch}, \overline{p}, L) & \overline{(\mathrm{un})} &= (\mathrm{un}) \\ \overline{(\mathrm{bsc}, d)} & \text{ is undefined} \end{aligned}$$

The next step is to compare transitions. Continuations of $\mathit{dom}_C(f)$ need to be dual. The data types that are sent or received need to be equivalent, hence transitions of $\mathit{dom}_D(f)$ need to go to bisimilar states. We capture this idea with the monotone map $f_\perp : \mathrm{Rel}_X \to \mathrm{Rel}_{F(X)}$ defined as follows:

$$f_\perp(R) = \left\{ ((a,f),(\overline{a},f')) \;\middle|\; \begin{array}{ll} (\forall\alpha \in dom_C(f)) & f(\alpha)\; R\; f'(\alpha) \text{ and} \\ (\forall\beta \in dom_D(f)) & f(\beta) \sim f'(\beta) \end{array} \right\}.$$

Definition 4.4 (Duality). Let $g_\perp = c^* \circ f_\perp$. A relation R is called a *duality relation* if it is a post-fixpoint of $g_\perp$. We call the greatest fixpoint of $g_\perp$ *duality* and denote it by $\perp$.

4.3 Parallelizability

Unlike a linear endpoint, a channel endpoint with an unrestricted type may be shared between multiple parallel processes.

There is no coordination of messages over a single channel: messages consecutively sent from one process may be received by different processes, and a single process may receive messages originating from different processes.

Futhermore, we wish to type-check each process independently. That is, to type-check P running in parallel with Q, we first look at P (forgetting entirely about Q), and then consider Q (again, not taking into account P at all). The type of the unrestricted channel should carry all information needed to make sure the parallel composition of P and Q will work correctly, assuming P and Q are individually well-formed. This is not the case for all types that can be represented by session coalgebras.

Example 4.5. Suppose we would allow a channel x of type $\mu X.\ \mathtt{un!int}.\ \mathtt{un!bool}.\ X$ to be used by, say, by some parallel processes P_1 and P_2 that first write some integer, and then some Boolean value to the channel:

$$\begin{aligned} P_1 &: \text{send “1” on } x\text{, then send “}\mathtt{true}\text{” on } x, \\ P_2 &: \text{send “2” on } x\text{, then send “}\mathtt{false}\text{” on } x. \end{aligned}$$

Both processes are checked independently, and both conform to the protocol, so all seems well. Nonetheless, the following is a possible trace of P_1 and P_2 running in parallel:

$$P_1 \text{ sends “1”} \quad\to\quad P_2 \text{ sends “2”} \quad\to\quad P_1 \text{ sends “}\mathtt{true}\text{”} \quad\to\quad P_2 \text{ sends “}\mathtt{false}\text{”}.$$

Although both processes locally conform to the type, when running in parallel the channel can carry two consecutive integers. A process reading from the channel might receive these consecutive integers, even though the type specifies that each integer must be followed by a Boolean.

So, only a subset of unrestricted types can safely be used (by parallel processes), we call such types *parallelizable*. We will later define the type system such that well-formed processes can only interact with unrestricted types that are parallelizable. Hence, unrestricted, non-parallelizable types can be thought of as being invalid in a certain sense.

Definition 4.6. Given a coalgebra (X, c), some subset $Y \subseteq X$ is *parallelizable*, written $\mathrm{par}(Y)$, if for every x and y in Y one of the following holds: $x \sim y$, or $\sigma(x) = \mathrm{un}$, or $\sigma(y) = \mathrm{un}$.

Intuitively, a set of states is parallelizable if every message that is sent or received is of the same form. That is, either all messages are branch labels, drawn from the same set, or all messages are communications of the same data type. We make this slightly stronger and say that all states should be pairwise bisimilar. The only exception are un states: Since they do not (directly) represent any channel interactions, they can effectively be ignored.

Often, we are interested in the parallelizability only of a specific state. Recall that $\langle x \rangle_c$ denotes the subcoalgebra generated by $x \in X$ in c.

$$
\begin{aligned}
h_{\sqsubseteq}(R) = \quad & \{((\mathrm{com}, \mathit{in}, f), (\mathrm{com}, \mathit{in}, g)) && \mid f(*)\ R\ g(*) \text{ and } f(\text{đ})\ R\ g(\text{đ})\ \} \\
\cup \quad & \{((\mathrm{com}, \mathit{out}, f), (\mathrm{com}, \mathit{out}, g)) && \mid f(*)\ R\ g(*) \text{ and } g(\text{đ})\ R\ f(\text{đ})\ \} \\
\cup \quad & \{((\mathrm{branch}, \mathit{in}, L_1, f), && \\
& \qquad\qquad (\mathrm{branch}, \mathit{in}, L_2, g)) && \mid L_1 \subseteq L_2 \text{ and } \forall l \in L_1.\ f(l)\ R\ g(l)\ \} \\
\cup \quad & \{((\mathrm{branch}, \mathit{out}, L_1, f), && \\
& \qquad\qquad (\mathrm{branch}, \mathit{out}, L_2, g)) && \mid L_2 \subseteq L_1 \text{ and } \forall l \in L_2.\ f(l)\ R\ g(l)\ \} \\
\cup \quad & \{((\mathrm{bsc}, d, f_\emptyset), (\mathrm{bsc}, d', f_\emptyset)) && \mid d \leq_D d'\ \} \\
\cup \quad & \{((\mathrm{end}, f_\emptyset), (\mathrm{end}, f_\emptyset))\ \} && \\
\cup \quad & \{((\mathrm{un}, f), (\mathrm{un}, g)) && \mid f(*)\ R\ g(*), \text{ and } \mathrm{par}(f(*)) \text{ iff } \mathrm{par}(g(*))\ \}
\end{aligned}
$$

Fig. 7. Monotone map $h_{\sqsubseteq} \colon \mathrm{Rel}_X \to \mathrm{Rel}_{F(X)}$ that defines simulations.

Definition 4.7. Let $\langle x \rangle_c^{\gg}$ be the smallest subset of $\langle x \rangle_c$ that contains x and is closed under continuation transitions:

$$\langle x \rangle_c^{\gg} = \bigcap \{ Y \subseteq X \mid x \in Y \text{ and } \delta(y)(\alpha) \in Y \text{ for all } y \in Y \text{ and } \alpha \in \mathit{dom}_C(\delta(y)) \}$$

A state x is parallelizable, written $\mathrm{par}(x)$, if $\langle x \rangle_c^{\gg}$ is parallelizable.

4.4 Simulation and Subtyping

Intuitively, a coalgebra simulates another if the behaviour of the latter "is contained in" the former. Subtyping, originally defined on session types by Gay and Hole [24], is a notion of substitutability of types [23]. We will define our notion of simulation such that it coincides with subtyping, just like bisimulation provides a notion of type equivalence [24].

We motivate subtyping in session types, and its related notions of *covariance* and *contravariance*. Consider a process P with a channel x with type $T = ?\texttt{real}$. That is, P is ready to receive on x a real value and to treat it as such. Suppose P is meant to interact with another process Q, which is ready to provide values on x. Clearly, P can operate correctly if Q were to send an `int` over x, because every `int` is a `real`. This way, to formalize the correct interaction between P and Q, channel x in P can also be safely specialized to have the type $S = ?\texttt{int}$. Indeed, because `int` is a subtype of `real`, we will have that `?int` is a subtype of `?real`. Thus, the session types (i.e., `?int` and `?real`) are related in the same order as the data types (i.e., `int` and `real`); this is called *covariance*.

Now consider a process Q with a channel y with type $T = !\texttt{int}$, and a process R with a channel with type `?real`. Again, because every `int` is a `real`, R can operate correctly with an integer received from Q. To formalize this, channel y in Q can be safely generalized to have the session type $S = !\texttt{real}$. That is, `!real` is a subtype of `!int`. Thus, in this case, the session types (i.e., `!real` and `!int`) are related in the opposite orders as the data types; this is called *contravariance*.

Subtyping can be justified similarly for labelled choices: the subtype of an external choice can have a subset of choices, while the subtype of an internal choice can add more options. For all types, it holds that states reached through transitions are covariant, i.e., if T is a subtype of U, then continuations of T must be subtypes of continuations (of the same label) of U.

Finally, consider a process P with a channel x of type $T = \mu X.\ ?\texttt{real}.\ ?\texttt{real}.\ X$. Following the reasoning above, one might suspect x could be specialized to have type $U = \mu X.\ ?\texttt{real}.\ ?\texttt{int}.\ X$, but this is not the case. T is parallelizable, so P can safely interact with it, but U is not parallelizable, so if x were of the latter type, the channel could not be used. Thus, subtyping of unrestricted types comes with the extra condition that subtypes must have the same parallelizability.

The monotone map $h_{\sqsubseteq}$ in Figure 7 captures these ideas formally.

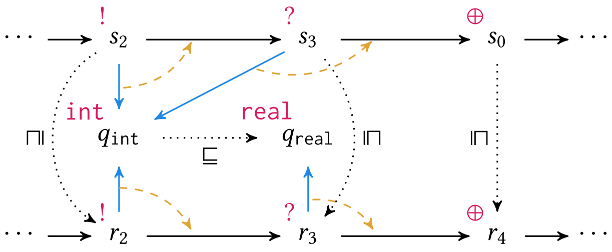


Fig. 8. Example 4.9: Simulation for two clients of the mathematical server (indicated by dotted arrows).

Definition 4.8 (Similarity). Let $g_{\sqsubseteq} = c^* \circ h_{\sqsubseteq}$. A relation R is called a *simulation* if it is a post-fixpoint of $g_{\sqsubseteq}$. We call the greatest fixpoint of $g_{\sqsubseteq}$ *similarity* and denote it by $\sqsubseteq$.

Let us illustrate similarity by means of an example.

Example 4.9. Recall the two client protocols for our mathematical server in Figures 2 and 3, which concerns states $s_0, s_1, \ldots$ and $r_0, r_1, \ldots$, respectively. We can now prove that the protocol in Figure 3 can also connect to the server, because it is a supertype of the protocol in Figure 2. To do that, we establish a simulation relation between the states of both client protocols. In Figure 8, we display a part of both session coalgebras side-by-side and indicate with dotted arrows the pairs that have to be related by a simulation relation to show that these states are similar, that is, related by $\sqsubseteq$. It should be noted that we simulate states from the second coalgebra by that of the first, that is, we show $s_k \sqsubseteq r_k$ for the shown states. There is one exception to this, namely, $q_{\texttt{int}} \sqsubseteq q_{\texttt{real}}$.

The following proposition records some properties of and tight connections between the relations that we introduced.

Proposition 4.10. *Bisimilarity $\sim$ is an equivalence relation, duality $\perp$ is symmetric, and similarity $\sqsubseteq$ is a preorder. Moreover, for all states x, y, and z of a session coalgebra, we have that*

(1) $x \sim y$ iff $x \sqsubseteq y$ and $y \sqsubseteq x$;
(2) $x \perp y$ and $x \perp z$ implies $y \sim z$; and
(3) $x \perp y$ and $y \sim z$ implies $x \perp z$.

The first claims, that bisimulation is an equivalence, duality is symmetric, and similarity is a preorder, follows from a straightforward coinductive analysis.

To prove the rest of the proposition, we start with a technical lemma.

Lemma 4.11. *If $x \sim y$, then x is parallelizable if and only if y is parallelizable.*

Proof. Suppose x is parallelizable. That means all pairs s, r of states in $\langle x \rangle_c^{\gg}$ are either bisimilar, or one of s or r is an "un" state. Because bisimulation requires all transitions to be bisimilar, any state in $\langle y \rangle_c^{\gg}$ has a bisimilar state in $\langle x \rangle_c^{\gg}$. So, any a, b in $\langle y \rangle_c^{\gg}$ where $\mathrm{op}(a) \neq \mathrm{un}$ and $\mathrm{op}(b) \neq \mathrm{un}$, has a', b' in $\langle x \rangle_c^{\gg}$ with $a \sim a' \sim b' \sim b$. Bisimulation is transitive, so $a \sim b$. Thus, y is parallelizable. The reverse case is analogous. □

Given a subtyping relation $\leq$, type equivalence is generally defined as the derived equivalence relation $x \equiv y$ iff $x \leq y$ and $y \leq x$. We defined bisimulation separately from simulation, but it coincides with this derived equivalence. This coincides with the claim that for any states x and y, $x \sim y$ if and only if $x \sqsubseteq y$ and $y \sqsubseteq x$.

Proof of Proposition 4.10. Recall that bisimulation is symmetric, so if $x \sim y$ then $\delta(x)(\alpha) \sim \delta(y)(\alpha)$ and $\delta(y)(\alpha) \sim \delta(x)(\alpha)$ for any $\alpha \in dom(x)$. Using Lemma 4.11 in the case that $\mathrm{op}(x) =$

$\mathrm{op}(y) = \mathrm{un}$, it is easy to confirm that the bisimulation relation is a simulation. So, $x \sim y$ implies $x \sqsubseteq y$ but also $y \sim x$, by symmetry, thus $y \sqsubseteq x$.

Suppose $x \sqsubseteq y$ and $y \sqsubseteq x$, which means that there are simulation relations R_1 and R_2 with $x \ R_1 \ y$ and $y \ R_2 \ x$. It is straightforward to prove that the relation $R = R_1 \cap R_2^{\mathrm{op}}$, where R_2^{op} is the dual relation, is a bisimulation relation. To this end, suppose that $x_1 \ R \ y_1$, $c(x_1) = (a, f)$ and $c(y_1) = (b, g)$. Let us consider first the case when $a = (\mathrm{bsc}, d)$. Since $x_1 \ R_1 \ y_1$, we necessarily have $b = (\mathrm{bsc}, e)$ with $d \leq e$. Symmetrically, we also have $y_1 \ R_2 \ x_1$ and thus $e \leq d$. Then clearly $x_1 \ g_{\sim}(R) \ y_1$ by definition. Second, we treat as the case where $a = (\mathrm{com}, \mathrm{in}) = b$, which have to be equal again, because R_1 (or R_2) is a simulation. Since $x_1 \ R_1 \ y_1$, we also have that $f(*) \ R_1 \ g(*)$ and $f(\mathscr{d}) \ R_1 \ g(\mathscr{d})$. Symmetrically, we obtain $g(*) \ R_2 \ g(*)$ and $g(\mathscr{d}) \ R_2 \ f(\mathscr{d})$ from $y_1 \ R_2 \ x_1$. Putting these together, we thus have $f(*) \ R \ g(*)$ and $f(\mathscr{d}) \ R \ g(\mathscr{d})$ and so $x_1 \ g_{\sim}(R) \ y_1$. All the other cases are analogous and R is indeed a bisimulation, which implies that $x \sim y$.

The second claim is that $x \perp y$ and $x \perp z$ implies $y \sim z$. By definition of duality (Definition 4.4), we have $\sigma(x) = \overline{\sigma(y)}$ and $\sigma(x) = \overline{\sigma(z)}$. Thus, $\sigma(y) = \sigma(z)$. Duality of the transitions follow from a coinductive analysis. Let $f = \delta(x)$, $g = \delta(y)$ and $h = \delta(z)$, then $f(\alpha) \perp g(x)$ and $f(\alpha) \perp h(\alpha)$ for all $\alpha \in \mathit{dom}(f)$ (which is equal to $\mathit{dom}(g)$ and $\mathit{dom}(h)$). The coinductive hypothesis implies $g(\alpha) \sim h(\alpha)$. Bisimilarity of y and z follows directly.

The third claim, that $x \perp y$ and $y \sim z$ implies $x \perp z$, is proven similarly. □

Property (1) of Proposition 4.10 unfortunately does not follow from the general principles established by Hughes and Jacobs [32], because $h_{\sqsubseteq}$ does not arise as a lifting induced by an order on the functor F. The problem is that the relation R occurs in $h_{\sqsubseteq}(R)$ both normally and dually, which forces us to define the lifting $h_{\sqsubseteq}$ and prove Proposition 4.10 manually.

4.5 Duality Functions

It is useful to have a function mapping any state x to their dual, as long as duality is defined on x. Note that although there may be multiple states that are dual to x, according to Proposition 4.10 all such states are bisimilar, so it does not matter which one we designate as *the* dual of x.

Definition 4.12. A partial function $f : X \to X$ is called a *duality function* on session coalgebra (X, c) if for all $x \in X$, we have

(1) If $\overline{\sigma(x)}$ is defined, then $f(x) \perp x$;
(2) If $\overline{\sigma(x)}$ is undefined, then $f(x)$ is also undefined.

Not each session coalgebra admits a duality function. However, we can close any coalgebra (X, c) under duality as such,

$$X^{\perp} = \{(x, \uparrow) \mid x \in X\} \ \cup \ \{(x, \downarrow) \mid x \in X, \overline{\sigma(x)} \text{ is defined}\},$$

where $\uparrow$ and $\downarrow$ are distinct, arbitrary objects used to distinguish between the states that should have the same meaning as in X, from those that should get a dual meaning. The coalgebra map $c^{\perp} : X^{\perp} \to F(X^{\perp})$ is defined as

$$\begin{aligned} c^{\perp}(x, \uparrow) &= (\sigma(x), \alpha \mapsto (\delta(x)(\alpha), \uparrow)), \\ c^{\perp}(x, \downarrow) &= \left(\overline{\sigma(x)}, \alpha \mapsto \begin{cases} (\delta(x)(\alpha), \downarrow) & \text{if } \alpha \in \mathrm{dom}_C(f) \\ (\delta(x)(\alpha), \uparrow) & \text{if } \alpha \in \mathrm{dom}_D(f) \end{cases} \right). \end{aligned}$$

The dual closure admits an almost trivial duality function $f : X^\perp \to X^\perp$:

$$\begin{aligned} f(x,\uparrow) &= (x,\downarrow) \qquad \text{if } (x,\downarrow) \in X^\perp \\ f(x,\downarrow) &= (x,\uparrow) \end{aligned}$$

Proposition 4.13. *Let $(X^\perp, c^\perp)$ be the dual closure of some session coalgebra, then $(x,\uparrow) \perp (x,\downarrow)$ for every state $x \in X$ such that $(x,\downarrow) \in X^\perp$.*

Proof. Using a coinductive reasoning, we have to show

$$\begin{aligned} \sigma^\perp(x,\uparrow) &= \overline{\sigma^\perp(x,\downarrow)} \\ \delta^\perp((x,\uparrow))(\alpha) \perp \delta^\perp((x,\downarrow))(\alpha) & \qquad \forall \alpha \in \mathrm{dom}_C(\delta((x,\uparrow))) \\ \delta^\perp((x,\uparrow))(\beta) \sim \delta^\perp((x,\downarrow))(\beta) & \qquad \forall \beta \in \mathrm{dom}_D(\delta((x,\uparrow))) \end{aligned}$$

The first claim follows by construction. For the second, note that $\delta^\perp((x,\uparrow))(\alpha) = (\delta(x)(\alpha),\uparrow)$ and $\delta^\perp((x,\downarrow))(\alpha) = (\delta(x)(\alpha),\downarrow)$, so the claim follows from the coinductive hypothesis. For the final claim, we see that $\delta^\perp((x,\uparrow))(\beta) = (\delta(x)(\beta),\uparrow)$ and $\delta^\perp((x,\downarrow))(\beta) = (\delta(x)(\beta),\uparrow)$, so the two sides are in fact equal. Bisimilarity is reflexive, thus concluding the coinductive proof. □

Since duality is symmetric, the above proposition suffices to conclude that f is indeed a duality function. Furthermore, it should be apparent that f is in fact a computable function.

In the following sections, we will assume that session coalgebras under consideration have a computable duality function, since we can always use the above procedure to construct one. In a slight abuse of notation, we reuse the $\overline{\cdot}$ operation to mean this assumed duality function on states, i.e., $\overline{x}$ denotes the dual of x.

4.6 Decidability

In a practical type checker, we need an algorithm to decide the relations defined above. In this subsection, we show an algorithm that computes the answer in finite time for a certain class of types.

Definition 4.14. A coalgebra c is *finitely generated* if $\langle x \rangle_c$ is finite for all x.

This restriction is not problematic for types, as the following lemma shows.

Lemma 4.15. *The coalgebra of types* $(\mathsf{Type}, c_{\mathsf{Type}})$ *is finitely generated.*

Proof. The full proof is quite technical, so we give a sketch. Consider some $T \in \mathsf{Type}$, and the coalgebra generated at that T. Note that instead of substituting recursion variables X at every unfold, we can generate an equivalent coalgebra (modulo the name of each state) by defining $\mathit{unfold}(\mu X.U) = U$. Then, every non-recursive transition goes to a strictly smaller expression. Simply removing the binder, without substituting anything for X in U does mean that now X is free in U. So, whenever we unfold a type (i.e., remove a binder), we keep a record of the bound value (which, in this case, is $\mu X.U$). Let ζ denote this record, mapping variables to their bound values (so, following the preceding example, we have $\zeta(X) = \mu X.U$), then we slightly augment the transition function of the coalgebra of types:

$$\delta'(U)(\alpha) = \begin{cases} \zeta(X) & \text{if } \delta(U)(\alpha) = X \text{ for some variable } X, \\ \delta(U)(\alpha) & \text{otherwise.} \end{cases}$$

That is, whenever the original coalgebra of types definition calls for a transition to a variable X, from some state U, we look up the value of X in the record we kept, and point the transition to the corresponding state. This state is guaranteed to have been seen previously, i.e., lie on the path from T to U. Finally, there is only a finite number of transitions per state. Hence, the total number of states reachable from T is finite. □

To determine whether two states x and y are bisimilar, we need to find a bisimulation R with $x\ R\ y$. We start with the simplest relation $R = \{(x, y)\}$, and ask if this is a bisimulation.

First, we check that for all $(u, w) \in R$, $\sigma(u) = \sigma(w)$, or in the case of bsc states that $\mathrm{da}(u) \leq_D \mathrm{da}(w)$ and $\mathrm{da}(w) \leq_D \mathrm{da}(u)$. If $\sigma(u) \neq \sigma(w)$ for any pair in R, then we know that no superset of R is a bisimulation, and the algorithm rejects.

Second, we check the matching transitions. For every $(u, w) \in R$ and $\alpha \in dom(\delta(u))$, we check whether $(\delta(u)(\alpha), \delta(w)(\alpha)) \in R$. If we encounter a missing pair, then we add it to R and ask whether this new relation is a bisimulation, i.e., return to the first step. If all destinations for matching transitions are present in R, then R is by construction a bisimulation for (x, y). Hence, $x \sim y$.

This algorithm tries to construct the smallest possible bisimulation containing (x, y), by only adding strictly necessary pairs. If the algorithm rejects, then there is no such bisimulation; hence, $x \nsim y$. Additionally, the algorithm only examines pairs in $\langle x \rangle_c \times \langle y \rangle_c$. If there are finitely many of such pairs, then the algorithm will terminate in finite time

The above described algorithm can be suitably adapted to similarity and duality, which gives us the following result.

Theorem 4.16. *Bisimilarity, similarity, and duality of any states x and y are decidable if $\langle x \rangle_c$ and $\langle y \rangle_c$ are finite. Parallelizability of any state x is decidable if $\langle x \rangle_c^{\gg}$ (as in Definition 4.7) is finite.*

Proof. We start with the decidability of bisimilarity.

A relation R is a post-fixpoint of $g_\sim$ if $(a, b) \in g_\sim(R)$ for all $(a, b) \in R$. This involves computing a pre-image, which is, in general, not easy. Because we are not interested in pair $g_\sim(R)$ that is not in R, and because $g_\sim(R) = c^*(f_\sim(R))$, R is a post-fixpoint if $(c(a), c(b)) \in f_\sim(R)$ for all $(a, b) \in R$. When R is finite, the latter is decidable—either trivially or by the assumption that $\leq_D$ is decidable.

By assumption, there are only finitely many states to be transitioned to, so the algorithm described above can only add finitely many pairs before reaching a relation that either already is a bisimulation or can never be made into a bisimulation.

The decidability of parallelizability for a finite set is straightforward.

If Y is finite, then the set $Y \times Y$ of all pairs is finite. We can enumerate all such pairs, and decide whether it either is in the bisimilarity relation or contains a un state, in finite time. Once we encounter a pair for which this does not hold, we know x is not parallelizable. If we have checked all pairs and not encountered such a counter-example, then we know x is parallelizable.

Each continuation is a transition, so for any state x such that $\langle x \rangle_c$ is finite, $\langle x \rangle_c^{\gg}$, i.e. the smallest set that is closed under continuations and contains x, is also finite. Thus, parallelizability of x is decidable.

Finally, the algorithm, and decidability proof, for duality and similarity are analogous to that for bisimilarity. □

Corollary 4.17. *Bisimilarity, similarity, duality and parallelizability are decidable for c_{Type}.*

Proof. The coalgebra $(\mathsf{Type}, c_{\mathsf{Type}})$ is finitely generated, so Theorem 4.16 applies. □

4.7 Session Coalgebras as Type Equations and the Rational Fixed Point

We have seen that the coalgebra c_{Type} is finitely generated, which means that every type T gives rise to a finite subcoalgebra $\langle T \rangle_{c_{\mathsf{Type}}}$ that contains T. All the examples of session coalgebras, apart from c_{Type}, were however finite. In the theory of automata, there is a familiar correspondence between finite deterministic automata and regular expressions. Thus, one may ask whether there

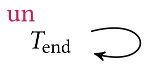


Fig. 9. An alternative completed protocol.

is such a correspondence between finite state session coalgebras and session types. The answer to this question is not a definite "yes" but requires a bit of care.

Indeed, it is simple to provide an example of a session coalgebra that has no corresponding session type, see Figure 16. But there is more to be said. Coming back to language theory, the above mentioned correspondence actually relates finite automata and regular languages with each other, and uses this to establish the correspondence with regular expressions. This means that we have to find an analogue to regular languages for session coalgebras.

In the theory of coalgebra, the correspondence of regular languages and regular expressions is approached by relating the so-called rational fixed point of the functor for deterministic automata to a coalgebra on the set of regular expressions [53]. To parallel this approach, we will thus have to find a correspondence between the rational fixed point for our session coalgebra functor F and the coalgebra c_{Type}. Let us begin by introducing the rational fixed point. We will use a simplified definition [41], which works in our setting and makes the theory more elementary.

Definition 4.18. Let $\mathbf{CoAlg}_{\mathrm{fg}}(F)$ be the category of finitely generated coalgebras, that is, the full subcategory of $\mathbf{CoAlg}(F)$ in which all coalgebras are finitely generated. The *rational fixed point* of F is the final object $(\rho F, r)$ in $\mathbf{CoAlg}_{\mathrm{fg}}(F)$.

It should be noted that the rational fixed point does indeed exist in our case, because the category **Set** is locally finitely presentable [3] and the functor F preserves filtered colimits [2]. Moreover, every finite session coalgebra is of course finitely generated and thus an object in $\mathbf{CoAlg}_{\mathrm{fg}}(F)$.

We can ask ourselves now whether the coalgebra on session types corresponds to the rational fixed point, which can be seen as semantic domain for finitely generated session coalgebras and thus taking the role similar to that of regular languages.

Unfortunately, this is not the case. Figure 9 depicts a session coalgebra that does not have a syntactical equivalent; it consists of a state T_{end} with $\sigma(T_{\text{end}}) = \text{un}$ and $\delta(T_{\text{end}})(*) = T_{\text{end}}$.

Theorem 4.19. *There is a unique homomorphism $t\colon (\mathsf{Type}, c_{\mathsf{Type}}) \to (\rho F, r)$ from the coalgebra of types to the rational fixed point of F. This homomorphism is not an isomorphism.*

Proof. In Lemma 4.15, we proved that c_{Type} is finitely generated and therefore finitely presentable [41, 42]. This gives the unique homomorphism t into the rational fixed point.

Suppose, toward a contradiction, that the homomorphism t would have an inverse t^{-1}. Then for any finitely presentable coalgebra (X, c), we get a homomorphism $h\colon X \to \rho F$ and thus a homomorphism $t^{-1} \circ h\colon X \to \rho F \to \mathsf{Type}$. In Figure 9, we give an example of a finite session coalgebra that has no corresponding type, i.e., there is no homomorphism $(X, c) \to (\mathsf{Type}, c_{\mathsf{Type}})$. Therefore, t cannot have an inverse. □

This result is disappointing, as session types are not in correspondence with finitely generated session coalgebras. However, the proof also instructs us how we can remedy this problem. The issue with the session coalgebra in Figure 9 is that the un-state T_{end} has a direct self-loop without going through a pretype, which would be required to be able to associate a session type with this state. It turns out that this is the only real obstacle and that we can obtain an inverse of sorts of the homomorphism t from Theorem 4.19, if we restrict to appropriate session coalgebras.

Theorem 4.20. *Let $c\colon X \longrightarrow F(X)$ be a finite session coalgebra. Suppose that for all $x \in X$, if whenever $c(x) = (\mathrm{un}, f)$, then $c(f(*)) = (\mathrm{com}, p, f')$ or $c(f(*)) = (\mathrm{branch}, p, L, f')$. Then there is a homomorphism $h\colon (X, c) \to (\mathsf{Type}, c_{\mathsf{Type}})$.*

Proof. We follow the usual idea of creating a finite system of type equations induced by the states of c and show that this system is solvable in Type [2]. The homomorphism then maps a state to the corresponding solution.

Toward this, we will use type variables X_y for every state $y \in X$, define metavariables T_y that represent y as type equation, and a family of pretypes p_u indexed by elements $u \in F(X)$. This family p_u of pretypes is defined as follows:

$$p_{(\mathrm{com},\mathrm{in},f)} = ?X_{f(*)}.\ X_{f(d)} \qquad p_{(\mathrm{com},\mathrm{out},f)} = !X_{f(*)}.\ X_{f(d)}$$

$$p_{(\mathrm{branch},\mathrm{in},L,f)} = \&\{l : X_{f(l)}\}_{l\in L} \qquad p_{(\mathrm{branch},\mathrm{out},L,f)} = \oplus\{l : X_{f(l)}\}_{l\in L}$$

For every state $y \in X$, we require the following equation to hold:

$$T_y = \begin{cases} \mathsf{lin}\ p_{(\mathrm{com},p,f)}, & c(y) = (\mathrm{com}, p, f),\\ \mathsf{lin}\ p_{(\mathrm{branch},p,L,f)}, & c(y) = (\mathrm{branch}, p, L, f),\\ \mathrm{end}, & c(y) = (\mathrm{end}, f),\\ d, & c(y) = (\mathrm{bsc}, d),\\ \mathsf{un}\ p_{c(f(*))}, & c(y) = (\mathrm{un}, f). \end{cases}$$

Note that the last case is well-defined because of the condition on the coalgebra c. Moreover, it is important to note that the variables only occur inside the pretypes p_u, which makes their occurrences contractive (or guarded). Therefore, we can solve the system of equations with types $\overline{T_y}$ by putting $\overline{T_y} = \mu X_y.\ T_y$ and then repeatedly substituting $\overline{T_{y'}}$ for all $X_{y'}$ with $y' \neq y$ in $\overline{T_y}$. This repeated substitution terminates, because each $\overline{T_{y'}}$ binds one or more variables. That we get closed and contractive types is thus ensured.

We can then define the map $h\colon X \longrightarrow \mathsf{Type}$ by $h(y) = \overline{T_y}$. That this is a homomorphism is easily proven by using the system of equations:

$$\begin{aligned} c_{\mathsf{Type}}(h(y)) = c\left(\overline{T_y}\right) &= \begin{cases} c_{\mathsf{Type}}(\mathsf{lin}\ p_{(\mathrm{com},\mathrm{out},f)}), & c(y) = (\mathrm{com}, \mathrm{out}, f)\\ \vdots & \\ c_{\mathsf{Type}}(\mathsf{un}\ p_{c(f(*))}), & c(y) = (\mathrm{un}, f) \end{cases}\\ &= \begin{cases} (\mathrm{com}, \mathrm{out}, \left[* \mapsto \overline{T_{f(*)}}, d \mapsto \overline{T_{f(*)}}\right]), & c(y) = (\mathrm{com}, \mathrm{out}, f)\\ \vdots & \\ (\mathrm{un}, [* \mapsto \mathsf{lin}\ p_{c(f(*))}]), & c(y) = (\mathrm{un}, f) \end{cases}\\ &= \begin{cases} (\mathrm{com}, \mathrm{out}, h \circ f), & c(y) = (\mathrm{com}, \mathrm{out}, f)\\ \vdots & \\ (\mathrm{un}, h \circ f), & c(y) = (\mathrm{un}, f) \end{cases}\\ &= F(h)(c(y)). \end{aligned}$$

As this holds for all $y \in X$, $c_{\mathsf{Type}} \circ h = F(h) \circ c$ and h is a homomorphism. □

Even though the homomorphism that we found in Theorem 4.20 seems canonical, it is not unique. This is because types are considered equal only syntactically, up to α-equivalence, but uniqueness would require that *bisimilar* types are equal. For instance, any closed type T is bisimilar to $\mu X.\ T$ for any variable X but these are not syntactically equal. Clearly, h could be turned into a homomorphism $X \longrightarrow \mathsf{Type}/{\sim}$ to the quotient of Type by bisimilarity. This is a standard technique [41, 53] to express the bisimulation proof principle that we have devised in Section 4.1 as a unique mapping property. Thus, $\mathsf{Type}/{\sim}$ with the coalgebra induced by the quotient [49] becomes final among appropriately restricted session coalgebras.

Corollary 4.21. *Let* $\mathbf{CoAlg}^{p}_{\mathrm{fg}}(F)$ *be the full subcategory of* $\mathbf{CoAlg}_{\mathrm{fg}}(F)$ *that consists of coalgebras that fulfil the condition of Theorem 4.20. The bisimilarity quotient* $(\mathsf{Type}/{\sim}, c_{\mathsf{Type}/\sim})$ *of* $(\mathsf{Type}, c_{\mathsf{Type}})$ *is final in* $\mathbf{CoAlg}^{p}_{\mathrm{fg}}(F)$.

Proof. We note that the quotient of a finitely generated coalgebra is also finitely generated [41]. Moreover, $(\mathsf{Type}/{\sim}, c_{\mathsf{Type}/\sim})$ fulfills the condition and is therefore an object in $\mathbf{CoAlg}^{p}_{\mathrm{fg}}(F)$. By Theorem 4.20 and the above discussion, it is final in this category. □

Let us illustrate the constructed homomorphism into c_{Type} in practice.

Example 4.22. We apply the method in the proof of Theorem 4.20 to the session coalgebra in Figure 14 on page 26. First, we have to represent the coalgebra as a system of type equations:

$$T_U = \mathsf{un}\,?X_q.\ X_U \qquad T_T = \mathsf{lin}\,?X_q.\ X_U \qquad T_q = \mathtt{int}$$
$$T_V = \mathsf{un}\,!X_q.\ X_V \qquad T_S = \mathsf{lin}\,!X_q.\ X_V$$

Next, we need to solve this system for every type by prefixing the equation with a fixed point operator and then substituting for the unbound variables. For instance, we get for the state U the following solution:

$$\begin{aligned}\overline{T_U} &= \mu X_U.\ T_U \\ &= \mu X_U.\ \mathsf{un}\,?X_q.\ X_U \\ &= \mu X_U.\ \mathsf{un}\,?\overline{T_q}.\ X_U \\ &= \mu X_U.\ \mathsf{un}\,?(\mu X_q.\ \mathtt{int}).\ X_U.\end{aligned}$$

This type is very generous in using fixed point operators. In the coalgebra $\mathsf{Type}/{\sim}$, this type would be considered equal to

$$\mu X_U.\,\mathsf{un}\,?\mathtt{int}.\ X_U,$$

which is the type that we may intuitively derive from Figure 14. The homomorphism $h\colon X \longrightarrow \mathsf{Type}/{\sim}$ assigns to every state x of the coalgebra (the equivalence class of) the solution $\overline{T_x}$. In particular, we have $h(T) = \mathsf{lin}\,?\mathtt{int}.\ \mu X_U.\ \mathsf{un}\,?\mathtt{int}.\ X_U$.

Theorems 4.20 and Corollary 4.21 show that there is a correspondence between session types (quotient by bisimilarity), finite session coalgebras (with an appropriate condition), and the rational fixed point. This is similar to the trinity of regular expressions (quotient thereof), deterministic automata, and regular languages. However, Theorem 4.20 also shows that this correspondence does not work out fully, because the rational fixed point is not in the category $\mathbf{CoAlg}^{p}_{\mathrm{fg}}(F)$. We suspect that the correspondence can be restored by altering the functor F. More precisely, one would have to provide a comonad, such that $\mathbf{CoAlg}^{p}_{\mathrm{fg}}(F)$ is isomorphic to the category of coalgebras over that comonad and then show that a rational fixed point exists in that category. Thus, the comonad would encode the condition that we had to impose on F-coalgebras in $\mathbf{CoAlg}^{p}_{\mathrm{fg}}(F)$ and the rational fixed point would be final among them. It seems that such a comonad can be easily obtained as sub-comonad of the cofree comonad of F, but we shall not pursue this route further in the present article. For the time being, we content ourselves with the correspondence between certain session coalgebras and session types as in Corollary 4.21.

$$
\begin{array}{rcll}
P, Q & ::= & \overline{x}\langle y\rangle.P & \text{output } y \text{ on channel } x\text{, continue as } P \\
 & \mid & x(y).P & \text{bind input from channel } x \text{ to variable } y\text{, continue as } P \\
 & \mid & x \triangleright \{l_i : P_i\}_{i \in I} & \text{offer choices } l_1, l_2, \ldots\text{, continue as } P_j \text{ (for some } j \in I) \\
 & \mid & x \triangleleft l.P & \text{select label } l\text{, continue as } P \\
 & \mid & P \mid Q & \text{parallel composition of } P \text{ and } Q \\
 & \mid & !P & \text{replication} \\
 & \mid & \mathbf{0} & \text{finished process} \\
 & \mid & (\nu xy)P & \text{channel creation}
\end{array}
$$

Fig. 10. Syntax of processes.

5 PROCESSES AND A TYPE SYSTEM BASED ON SESSION COALGEBRAS

Session types are meant to discipline the behaviour of the channels of an interacting process, to ensure that prescribed protocols are executed as intended. Up to here, we have focused on session types (i.e., their representation as session coalgebras and coinductively-defined relations on them) without committing to a specific syntax for processes. This choice is on purpose: Our goal is to provide a truly syntax-independent justification for session types.

In this section, we introduce a syntax of processes and rely on session coalgebras to define a session type system, similar to the one defined by Vasconcelos [61]. Because our goal is to precisely illustrate how session coalgebras (and their underpinning notions) arise in a concrete formulation of the typing rules, our type system includes only a basic support for processes with infinite behaviours and recursive session types—indeed, following Reference [24], we consider processes with replication (noted $!P$) rather than with recursion. There is no fundamental difficulty in extending our type system based on session coalgebras to match the expressive forms of recursive session types defined in Reference [61].

A note in terminology. In the following, we often use "type" to refer to the corresponding "state" of the underlying session coalgebra. In our developments, both "state" and "type" can be used interchangeably.

5.1 A Session π-calculus

The π-calculus is a formal model of interactive computation in which processes exchange messages along channels (or names) [43, 51]. As such, it is an abstract framework in which key features such as name mobility, (message-passing) concurrency, non-determinism, synchronous communication, and infinite behaviour have rigorous syntactic representations and precise operational meaning.

We consider a *session* π-calculus, i.e., a variant of the π-calculus whose operators are tailored to the protocols expressed by session types. We assume base sets of *variables*, ranged over by $x, y, z, v, \ldots$, and *labels* $\mathbb{L}$, ranged over by $l, l', \ldots$. The syntax of processes $P, Q, \ldots$ is based on References [24, 61] and given by the grammar in Figure 10. We discuss the salient aspects of the syntax.

- A process $\overline{x}\langle y\rangle.P$ denotes the output of channel y along channel x, which precedes the execution of P. Dually, a process $x(z).P$ denotes the input of a value v along channel x, which precedes the execution of process $P[y/z]$, i.e., the process P in which all free occurrences of z have been substituted by y.
- Processes $x \triangleright \{l_i : P_i\}_{i \in I}$ and $x \triangleleft l.P$ implement a labelled choice mechanism. Given a finite index set I, the *branching* process $x \triangleright \{l_i : P_i\}_{i \in I}$ denotes an external choice: the reception of a label l_j (with $j \in I$) along channel x precedes the execution of the continuation P_j. The *selection* process $x \triangleleft l.P$ denotes an internal choice; it is meant to interact with a complementary branching.

Reduction

$$(\nu xy)(\overline{x}\langle v\rangle.P \mid y(z).Q \mid R) \longrightarrow (\nu xy)(P \mid Q[v/z] \mid R) \qquad \text{[R-COM]}$$

$$(\nu xy)(x \triangleleft l_j.P \mid y \triangleright \{l_i : Q_i\}_{i\in I} \mid R) \longrightarrow (\nu xy)(P \mid Q_j \mid R) \quad (j \in I) \qquad \text{[R-SYNC]}$$

$$\frac{P \longrightarrow Q}{(\nu xy)P \longrightarrow (\nu xy)Q} \qquad \frac{P \longrightarrow Q}{P \mid R \longrightarrow Q \mid R} \qquad \text{[R-RES][R-PAR]}$$

$$\frac{P \equiv P' \qquad P' \longrightarrow Q' \qquad Q' \equiv Q}{P \longrightarrow Q} \qquad \text{[R-CONG]}$$

Structural congruence

Parallel composition:

$$P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad P \mid \mathbf{0} \equiv P \qquad !P \equiv P \mid !P$$

Scope restriction:

$$(\nu xy)(\nu vw)P \equiv (\nu vw)(\nu xy)P \qquad (\nu xy)\mathbf{0} \equiv \mathbf{0} \qquad (\nu xy)P \equiv (\nu yx)P$$

$$(\nu xy)(P \mid Q) \equiv ((\nu xy)P) \mid Q \qquad \text{if } x \text{ and } y \text{ not free in } Q$$

Fig. 11. Reduction semantics.

- Given processes P and Q, process $P \mid Q$ denotes their parallel composition, which enables their simultaneous execution.
- The process $!P$, the replication of P, denotes the composition of infinite copies of P running in parallel, i.e., $P \mid P \mid \cdots$.
- Process $\mathbf{0}$ denotes inaction.
- Process $(\nu xy)P$ denotes a restriction operator that declares x and y as *co-variables*, i.e., as complementary endpoints of the same channel, with scope P. This construct is arguably the main difference with respect to usual presentations of the π-calculus.

Our process syntax can be seen as a variant of that in Reference [24] with (co)variables and restriction as in Reference [61]. Binding occurrences of names are z in $x(z).P$ and both x and y in $(\nu xy)P$.

The operational semantics for processes is defined as a *reduction relation*, denoted $\longrightarrow$, by relying on a notion of *structural congruence* on processes, denoted $\equiv$. Figure 11 defines these two notions. Intuitively, two processes are structurally congruent if they are identical in behaviour but not necessarily in structure. It is the smallest congruence relation satisfying the axioms in Figure 11 (bottom), which formalise expected equalities for parallel composition, replication, and restriction.

We say a process P reduces to Q, written $P \longrightarrow Q$, when there is a single execution step yielding Q from P. We comment on the rules in Figure 11 (top). Following Reference [61], R-COM formalises the exchange of a value over a channel formed by two covariables. Similarly, R-SYNC formalises the synchronisation between a branching and a selection that realises the labelled choice. Rules R-RES and R-PAR are contextual rules, which allow reduction to proceed under restriction and parallel composition. Finally, Rule R-CONG says that reduction is closed under structurally congruence; we can use $\equiv$ to promote interactions that match the structure of the rules above.

5.2 Typing Rules

Let $T, U, V, \ldots$ denote states of some fixed, but arbitrary, session coalgebra (X, c) (cf. Definition 3.2). Variables are associated with these states in a *context* Γ, as described by $\Gamma ::= \emptyset \mid \Gamma, x : T$. Notice that the use of $\Gamma, x : T$ carries the assumption that x is not in Γ. A context is an unordered, finite set of pairs, that may have at most one pair (x, T) for each variable x. A context is thus isomorphic to a (partial) function from a finite set of variables to the coalgebra states that represent their types. We

$$\emptyset = \emptyset \circ \emptyset \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad \mathrm{un}(T)}{\Gamma, x : T = (\Gamma_1, x : T) \circ (\Gamma_2, x : T)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : T = (\Gamma_1, x : T) \circ \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : T = \Gamma_1 \circ (\Gamma_2, x : T)}$$

Fig. 12. Context split.

$$\frac{\mathrm{un}(\Gamma)}{\Gamma \vdash \mathbf{0}} \qquad \frac{\Gamma, x : T, y : U \vdash P \qquad T \perp U}{\Gamma \vdash (\nu xy)P} \qquad [\text{T-Inact}][\text{T-Res}]$$

$$\frac{\Gamma_1 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash P \mid Q} \qquad \frac{\Gamma \vdash P \qquad \mathrm{un}(\Gamma)}{\Gamma \vdash !P} \qquad [\text{T-Par}][\text{T-Rep}]$$

$$\frac{c(T) = (?, f) \qquad \Gamma,\ y : U,\ x : f(*) \vdash P \qquad f(\not{d}) \sqsubseteq U}{\Gamma, x : T \vdash x(y).P} \qquad [\text{T-In}]$$

$$\frac{c(T) = (!, f) \qquad \Gamma,\ x : f(*)\ \vdash P \qquad U \sqsubseteq f(\not{d})}{\Gamma, x : T, y : U \vdash \overline{x}\langle y \rangle.P} \qquad [\text{T-Out}]$$

$$\frac{c(T) = (\&, L_1, f) \qquad L_1 \subseteq L_2 \qquad \Gamma, x : f(l) \vdash P_l \qquad \forall l \in L_1}{\Gamma, x : T \vdash x \rhd \{l : P_l\}_{l \in L_2}} \qquad [\text{T-Branch}]$$

$$\frac{c(T) = (\oplus, L, f) \qquad \Gamma, x : f(l) \vdash P_l \qquad l \in L}{\Gamma, x : T \vdash x \lhd l.P_l} \qquad [\text{T-Sel}]$$

$$\frac{c(T) = (\mathrm{un}, f) \qquad \mathrm{par}(T) \qquad \Gamma, x : f(*) \vdash P}{\Gamma, x : T \vdash P} \qquad [\text{T-Unpack}]$$

Fig. 13. Declarative typing rules.

use Γ to denote this isomorphic function as well: $\Gamma(x) = T$ if $(x, T) \in \Gamma$. The domain of a context is defined accordingly.

We know un states are unrestricted, but they are not the only ones.

Definition 5.1. A state is *unrestricted*, written $\mathrm{un}(T)$, if its operation is un, end, or bsc. A context is unrestricted, written $\mathrm{un}(\Gamma)$, if all states in Γ are unrestricted, i.e., if $(x, T) \in \Gamma$ implies $\mathrm{un}(T)$. A state is *linear*, written $\mathrm{lin}(T)$, if it is not unrestricted. A context is linear, if all its states are linear.

A context Γ may be *split* into two parts Γ_1 and Γ_2, such that the linear states are strictly divided between Γ_1 and Γ_2, but unrestricted states can be copied. Following Reference [61], context split is a ternary relation, defined by the axioms in Figure 12. We may write $\Gamma_1 \circ \Gamma_2$ to refer to a context Γ for which $\Gamma = \Gamma_1 \circ \Gamma_2$ is in the context split relation. Such a context is not necessarily defined for any given contexts; we implicitly assume its existence when writing $\Gamma_1 \circ \Gamma_2$. Otherwise, $\Gamma, x : T$ would have two pairs with x, which is not allowed.

The type system considers judgments of the form $\Gamma \vdash P$, and is defined by the rules in Figure 13. A process P is *well-formed*, under a context Γ, if there is some inference tree whose root is $\Gamma \vdash P$ and whose nodes are all valid instantiations of these typing rules. As T-Inact is the only rule that does not depend on the correctness of another process, it forms the leaves of such trees.

We discuss the typing rules, which can be conveniently read keeping in mind the notations introduced in Definition 3.2 and Proposition 3.4. Rule T-Inact ensures that all linear channels in the context are interacted with until the type becomes unrestricted. If our context contains a variable x of type `?int`, then the process is required to read an `int` from it. Thus, $x :$ `?int.` $\nvdash \mathbf{0}$. In contrast, process $x(z).\mathbf{0}$ is well-formed for the same context, using Rules T-Inact and T-In:

$$\dfrac{\overline{x : \mathsf{end}, z : \mathtt{int} \vdash \mathbf{0}}}{x : ?\mathtt{int} \vdash x(z).\mathbf{0}}$$

Rule T-Res creates a channel by binding together two covariables x and y, of dual type. Rule T-Par causes unrestricted channels to be copied and linear channels to get split between composite processes, ensuring the latter occur in only a single process (cf. Figure 12). Rule T-Rep handles the replicated process $!P$, which denotes an infinite composition of copies of process P; hence, a replicated process can only use an unrestricted context.

Together, Rules T-Par and T-Res allow us to introduce new co-variables, with new types, and distribute them. But, only unrestricted types may be copied. Notice that a process does not specify which types to give the newly bound variables:

$$\begin{array}{rcl} v : \mathtt{int} & \vdash & (\nu xy)\, x(z).\mathbf{0} \mid \overline{y}\langle v\rangle.\mathbf{0} \\ x : \mu X.\ \mathsf{un}\, ?\mathtt{int}.\ X & \vdash & x(z).\mathbf{0} \mid x(z).\mathbf{0} \\ x : ?\mathtt{int} & \nvdash & x(z).\mathbf{0} \mid x(z).\mathbf{0} \end{array}$$

Each action on a channel has its own rule: Rule T-In handles input, binding the channel x to the continuation type and y to some supertype of the received type. Rule T-Out handles output, which requires the sent variable to have a subtype of whatever type the channel expects to send. Rule T-Branch handles external choice, where the process needs to offer at least all choices the type describes, coupled with processes that are correctly typed under the respective continuation types. Rule T-Sel only checks whether the single label that was chosen by the process is a valid option, and if the rest of the process is correct under the continuation type. These rules are only specified for linear states; Rule T-Unpack allows a “un” state to be used as if it was the underlying type, as long as it is parallelizable (cf. Definition 4.6).

5.3 Properties of Well-Formed Processes

For well-formed processes, the type system guarantees that:

- If the process terminates, then all linear sessions were completed.
- If a process reads a value from a channel, then the value has the type specified by the channel’s session type. If a process receives a label, then it is one of the labels specified by the channel’s session type.

We establish a standard *type preservation* result, which rests upon two auxiliary properties: substitution and subject congruence. Furthermore, substitution preserves the height of inference trees.

Lemma 5.2 (Substitution). *If $U \sqsubseteq T$ and $(y, V) \notin \Gamma$ and $\Gamma, x : T \vdash P$ as shown by some inference tree of height n, then $\Gamma, y : U \vdash P[y/x]$ can be shown by an inference tree of height n.*

Proof. By induction on the derivation of $\Gamma, x : T \vdash P$, with a case analysis on the last typing rule used, exploiting the fact that $\sqsubseteq$ is a pre-order (Proposition 4.10) and therefore transitive. The case analysis constructs an inference tree of the conclusion by simply copying the rules from the premisse inference tree, only changing the concrete types and processes, so the transformation is indeed height-preserving. □

Lemma 5.3 (Subject Congruence). *If $\Gamma \vdash P$ and $P \equiv Q$, then $\Gamma \vdash Q$.*

Proof. By induction on the derivation of $P \equiv Q$, with a case analysis on the last axiom used. □

Theorem 5.4 (Type Preservation). *If $\Gamma \vdash P$ and $P \longrightarrow Q$, then $\Gamma \vdash Q$.*

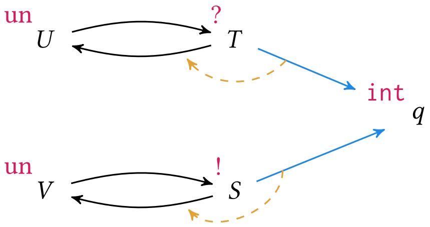


Fig. 14. Session coalgebra for P_0.

Proof. By induction on the derivation of $P \longrightarrow Q$, with a case analysis on the last applied rule, using Lemmas 5.2 and 5.3, and inversion principles derived from the assumption $\Gamma \vdash P$—all rules, except Rule T-Unpack, are syntax-directed, but ambiguities can be avoided by the conditions on types. □

In Section 4, we defined simulation through the intuition of subtyping as substitutability in one direction. The substitution lemma tells us this is indeed allowed for simulated types.

Theorem 5.5. *The following, more common, rule is admissible from the rules in Figure 13:*

$$\frac{\Gamma, x : T \vdash P \qquad U \sqsubseteq T}{\Gamma, x : U \vdash P}$$

Proof. Direct consequence of Lemma 5.2. □

That is, we could add the rule as an axiom, without changing the set of typable processes. As a corollary, bisimulation of states implies the states are equivalent with respect to the type system, in a height-preserving fashion.

Corollary 5.6. *For all bisimilar types $T \sim U$, contexts Γ and processes P, it holds that $\Gamma, x : T \vdash P$ can be shown by some inference tree of height n if and only if $\Gamma, x : U \vdash P$ can be shown by an inference tree of height n.*

Proof. If $T \sim U$, then by Lemma 4.11, we have $T \sqsubseteq U$ and $U \sqsubseteq T$, the rest follows by Lemma 5.2. □

5.4 Examples

To illustrate unrestricted behaviours and the T-Unpack rule, we first consider the process

$$P_0 = !(x(y).\overline{z}\langle y\rangle.\mathbf{0}).$$

P_0 is an unrestricted forwarder: It represents arbitrary copies of a process that first inputs a value y on x, which is then sent along z. Clearly, these are two unrestricted behaviours that determine separate protocols. Suppose the exchanged value is of type `int`, then the protocols can be specified using the session coalgebra in Figure 14.

We have the following typing derivation for $x(y).\overline{z}\langle y\rangle.\mathbf{0}$:

$$\text{T-Unpack}\ \dfrac{\text{T-In}\ \dfrac{\text{T-Unpack}\ \dfrac{\text{T-Out}\ \dfrac{\text{T-Inact}\ \dfrac{\mathrm{un}(x : U, z : V)}{x : U, z : V \vdash \mathbf{0}}}{x : U, z : S, y : \mathtt{int} \vdash \overline{z}\langle y\rangle.\mathbf{0}} \qquad \mathrm{par}(V)}{x : U, z : V, y : \mathtt{int} \vdash \overline{z}\langle y\rangle.\mathbf{0}} \qquad \mathtt{int} \sqsubseteq \mathtt{int}}{x : T, z : V \vdash x(y).\overline{z}\langle y\rangle.\mathbf{0}} \qquad \mathrm{par}(U)}{x : U, z : V \vdash x(y).\overline{z}\langle y\rangle.\mathbf{0}}$$

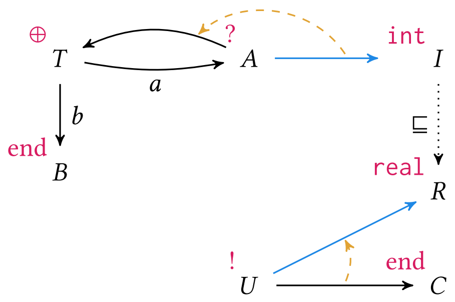


Fig. 15. Session coalgebra for process P_1.

Note that the shift from $x : U$ to $x : T$ (and from $z : V$ to $z : S$) is justified by applications of Rule T-Unpack. This is a silent rule on processes—only the type context changes. Their associated premise holds easily by Definitions 4.6 and 4.7. The derivation for P_0 finishes with an application of Rule T-Rep:

$$\textsc{T-Rep}\ \dfrac{\dfrac{\vdots}{x : U, z : V \ \vdash\ x(y).\overline{z}\langle y\rangle.\mathbf{0}} \qquad \mathrm{un}(x : U, z : V)}{x : U, z : V \ \vdash\ !x(y).\overline{z}\langle y\rangle.\mathbf{0}}$$

Consider now the process

$$P_1 = x \triangleleft a.x(y).\overline{z}\langle y\rangle.x \triangleleft b.\mathbf{0}.$$

Clearly, P_1 implements three different actions on x: first a selection, then an input of a channel y, and finally another selection. Before the second selection, the output on z forward y.

Rather than considering a sequential protocol involving the three actions on x, we will consider the two selections on x as indicating two independent branches of the same protocol, as specified by the session coalgebra in Figure 15.

Accordingly, we have the following typing derivation for P_1:

$$\textsc{T-Sel}\ \dfrac{\textsc{T-In}\ \dfrac{\textsc{T-Out}\ \dfrac{\textsc{T-Sel}\ \dfrac{\textsc{T-Inact}\ \dfrac{\mathrm{un}(z : C, x : B)}{z : C, x : B \vdash \mathbf{0}}}{z : C, x : T \vdash x \triangleleft b.\mathbf{0}} \qquad \texttt{int} \sqsubseteq \texttt{int}}{z : U, x : T, y : R \vdash \overline{z}\langle y\rangle.x \triangleleft b.\mathbf{0}} \qquad \texttt{int} \sqsubseteq \texttt{real}}{z : U, x : A \vdash x(y).\overline{z}\langle y\rangle.x \triangleleft b.\mathbf{0}}}{z : U, x : T \vdash x \triangleleft a.x(y).\overline{z}\langle y\rangle.x \triangleleft b.\mathbf{0}}$$

Recall the alternative completed protocol T_{end} of Figure 9. Just like regular end, T_{end} allows no interactions on the channel, but it does not cause a “un” type to be unparallelizable. The following example illustrates a more complex session coalgebra that uses T_{end}, here called q_2.

Figure 16 describes a parallelizable unrestricted state T such that each copy of a channel in state T can only do a single receive. However, because it is unrestricted, we can still copy the channel across threads and read a value per copy. We can even read infinitely many values through replication:

$$\begin{array}{lll} x : T & \nvdash & x(y_1).x(y_2).x(y_3).\mathbf{0} \\ x : T & \vdash & x(y_1).\mathbf{0} \mid x(y_2).\mathbf{0} \mid x(y_3).\mathbf{0} \\ x : T & \vdash & !(x(y).\mathbf{0}) \end{array}$$

Such a structure might be interesting in combination with session delegation. Figure 17 takes the type of the previous example, and replaces `int` by the session U = `!int. ?bool.` end as the type to be communicated over the unrestricted channel.

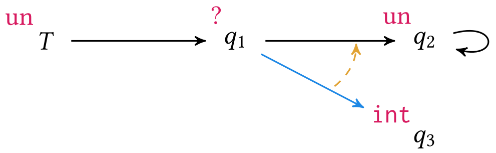


Fig. 16. Session coalgebra using an alternative completed protocol.

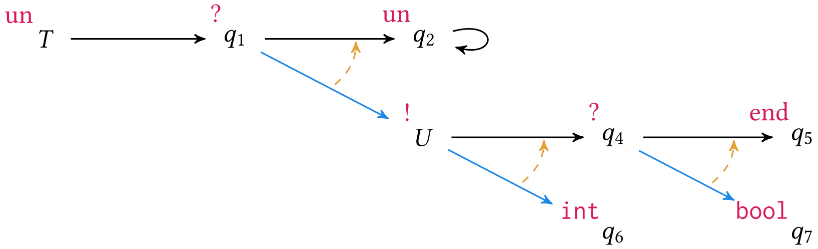


Fig. 17. Session coalgebra using session delegation and an alternative completed protocol.

The unrestricted channel still allows at most one read per thread, but now each such read establishes a linear session of type U. Let $P_i = \overline{y_i}\langle z_i \rangle.y_i(b_i).\mathbf{0}$ be a family of processes that each use a single channel of type U, then

$$y_i : U, z_i : \texttt{int} \quad \vdash \quad P_i.$$

We can use a channel of type T to run arbitrarily many instances of P_i in parallel:

$$x : T,\ z_1 : \texttt{int},\ z_2 : \texttt{int},\ z_3 : \texttt{int} \quad \vdash \quad x(y_1).P_1 \ \mid\ x(y_2).P_2 \ \mid\ x(y_3).P_3$$
$$x : T,\ z_0 : \texttt{int} \quad \vdash \quad !(x(y_0).P_0)$$

However, we still cannot use the channel to create multiple sessions of type U in a single thread. For any context Γ and process Q, we have

$$\Gamma,\ x : T \quad \nvdash \quad x(y_1).x(y_2).Q.$$

6 ALGORITHMIC TYPE CHECKING

As observed by Vasconcelos [61], the typing rules describe what well-formed processes look like, but do not directly allow us to decide whether an arbitrary process is well-formed or not. This is because, beforehand, we do not know:

(1) Which type to introduce in reading (Rule T-In) or scope restriction (Rule T-Res), or
(2) How to split the context among processes in parallel composition (Rule T-Par).

We closely follow Vasconcelos [61] to address these issues. Rather than trying to infer the introduced types, we augment the language of processes in Figure 10 with type annotations:

$$P ::= \ldots \mid (\nu xy : T)\, P \mid x(y : T).P.$$

Recall that we assumed all session coalgebras we work with come equipped with a duality function, so we only need to annotate one type for scope restrictions. That is, in writing $(\nu xy : T)\, P$, we assume that T is the type of the first variable, x, and that its dual $\overline{T}$ is the type of the second variable, y. Other productions are kept unchanged.

The key idea to circumvent splitting is that a process $P \mid Q$ is checked as follows: We first pass the entire context to P, keeping track of all linear variables used and unused; then, we remove the used linear variables from the context given to Q. To do this, we consider rules of the form

$$\Gamma \vdash \mathbf{0}; \Gamma \qquad \frac{\Gamma_1 \vdash P; \Gamma_2 \quad \Gamma_1 = \Gamma_2}{\Gamma_1 \vdash\, !P; \Gamma_2} \qquad \text{[A-Inact][A-Rep]}$$

$$\frac{\Gamma_1 \vdash P \ ;\ \Gamma_2 \quad \Gamma_2 \vdash Q \ ;\ \Gamma_3}{\Gamma_1 \vdash P \mid Q \ ;\ \Gamma_3} \qquad \frac{\Gamma_1, x : T, y : \overline{T} \vdash P; \Gamma_2}{\Gamma_1 \vdash (\nu xy : T)P \ ;\ \Gamma_2 \div \{x, y\}} \qquad \text{[A-Par][A-Res]}$$

$$\frac{c(T) = (?, f) \quad f(d) \sqsubseteq U \quad \Gamma_1,\ y : U,\ x : f(*) \vdash P \ ;\ \Gamma_2}{\Gamma_1, x : T \vdash x(y : U).P \ ;\ \Gamma_2 \div \{x, y\}} \qquad \text{[A-In]}$$

$$\frac{c(T) = (!, f) \quad U \sqsubseteq f(d) \quad \Gamma_1,\ x : f(*) \vdash P \ ;\ \Gamma_2}{\Gamma_1, x : T, y : U \vdash \overline{x}\langle y \rangle.P \ ;\ \Gamma_2 \div \{x\}} \qquad \text{[A-Out]}$$

$$\text{[A-Branch]}$$

$$\frac{c(T) = (\&, L_1, f) \quad L_1 \subseteq L_2 \quad \Gamma_1, x : f(l) \vdash P_l \ ;\ \Gamma_l \quad \Gamma_2 = \Gamma_l \div \{x\} \quad \forall l \in L_2}{\Gamma_1, x : T \vdash x \triangleright \{l : P_l\}_{l \in L_2} \ ;\ \Gamma_2}$$

$$\frac{c(T) = (\oplus, L, f) \quad \Gamma_1, x : f(l) \vdash P_l \ ;\ \Gamma_2 \quad l \in L}{\Gamma_1, x : T \vdash x \triangleleft l.P_l \ ;\ \Gamma_2 \div \{x\}} \qquad \text{[A-Sel]}$$

$$\frac{c(T) = (\mathrm{un}, f) \quad \mathrm{par}(T) \quad \Gamma_1, x : f(*) \vdash P; \Gamma_2}{\Gamma_1, x : T \vdash P \ ;\ (\Gamma_2 \div \{x\}), x : T} \qquad \text{[A-Unpack]}$$

Fig. 18. Algorithmic Type Checking Rules.

$$\Gamma \div \emptyset = \Gamma \qquad \frac{\Gamma_1 \div F = \Gamma_2, x : T \quad \mathrm{un}(T)}{\Gamma_1 \div (F, x) = \Gamma_2} \qquad \frac{\Gamma_1 \div F = \Gamma_2 \quad x \notin \mathit{dom}(\Gamma_2)}{\Gamma_1 \div (F, x) = \Gamma_2}$$

Fig. 19. Context Difference.

$\Gamma_1 \vdash P \ ;\ \Gamma_2$, where Γ_1 and P are the *input* to the algorithm and Γ_2 is the *output*: in an execution, Γ_2 is the subset of Γ_1 containing only those variables that had unrestricted types or were not used in P. We say subset, because we want these variables, if present, to have the same type in Γ_2 as in Γ_1.

Figure 18 lists the algorithmic versions of the typing rules. Rule A-Par, for example, checks parallel processes as just described. By construction, Γ_2 is one part of the context split required to instantiate Rule T-Par. The linear variables of the other part is exactly those that are present in Γ_1 but not in Γ_2. This change in A-Par requires adjusting the other rules. First, we need the algorithm to accept, even when we do not fully complete all sessions of Γ_1 in P. We do this by unconditionally accepting the terminated process. Note that acceptance of the algorithm now only implies well-formedness if the returned context is unrestricted.

Second, the algorithm needs to remove linear variables from the output as we use them. We do not, however, want to remove any variable that has a linear type, as that would allow us to accept processes that do not complete all linear sessions. Thus, we introduce the context difference operator $\div$ in Figure 19. We write $\Gamma \div \{x\}$ to denote the context of all variable/type pairs in Γ minus a potential pair including x, but is only defined if $(x, T) \in \Gamma$ implies that T is unrestricted.

We elaborate on Rule A-Branch; the algorithm is called once for every branch, yielding a context Γ_l each time. Excluding x, each branch must use the exact same set of linear variables. Thus, we require that all these contexts are equal up to a potential (x, U_l) pair. Specifically, there is some Γ_2 such that $\Gamma_2 = \Gamma_l \div \{x\}$ for any $l \in L_2$, this Γ_2 is the output context.

To motivate this, consider a type $T = \&\{a : T_{un},\ b : \mathrm{end}\}$, where T_{un} is some unrestricted type distinct from end, and some process $P = x \triangleright \{a : \mathbf{0},\ b : \mathbf{0}\}$. Let Γ be some unrestricted context, $\mathbf{0}$ is well-formed for both $\Gamma, x : T_{un}$ and $\Gamma, x : \text{end}$; the algorithm agrees:

$$\Gamma, x : T_{un} \vdash \mathbf{0} \; ; \; (\Gamma, x : T_{un}),$$
$$\Gamma, x : \text{end} \vdash \mathbf{0} \; ; \; (\Gamma, x : \text{end}).$$

The resulting contexts are not equal. P is well-formed for Γ, so we have to allow x to have different types in the output of different branches in a complete algorithm. Rules A-In, A-Out, and A-Sel do not have multiple branches to check, but follow similar ideas. When introducing a new variable, either through a read or scope restriction, the new variable is also removed from the output. Rule A-Unpack only unpacks unrestricted types. We want those to have the same type in the input as in the output, so we remove the variable and add a pair with the original type.

Take, for example, the process

$$x : \texttt{?int},\; y : \texttt{?int} \quad \vdash \quad x(z_1).\mathbf{0} \mid y(z_2).\mathbf{0}.$$

The variables are split correctly, and both split contexts are unrestricted when the process is completed, thus it is well-formed.

If, however, the left process did not complete the linear session, then the context difference would not have been defined. Take one such process:

$$x : \texttt{?int.?int},\; y : \texttt{?int} \quad \nvdash \quad x(z_1).\mathbf{0} \mid y(z_2).\mathbf{0}.$$

We succeed in checking the terminated process of the left part:

$$x : \texttt{?int},\; y : \texttt{?int} \quad \vdash \quad \mathbf{0} \; ; \quad (x : \texttt{?int}, y : \texttt{?int}).$$

But x has a linear type in the output. $(x : \texttt{?int}, y : \texttt{?int}) \div \{x\}$ is undefined, so the algorithm rejects this input entirely. The process was indeed not well-formed, and no further parallel processes could fix it; the rejection is expected.

For each process and context there is at most one applicable algorithmic rule: Which one is directed by the process syntax and unrestrictedness of a channel being interacted with.

Under the same assumptions as before (i.e., the session coalgebra describing the types is finitely generated), this induced type checking algorithm is decidable, sound, and complete with respect to the typing rules defined in Section 5.

Theorem 6.1 (Decidability). *The type checking algorithm terminates in finite time for every input, assuming a finitely generated session coalgebra.*

Proof. The input of the algorithm is a finite context Γ and a process, a finite expression, P. Just like a proof of well-formedness is a tree of typing rules, an execution of the algorithm produces a tree of algorithmic rules. For any non-Unpack node in the tree, the rule removes some element from the process(es) to be recursively type checked. The process of any such node is thus strictly larger than the concatenation of all its childrens' processes. Because a process is a finite expression, one can only remove finitely many elements; hence, there can only be finitely many of these non-Unpack nodes in the tree.

For example, T-Par checks a process $P \mid Q$. Its children check P and Q, and $length(P) + length(Q) < length(P \mid Q)$, in terms of their string concatenation.

A-Unpack does not change the process, but it does change the type of a variable. Because we assumed finitely generated coalgebra, each un state can either be unpacked into a non-un state, or forms a finite cycle of purely un states. In the former case, the algorithm proceeds with one of the other, non-A-Unpack, rules. In the latter case, we know the channel in question does not allow any interactions. Because the algorithm only tries to unpack types of variables that the process in question interacts with, detecting such a cycle immediately allows the algorithm to reject. There are finitely many variables in a context, finitely many non-Unpack nodes in the tree and finitely many Unpack nodes per regular node. Thus, there are finitely many nodes in total.

All of the non-recursive premises are decidable (see Theorem 4.16). As such, a finite tree corresponds to an execution that finishes in finite time. □

Having defined algorithmic typechecking, we can go back to the process syntax that we used to define our typing rules (Figure 10) by erasing type annotations in input and restriction operators. Let $erase(\cdot)$ denote a function on processes defined as

$$erase((\nu xy : T).Q) = (\nu xy).erase(Q),$$
$$erase(x(y : T).Q) = x(y).erase(Q)$$

and as an homomorphism on the remaining process constructs. We have:

Theorem 6.2 (Correctness). *$\Gamma_1 \vdash P$ iff there is an annotated process P' with $P = erase(P')$, $\Gamma_1 \vdash P'; \Gamma_2$ and $un(\Gamma_2)$.*

The rest of this section is dedicated to proving this theorem. Correctness is generally broken down in two parts: *soundness* and *completeness*. An algorithm is sound if every accepted program is valid (the right-to-left implication) and it is complete if every valid program—annotated with the correct types—is accepted (left-to-right). We start by formalising the algorithm output. Let $\mathcal{L}(\Gamma)$ and $\mathcal{U}(\Gamma)$ denote the linear and unrestricted parts, respectively, of any context Γ. That is,

$$\mathcal{L}(\Gamma) = \{(x : T) \in \Gamma \mid \mathrm{lin}(T)\},$$
$$\mathcal{U}(\Gamma) = \{(x : T) \in \Gamma \mid \mathrm{un}(T)\}.$$

Lemma 6.3 (Algorithmic monotonicity). *If $\Gamma_1 \vdash P; \Gamma_2$, then*

(1) $\Gamma_2 \subseteq \Gamma_1$, and
(2) $\mathcal{U}(\Gamma_2) = \mathcal{U}(\Gamma_1)$.

Proof. The proof is an induction on the structure of the execution tree. We elaborate on A-Unpack. Suppose $\Delta_1, x : T \vdash P \ ; (\Delta_2 \div x), x : T$, for some unrestricted T. We start from the premise of the rule

$$\Delta_1, x : \delta(T)(*) \vdash P \ ; \Delta_2.$$

By induction,

$$\Delta_2 \subseteq \Delta_1, x : \delta(T)(*),$$
$$\mathcal{U}(\Delta_2) = \mathcal{U}(\Delta_1, x : \delta(T)(*)).$$

Neither of these relations is invalidated by removing x from both contexts:

$$\Delta_2 \div x \subseteq \Delta_1,$$
$$\mathcal{U}(\Delta_2 \div x) = \mathcal{U}(\Delta_1).$$

Nor by adding the same $x : T$ pair to both sides.

$$(\Delta_2 \div x), x : T \subseteq \Delta_1, x : T,$$
$$\mathcal{U}((\Delta_2 \div x), x : T) = \mathcal{U}(\Delta_1, x : T).$$

□

In our proof of soundness, we need algorithmic linear strengthening. In A-Par the entire context is passed along to the first process, but the typing rules require a strict split of linear variables. Linear variables that are still present in the output (thus, not referenced in the process) are safe to remove from the input context.

Lemma 6.4 (Algorithmic linear strengthening). *If $lin(T)$ and $\Gamma_1, x : T \vdash P \ ; \Gamma_2, x : T$ can be shown by tree of height n, then $\Gamma_1 \vdash P \ ; \Gamma_2$ can also be shown by a tree of height n.*

Proof The proof requires an inductive analysis on the structure of the execution tree. Let us detail two cases, the rest are done in a similar fashion.

When the root of the execution tree is A-Par, suppose that

$$\Delta_1, x : T \vdash P \mid Q \ ; \Delta_3, x : T.$$

Then, by premise of the rule

$$\Delta_1, x : T \vdash P \ ; \Delta_2, x : T \qquad \text{and} \qquad \Delta_2, x : T \vdash Q \ ; \Delta_3, x : T.$$

Note that monotonicity and $x : T$ in the output context imply that $x : T$ is an element of the input and intermediate contexts as well. Since these are the direct subtrees of the root, they are of height $n - 1$.

We can use induction on both to get $\Delta_1 \vdash P \ ; \Delta_2$ and $\Delta_2 \vdash Q \ ; \Delta_3$, with trees of height $n - 1$, so we can construct a tree of height n with A-Par at the root, and $\Delta_1 \vdash P \mid Q \ ; \Delta_3$ as conclusion.

When the root is A-In, suppose

$$\Gamma_1, x : T, z : V \vdash z(y : U).P \ ; \Gamma_2, x : T.$$

The premise of the rule tells us, for the continuation type $V_* = \delta(V)(*)$,

$$\Gamma_1, x : T, y : U, z : V_* \vdash P \ ; \Gamma_3, x : T.$$

Again, this is a direct subtree, so it is of height $n - 1$. The inductive hypothesis lets us remove x from both sides, while preserving the height:

$$\Gamma_1, y : U, z : V_* \vdash P \ ; \Gamma_3.$$

The rule specifies that $\Gamma_2, x : T = (\Gamma_3, x : T) \div \{z, y\}$, for some Γ_3. The variable x is preserved through the difference, so it must be distinct from z and y:

$$(\Gamma_3, x : T) \div \{z, y\} = (\Gamma_3 \div \{z, y\}), x : T.$$

Hence, $\Gamma_3 \div \{z, y\} = \Gamma_2$; the desired result follows from an application of A-In, resulting in a tree of height n:

$$\Gamma_1, z : V \vdash z(y : U).P \ ; \Gamma_2. \qquad \square$$

Finally, we can prove soundness.

Theorem 6.5. *$\Gamma_1 \vdash P \ ; \Gamma_2$ and $un(\Gamma_2)$ implies $\Gamma_1 \vdash erase(P)$.*

Proof. By induction on the height of inference trees, using a case analysis on the last rule used. Cases other than A-Par are straightforward; let us illustrate the procedure with A-Branch. Let $c(T) = (\&, L_1, f)$ and suppose that

$$\Gamma_1, x : T \vdash x \triangleright \{l : P_l\}_{l \in L_2} \ ; \Gamma_2.$$

The premise says that $\Gamma_1, x : f(l) \vdash P_l \ ; \Gamma_2$ for every $l \in L_1$. By induction, $\Gamma_1, x : \delta(T)(l) \vdash erase(P_l)$ for the same $l \in L_1$. The result is directly implied by T-Branch.

Let us elaborate on A-Par. Suppose that $\Gamma_1 \vdash P \mid Q \ ; \Gamma_3$. We know that both processes are accepted, as $\Gamma_1 \vdash P \ ; \Gamma_2$ and $\Gamma_2 \vdash Q \ ; \Gamma_3$. Obviously, any type in $\mathcal{L}(\Gamma_2)$ is linear, so we can strengthen the first premise: $\Gamma_1 - \mathcal{L}(\Gamma_2) \vdash P \ ; \Gamma_2 - \mathcal{L}(\Gamma_2)$. The output is trivially unrestricted (all linear types were removed), so we can apply the inductive hypothesis to P. Similarly, Γ_3 is unrestricted by assumption, so we can apply the inductive hypothesis to Q as well:

$$\Gamma_1 - \mathcal{L}(\Gamma_2) \vdash \mathit{erase}(P),$$
$$\Gamma_2 \vdash \mathit{erase}(Q).$$

Monotonicity tells us that $\Gamma_1 = (\Gamma_1 - \mathcal{L}(\Gamma_2)) \circ \Gamma_2$ is in the context split relation. We can conclude $\Gamma_1 \vdash \mathit{erase}(P \mid Q)$. □

Just like we can strengthen a context by removing variables, we can weaken it by adding variables. The linearity of these additions does not matter.

Lemma 6.6 (Algorithmic Weakening). *If $\Gamma_1 \vdash P \ ; \Gamma_2$, then $\Gamma_1, x : T \vdash P \ ; \Gamma_2, x : T$ for any pair $x : T$ where $x \notin \mathit{dom}(\Gamma)$ and x does not occur bound in P.*

Proof. A fairly simple inductive analysis. We detail a single case.

Suppose that $\Gamma_1 \vdash P \mid Q \ ; \Gamma_3$. We can apply induction to both premises, yielding $\Gamma_1, x : T \vdash P \ ; \Gamma_2, x : T$ and $\Gamma_2, x : T \vdash P \ ; \Gamma_3, x : T$. Therefore, $\Gamma_1, x : T \vdash P \mid Q \ ; \Gamma_3, x : T$. □

Theorem 6.7 (Algorithmic Completeness). *If $\Gamma_1 \vdash P$, then there exists a P' with $\mathit{erase}(P') = P$, $\Gamma_1 \vdash P \ ; \Gamma_2$ and Γ_2 unrestricted.*

Proof. An inductive analysis on the height of the tree of typing rules. We detail two inductive cases.

Suppose the root of the tree is [T-Rep], then the process being checked must be $!P$. That is, $\Gamma_1 \vdash !P$. By the inductive hypothesis, there is some P' such that $\Gamma_1 \vdash P' \ ; \Gamma_2$ and $P = \mathit{erase}(P')$. Context Γ_1 is unrestricted, by premise of the typing rule, and Γ_2 is a subset containing at least all unrestricted variables. Hence, the two must be equal, implying $\Gamma_1 \vdash !P' \ ; \Gamma_2$ with unrestricted Γ_2.

Suppose the root is scope restriction, i.e., $\Gamma_1 \vdash (\nu xy)Q$. By that rule, there exists some types T and U such that $\Gamma_1, x : T, y : U \vdash Q$ and $T \perp U$. Since duality is unique up to bisimulation (Proposition 4.10), and the duality function yields dual states, we have that $\overline{T} \sim U$. We use that we can freely substitute bisimilar types (Corollary 5.6) to conclude $\Gamma_1, x : T, y : \overline{T} \vdash Q$. Furthermore, this substitution is height-preserving, so we can apply the inductive hypothesis and get some annotated process Q' such that $Q = \mathit{erase}(Q')$ and $\Gamma_1, x : T, y : \overline{T} \vdash \Gamma_2$, where Γ_2 is unrestricted. In particular, then the types of x and y, if present in Γ_2, must be unrestricted, so $\Gamma_3 = \Gamma_2 \div \{x, y\}$ is a well-defined, unrestricted context. Continuing, $Q = \mathit{erase}(Q')$ implies that $(\nu xy)Q = \mathit{erase}((\nu xy : T)Q')$, so A-Res is applicable, yielding the desired result: $\Gamma_1 \vdash (\nu xy : T)Q'; \Gamma_3$. □

Correctness, Theorem 6.2, follows directly from soundness and completeness.

7 CONTEXT-FREE SESSION TYPES

Besides the session types we have considered so far, there is also a strictly more powerful system called *context-free* session types [58]. There is a strong resemblance between regular languages and the session types as we have seen them. Indeed, context-free session types expand the discipline to be more like context-free languages.

At the basis lies the sequencing operator ";". A session type $T; U$ will first execute T, and once that is finished, execute U. Combined with recursion, this allows for interesting interactions that we could not describe before.

Example 7.1. Suppose our protocol dictates that we send some number of integers, and then read exactly as many Boolean values. Notice that this behaviour is not expressible in the session

$$
\begin{aligned}
q &::= \mathsf{lin} \mid \mathsf{un} \\
p &::= \; ?T \mid \; !T \mid \&\{\ell_i : S_i\}_{i\in I} \mid \oplus\{\ell_i : S_i\}_{i\in I} \\
S &::= \; \mathsf{end} \mid q\,p \mid X \in \mathsf{Var} \mid \mu X.S \mid S \,;\, S \\
T &::= \; S \mid d \in D
\end{aligned}
$$

Fig. 20. Context-free session types over sets of basic data types D and of variables Var.

types we have discussed, but we can define a session coalgebra for it. That session coalgebra will, however, not be finitely generated (Definition 4.14):

$$
T = \mu X. \oplus \begin{cases} \mathit{next}: & \mathtt{!int}\,;\, X \,;\, \mathtt{?bool}\,;\, \mathrm{end} \\ \mathit{quit}: & \mathrm{end} \end{cases}
$$

The grammar for these context-free session types is very similar to before (i.e., as defined in Figure 6); we remove the continuations from in- and output, and replace it with the sequencing operator between session types.

We let CFType be the set of all T in the grammar of Figure 20 that are closed and contractive, where contractive means that any type variable X must occur in or after a pretype p. For instance, the type $\mu X.\ \mathtt{?int}\,;\, X$ is contractive, while $\mu X.\ X\,;\, X$ is not.

7.1 Session Coalgebras with Stacks

In principle, it is possible to equip the set of context-free session types with a session coalgebra structure by using a stack of active states. The issue with this is, however, that the coalgebra is not finitely generated any longer, because the stacks may grow indefinitely and thus a state may reach infinitely many other states. This is the same issue why one has to consider push-down automata to recognise context-free language instead of (non-)deterministic finite automata. The goal of this section is to devise a notion of session coalgebra with stack, which allows the finitary presentation of sessions with sequential composition, together with trace semantics and a determinisation procedure. Trace semantics allows us to understand the behaviour of session coalgebras with stack, while the determinisation procedure allows us to turn them back into session coalgebras, albeit not finitely generated ones.

We first introduce session coalgebras with stacks. The idea is that any state can push states on a stack, which keeps track of the states that will be visited after the current execution has terminated in an end- or bsc-state.

Definition 7.2 (Stack Session Coalgebras). Let A be the set of labels as in Definition 3.2 and define the A-indexed family C, which differs from B only for communication labels, by

$$
C_a = \begin{cases} \{đ\}, & a = (\mathrm{com}, p), \\ B_a, & \text{otherwise.} \end{cases}
$$

We define the functor $G\colon \mathbf{Set} \longrightarrow \mathbf{Set}$ on objects by

$$
G(X) = \left(\coprod_{a\in A} X^{C_a} \right) \times X^{*} \qquad \text{and} \qquad G(f)(a, \beta, u) = (a, f \circ \beta, f^{*}(u)),
$$

where $(-)^{*}$ is the list functor. A coalgebra (X, c) for the functor G is called a *stack session coalgebra.*

Note that G is isomorphic to the polynomial functor given by $X \mapsto \coprod_{(a,n)\in A\times\mathbb{N}} X^{C_a + N_n}$, where $N_n = \{1, \ldots, n\}$. This makes G again a very well-behaved functor, which we will use later on.

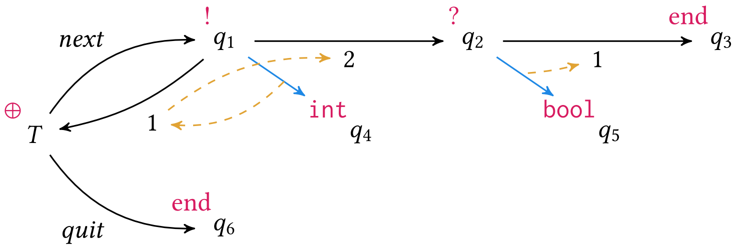


Fig. 21. A stack-based session coalgebra.

For instance, we can easily define bisimilarity for a session coalgebra $c\colon X \to G(X)$ as the greatest fixpoint of the map $c^* \circ g_\sim$ for a relation lifting $g_\sim$ that is defined analogous to $f_\sim$ from Definition 4.2.

Stack session coalgebras should, intuitively, be an extension of session coalgebras with the possibility of composing executions. This intuition is confirmed by the following lemma.

Lemma 7.3. *There is a natural transformation $\rho\colon F \to G$ defined by*

$$\rho_X(\mathrm{com}, p, \beta) = (\mathrm{com}, p, \mathscr{d} \mapsto \beta(\mathscr{d}), [\beta(*)])$$

$$\rho_X(a, \beta) = (a, \beta, \varepsilon)\,, \qquad a \notin \{\mathrm{com}\} \times P,$$

where $[-]$ denotes the singleton list and ε the empty list. This natural transformation induces a functor $\mathbf{CoAlg}(F) \to \mathbf{CoAlg}(G)$ *between the categories of session coalgebras without and with stack.*

Proof. We only need to check naturality of ρ, the existence of the induced functor is standard. Let $f\colon X \to Y$ be a map. We need to prove that $\rho_Y \circ Ff = Gf \circ \rho_X$, where we write Ff instead of $F(f)$ for the application of F to f. This shorter notation will be used only in proofs in this section. For $p \in P$ and $\beta\colon \{*, \mathscr{d}\} \to X$, we have

$$\begin{aligned}(\rho_Y \circ Ff)(\mathrm{com}, p, \beta) &= \rho_Y(\mathrm{com}, p, f \circ \beta)\\ &= (\mathrm{com}, p, \mathscr{d} \mapsto (f \circ \beta)(\mathscr{d}), [(f \circ \beta)(*)])\\ &= (\mathrm{com}, p, f \circ (\mathscr{d} \mapsto \beta(\mathscr{d})), f^*[\beta(*)])\\ &= (Gf)(\mathrm{com}, p, \mathscr{d} \mapsto \beta(\mathscr{d}), [\beta(*)])\\ &= (Gf \circ \rho_Y)(\mathrm{com}, p, \beta)\,.\end{aligned}$$

Similarly, we also have for all other (a, β) that

$$(\rho_Y \circ Ff)(a, \beta) = (a, f \circ \beta, \varepsilon) = (a, f \circ \beta, f^*(\varepsilon)) = (Gf \circ \rho_X)(a, \beta)$$

and thus ρ is a natural transformation. □

The natural transformation from Lemma 7.3 shows that the continuation transitions originating from communication states are equivalent to sequentially composing a simple data transmission with the continuation state. This interpretation allows us to extend the graphical representation of session coalgebras to stack session coalgebras.

Example 7.4. Let us return to the type of Example 7.1, but now as a stack session coalgebra. We present the coalgebra graphically in Figure 21, where numbers on transitions indicate where the state will be in the list of states that is pushed on the stack. Formally, the stack session coalgebra induced by Figure 21 is given as follows. We put $X = \{T\} \cup \{q_k \mid 1 \le k \le 6\}$ and then define

$c\colon X \longrightarrow G(X)$:

$$\begin{aligned}
c(T) &= (\oplus, \beta_0, \varepsilon) \text{ with } \beta_0(\mathit{next}) = q_1 \text{ and } \beta_0(\mathit{quit}) = q_6,\\
c(q_1) &= (!, \beta_1, [T, q_2]) \text{ with } \beta_1(\mathscr{d}) = q_4,\\
c(q_2) &= (?, \beta_2, [q_3]) \text{ with } \beta_2(\mathscr{d}) = q_5,\\
c(q_3) &= (\text{end}, f_\emptyset, \varepsilon),\\
c(q_4) &= (\text{bsc}, \texttt{int}, f_\emptyset, \varepsilon),\\
c(q_5) &= (\text{bsc}, \texttt{bool}, f_\emptyset, \varepsilon),\\
c(q_6) &= (\text{end}, f_\emptyset, \varepsilon).
\end{aligned}$$

More generally, we can see the set of (closed and contractive) context free session types as a stack session coalgebra.

Lemma 7.5. *There is a stack session coalgebra $c_{\mathsf{CFType}}\colon \mathsf{CFType} \longrightarrow G(\mathsf{CFType})$.*

Proof. We define first a map o from pretypes to $G(\mathsf{CFType})$ as follows:

$$\begin{aligned}
o(?T) &= (\text{com}, \text{in}, \mathscr{d} \mapsto T, \varepsilon), & o(!T) &= (\text{com}, \text{out}, \mathscr{d} \mapsto T, \varepsilon),\\
o(\&\{\ell_i : S_i\}_{i\in I}) &= (\text{branch}, \text{in}, \ell_i \mapsto S_i, \varepsilon), & o(\oplus\{\ell_i : S_i\}_{i\in I}) &= (\text{branch}, \text{out}, \ell_i \mapsto S_i, \varepsilon).
\end{aligned}$$

Using this map, can define c_{CFType} recursively:

$$\begin{aligned}
c_{\mathsf{CFType}}(d) &= (\text{bsc}, d, f_\emptyset, \varepsilon),\\
c_{\mathsf{CFType}}(\text{end}) &= (\text{end}, f_\emptyset, \varepsilon),\\
c_{\mathsf{CFType}}(\mathsf{lin}\ p) &= o(p),\\
c_{\mathsf{CFType}}(\mathsf{un}\ p) &= (\text{un}, * \mapsto \mathsf{lin}\ p, \varepsilon),\\
c_{\mathsf{CFType}}(S_1 \,;\, S_2) &= (a, f, u : S_2), \text{ where } c_{\mathsf{CFType}}(S_1) = (a, f, u),\\
c_{\mathsf{CFType}}(\mu X.\ S) &= c_{\mathsf{CFType}}(\mathit{unfold}(\mu X.\ S)).
\end{aligned}$$

This recursion terminates, because all types in CFType are closed and contractive. □

The reader may have noticed that we chose to allow a stack session coalgebra to push an arbitrary, but finite, number of states on the stack. This is not strictly necessary, as it could be simulated by repeated pushing. However, it is more convenient to allow an arbitrary increase of the stack. For instance, in the definition of c_{CFType} above, we turned the sequential composition into adding an element to the end of the list of types that we push on the stack. If we were to restrict this, then we would have to find the left-most type in a sequence of sequential compositions, use this type for the output, and then push the *sequential composition* of the remaining types on the stack. For instance, instead of $c_{\mathsf{CFType}}((\text{end} \,;\, T) \,;\, S) = (\text{end}, [T, S])$, we would have $c_{\mathsf{CFType}}((\text{end} \,;\, T) \,;\, S) = (\text{end}, [T \,;\, S])$. This would be an inconvenient and rather odd definition.

7.2 Trace Semantics of Stack Session Coalgebras

Example 7.4 gave an intuitive description of the behaviour of the stack session coalgebra in Figure 21. As G is a polynomial functor, there is a final stack session coalgebra [1, 62]. However, this final coalgebra will just consist of coinductively defined trees with potentially infinite depth that are labelled in A and where the branching at nodes is given by the family C and lists [60]. Such trees correspond to unravelling stack session coalgebras, but these trees carry no information about the *order of execution*. Instead, we need to enforce the sequential execution in the semantics of stack session coalgebras, which essentially corresponds to visiting the unravelled tree depth-first from left to right. This can be achieved by either altering the functor to a monad that enforces the stack behaviour [26] or by considering traces in stack session coalgebras. We will opt for the latter approach, because it is more elementary and it demonstrates how functors can be used to formalise semantics, called functorial semantics. Traces can then be used to verify the execution order that a type system based on stack session coalgebras has to enforce.

Trace semantics are well-studied for various kinds of transition systems. We will follow here the approach taken in *coalgebraic modal logic* [36, 37], although there is some discrepancy that we are unable to resolve for the time being. This will be discussed toward the end.

To use coalgebraic modal logic, let us recall the contravariant powerset functor $2^{-} \colon \mathbf{Set}^{\mathrm{op}} \longrightarrow \mathbf{Set}$. This functor maps a set X to the set of maps into the two element set 2, or in other words, 2^X is the set of predicates on X. The action of this functor on maps is given by $2^f(p) = p \circ f$. Note that for $f \colon X \longrightarrow Y$, we have $2^f \colon 2^Y \longrightarrow 2^X$, which makes this functor contravariant. An important property of the contravariant powerset functor is that it forms a contravariant adjunction with itself, meaning that the set of maps $X \longrightarrow 2^Y$ is isomorphic to the set of maps $Y \longrightarrow 2^X$. This can be recognised as the transposition of maps.

The variant of coalgebraic modal logic, which we will use here, starts with the following diagram. In this diagram, we use arrows with a tail to indicate contravariant functors. The (covariant) functor L will determine the formulas of a *testing logic.*

$$\begin{tikzcd} \mathbf{Set} \arrow[loop above, "G"] \arrow[r, bend left, tail, "2^{-}"] & \mathbf{Set} \arrow[loop above, "L"] \arrow[l, bend left, tail, "2^{-}"] \end{tikzcd}$$

The idea behind this diagram is that the left-hand side determines the *coalgebraic* behaviour of (stack session) coalgebras, while the right-hand side determines the *logical* behaviour. These two sides of the diagram are intimately related through the contravariant adjunction. We use this picture to provide trace semantics for stack session coalgebras [37].

Definition 7.6. Let T be the set of trace labels given by

$$T = \left(\coprod_{a \in A \setminus \{\mathrm{end}\} \cup \{\mathrm{bsc}\} \times D} C_a \right) + \{\mathrm{end}\} + D,$$

and for $\mathbf{1} = \{*\}$, we define the logic functor $L \colon \mathbf{Set} \longrightarrow \mathbf{Set}$ by

$$L(X) = T \times X + \mathbf{1} \qquad \text{and} \qquad L(f) = \mathrm{id}_T \times f + \mathrm{id}_{\mathbf{1}}\,.$$

Note that T is defined as the coproduct of all C_a with the exception of C_{end} and C_{bsc}. The reason for this is that the latter two sets are empty, but we want to retain the labels in the traces, even though they states labelled with end or bsc are terminal. Also, we use the coproduct here, instead of set union, to guarantee that we obtain a disjoint union. Our goal is to use lists over T as traces for stack session coalgebras. For this, we will make use of the fact that L has the set T^* of lists over T as initial algebra. The following theorem shows that we can compute traces that are compatible with bisimilarity.

Theorem 7.7. *There is a contravariant functor* $\Gamma \colon \mathbf{CoAlg}(G) \longrightarrow \mathbf{Alg}(L)$ *from the category of stack session coalgebras to the category of L-algebras that makes the following diagram commute, where*

$2^{\mathrm{Id}\times\mathrm{Id}^*}(X) = 2^{X\times X^*}$ *is the functor mapping a set* X *to predicates over* $X\times X^*$ *and the downward arrows are the forgetful functors. This makes* Γ *a so-called* lifting *of* $2^{\mathrm{Id}\times\mathrm{Id}^*}$*:*

$$\begin{tikzcd}
\mathbf{CoAlg}(G) \arrow[r, tail, "\Gamma"] \arrow[d] & \mathbf{Alg}(L) \arrow[d] \\
\mathbf{Set} \arrow[r, tail, "2^{\mathrm{Id}\times\mathrm{Id}^*}"] & \mathbf{Set}
\end{tikzcd}$$

Proof. We first define Γ on a coalgebra $c\colon X \to GX$:

$$\Gamma(c)\colon L\big(2^{X\times X^*}\big) \to 2^{X\times X^*}$$

$$\Gamma(c)(*)(x,u) = 1$$

$$\Gamma(c)(\mathrm{end},P)(x,u) = \begin{cases} P(x',u'), & c(x) = (\mathrm{end}, f_\emptyset, v) \text{ and } v + u = x' : u' \\ 0, & \text{otherwise} \end{cases}$$

$$\Gamma(c)(d,P)(x,u) = \begin{cases} P(x',u'), & c(x) = (\mathrm{bsc}, d, f_\emptyset, v) \text{ and } v + u = x' : u' \\ 0, & \text{otherwise} \end{cases}$$

$$\Gamma(c)(a,b,P)(x,u) = \begin{cases} P(\beta(b)), v + u), & c(x) = (a, \beta, v) \\ b0, & \text{otherwise} \end{cases}$$

and for another coalgebra $c'\colon Y \to GY$ and a homomorphism $h\colon c \to c'$, we define

$$\Gamma(h)\colon 2^{Y\times Y^*} \to 2^{X\times X^*},$$

$$\Gamma(h) = 2^{h\times h^*}.$$

Clearly, $\Gamma(\mathrm{id}) = \mathrm{id}$ and $\Gamma(h \circ h') = \Gamma(h') \circ G(h)$. Also the diagram clearly commutes by the definition of Γ.

Note that the case, in which $\mathrm{dom}(\beta) = \emptyset$ means that a is either end or (bsc, d) for some basic type d. In either case, a state is taken from the top of the stack.

Thus, it remains to prove that $\Gamma(h)$ is an algebra homomorphism, which means we have to prove that the following diagram commutes:

$$\begin{tikzcd}
L\big(2^{Y\times Y^*}\big) \arrow[r, "L(\Gamma(h))"] \arrow[d, "\Gamma(c')"] & L\big(2^{X\times X^*}\big) \arrow[d, "\Gamma(c)"] \\
2^{Y\times Y^*} \arrow[r, "\Gamma(h)"] & 2^{X\times X^*}
\end{tikzcd}$$

We will prove this by going through the three cases in the definition of $\Gamma(c')$. The first case is the application to $* \in \mathbf{1}$, where we use that $L(\Gamma h)(*) = *$:

$$(\Gamma h \circ \Gamma c')(*)(x,u) = (\Gamma c')(*)(h(x), h^*(u)) = 1 = (\Gamma c)(*)(x,u) = (\Gamma c \circ L(\Gamma h))(*)(x,u).$$

The second case is the application to end and $P\colon Y \times Y^* \longrightarrow 2$:

$$\begin{aligned}
(\Gamma c \circ L(\Gamma h))(\text{end}, P)(x, u) &= (\Gamma c)(\text{end}, P \circ (h \times h^*))(x, u) \\
&= \begin{cases} (P \circ (h \times h^*))(x', u'), & c(x) = (\text{end}, \beta, v) \text{ and } v + u = x' : u' \\ 0, & \text{otherwise} \end{cases} \\
&= \begin{cases} P(h(x'), h^*(u')), & c(x) = (\text{end}, \beta, v) \text{ and } v + u = x' : u' \\ 0, & \text{otherwise} \end{cases} \\
&= \begin{cases} P(y', w'), & c(x) = (\text{end}, \beta, v) \text{ and } h^*(v) + h^*(u) = y' : w' \\ 0, & \text{otherwise,} \end{cases} \\
&\qquad \text{using } y' : w' = h(x') : h^*(u') = h^*(x' : u') \text{ and } h^*(v + u) = h^*(v) + h^*(u) \\
&= \begin{cases} P(y', w'), & Gh(c(x)) = (\text{end}, h \circ \beta, h^*(v)) \\ & \ \text{and } h^*(v) + h^*(u) = y' : w' \\ 0, & \text{otherwise} \end{cases} \\
&= \begin{cases} P(y', w'), & c'(h(x)) = (\text{end}, \gamma, w) \text{ and } w + h^*(u) = y' : w' \\ 0, & \text{otherwise} \end{cases} \\
&\qquad \text{using } Gh(c(x)) = c'(h(x)), \gamma = h \circ \beta, w = h^*(v) \\
&= (\Gamma c')(\text{end}, P)(h(x), h^*(u)) \\
&= (\Gamma h \circ \Gamma c')(\text{end}, P)(x, u).
\end{aligned}$$

The third case, application to an element in $\{\text{bsc}\} \times D$ and a predicate P follows the same reasoning.

The last case is the application to $a \in A \setminus \{\text{end}\}$, $b \in C_a$ and $P\colon Y \times Y^* \longrightarrow 2$:

$$\begin{aligned}
(\Gamma c \circ L(\Gamma h))(a, b, P)(x, u) &= (\Gamma c)(a, b, P \circ (h \times h^*))(x, u) \\
&= \begin{cases} (P \circ (h \times h^*))(\beta(b)), v + u), & c(x) = (a, \beta, v) \\ 0, \text{otherwise} \end{cases} \\
&= \begin{cases} P(h(\beta(b)), h^*(v + u)), & c(x) = (a, \beta, v) \\ 0, \text{otherwise} \end{cases} \\
&= \begin{cases} P(\gamma(b), w + h^*(u)), & Gh(c(x)) = (a, \gamma, w) \\ 0, \text{otherwise,} \end{cases} \\
&\qquad \text{using } (a, \gamma, w) = (a, h \circ \beta, h^*(v)) \\
&= \begin{cases} P(\gamma(b), w + h^*(u)), & c'(h(x)) = (a, \gamma, w) \\ 0, \text{otherwise} \end{cases} \\
&= (\Gamma c')(a, b, P)(h(x), h^*(u)) \\
&= (\Gamma h \circ \Gamma c')(a, b, P)(x, u).
\end{aligned}$$

This shows that $\Gamma c \circ L(\Gamma h) = \Gamma h \circ \Gamma c'$ in all three cases and $\Gamma(h)$ is thus a functor. □

Using Theorem 7.7 and the fact that L has the set T^* of lists over T as initial algebra, we obtain traces of stack session coalgebras.

Corollary 7.8. *For every stack session coalgebra $c\colon X \longrightarrow G(X)$ there is a map $\tilde{c}\colon X \longrightarrow 2^{T^*}$ that assigns to a state the (finite) traces originating from it. Moreover, whenever two states x and y are bisimilar in c, then $\tilde{c}(x) = \tilde{c}(y)$.*

Proof. Since T^* is an initial L-algebra, we obtain from c in the diagram below on the left, the unique algebra homomorphism h by applying Theorem 7.7. We can then transpose h to $\hat{h}$ on the right and compose it with the map that sends any $x \in X$ to the pair $(x, \varepsilon) \in X \times X^*$. This gives use the map $\tilde{c}$:

$$\begin{tikzcd}
X \arrow[d, "c"] & L(T^*) \arrow[r, "L(h)"] \arrow[d] & L\big(2^{X\times X^*}\big) \arrow[d, "\Gamma(c)"] & X \times X^* \arrow[r, "\hat{h}"] & 2^{T^*} \\
G(X) & T^* \arrow[r, dashed, "h"] & 2^{X\times X^*} & X \arrow[u, "\langle \mathrm{id}, \varepsilon\rangle"] \arrow[ur, "\tilde{c}"] &
\end{tikzcd}$$

Since G is a polynomial functor, a bisimulation on c is given equivalently [54] by a span of coalgebra homomorphisms in $\mathbf{CoAlg}(G)$,

$$\begin{tikzcd}
(X, c) & (R, c_R) \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & (X, c)
\end{tikzcd}$$

where $R \subseteq X \times X$ is a relation with a coalgebra $c_R\colon R \longrightarrow G(R)$ on it, and π_1 and π_2 are the product projections. This means that π_1 and π_2 must be coalgebra homomorphism. By Theorem 7.7, we then obtain *algebra* homomorphisms in the other direction:

$$\begin{tikzcd}
\big(2^{X\times X^*}, \Gamma(c)\big) \arrow[r, "\Gamma(\pi_1)"] & \big(2^{R\times R^*}, \Gamma(c_R)\big) & \big(2^{X\times X^*}, \Gamma(c)\big) \arrow[l, "\Gamma(\pi_2)"']
\end{tikzcd}$$

As $\Gamma(\pi_1)$ and $\Gamma(\pi_2)$ are algebra homomorphisms, the following diagram commutes for $k = 1, 2$:

$$\begin{tikzcd}
L(T^*) \arrow[r, "L(h)"] \arrow[d] & L\big(2^{X\times X^*}\big) \arrow[r, "L(\Gamma(\pi_k))"] \arrow[d, "\Gamma(c)"] & L\big(2^{R\times R^*}\big) \arrow[d, "\Gamma(c_R)"] \\
T^* \arrow[r, dashed, "h"] & 2^{X\times X^*} \arrow[r, "\Gamma(\pi_k)"] & 2^{R\times R^*}
\end{tikzcd}$$

Since T^* is an initial algebra, we obtain $\Gamma(\pi_1) \circ h = \Gamma(\pi_2) \circ h$. We can use this to obtain trace equivalence for bisimilar states: Let $x, y \in X$ be bisimilar, then there is a relation R that gives rise to a span of coalgebras as above. Applying the equality we just obtained, we then have for all $w \in T^*$ that

$$\begin{aligned}
\tilde{c}(x)(w) &= \hat{h}(x, \varepsilon)(w) \\
&= h(w)(x, \varepsilon) \\
&= (\Gamma(\pi_1) \circ h)(w)((x, y), \varepsilon) \\
&= (\Gamma(\pi_2) \circ h)(w)((x, y), \varepsilon) \\
&= \tilde{c}(y)(w)\,.
\end{aligned}$$

Since this holds for all w, we have $\tilde{c}(x) = \tilde{c}(y)$, as desired. □

Let us try to understand this result a bit better. For any coalgebra $c\colon X \longrightarrow G(X)$, $x \in X$ and $w \in T^*$, we obtain from the proof that

$$\tilde{c}(x)(w) = h(w)(x, \varepsilon),$$

and thus for the empty list,

$$\tilde{c}(x)(\varepsilon) = h(\varepsilon)(x, \varepsilon) = \Gamma(c)(*)(x, \varepsilon) = 1\,.$$

This means that the empty list is a trace for any stack session coalgebra. For composed lists, however, we have for $a \neq \mathrm{end}$ that

$$\begin{aligned} \tilde{c}(x)((a,b) : w) &= h((a,b) : w)(x,u) \\ &= \Gamma(c)(a,b,h(w))(x,u) \\ &= h(w)(\beta(b)), v + u) && \text{for } c(x) = (a,\beta,v), \end{aligned}$$

and similarly for $a = \mathrm{end}$. Combining this with the fact that the empty list is always a trace, we have that any list of labels, which is compatible with the transitions of a stack session coalgebra, is a (partial) trace.

More concretely, let us compute some traces for the coalgebra given in Example 7.4.

Example 7.9. We shall refer to the stack session coalgebra induced by Figure 21 as $c \colon X \to G(X)$ with $X = \{T\} \cup \{q_k \mid 1 \leq k \leq 6\}$. As any list of labels that is compatible with c is a trace, we would expect that, for instance, $[(\oplus, \mathit{next}), (!, \not{d}), \mathtt{int}]$, is a trace. Indeed, using the above formulas and

$$\begin{gathered} c(T) = (\oplus, \beta_1, \varepsilon) \text{ with } \beta_1(\mathit{next}) = q_1, \\ c(q_1) = (!, \beta_2, [T, q_2]) \text{ with } \beta_2(\not{d}) = q_4, \end{gathered}$$

we have

$$\begin{aligned} \tilde{c}(T)[(\oplus, \mathit{next}), (!, \not{d}), \mathtt{int}] &= h[(!, \not{d}), \mathtt{int}](q_1, \varepsilon) \\ &= h[\mathtt{int}](q_4, [T, q_2]) \\ &= h(\varepsilon)(T, [q_2]) \\ &= 1\,. \end{aligned}$$

It should be noted that in the second to last line, we stop computing in state T and still have state q_2 remaining on the stack. Using a similar computation, is can be easily verified that the word w with

$$w = (\oplus, \mathit{next}) : (!, \not{d}) : \mathtt{int} : (\oplus, \mathit{quit}) : \mathrm{end} : (?, \not{d}) : \mathtt{bool} : \mathrm{end}$$

is a trace of c and that the computation ends in state q_6 with an empty stack.

Keeping track of the stack allows us to "determinise" stack session coalgebras into session coalgebras [37].

Theorem 7.10. *There is a* determinisation *functor D that turns stack session coalgebras into session coalgebra, such that the following diagram commutes where the downward arrows are again the respective forgetful functors:*

$$\begin{tikzcd} \mathbf{CoAlg}(G) \arrow[r, "D"] \arrow[d] & \mathbf{CoAlg}(F) \arrow[d] \\ \mathbf{Set} \arrow[r, "(-)^*"] & \mathbf{Set} \end{tikzcd}$$

Proof. Toward defining the determinisation functor, we define two natural transformations $\lambda \colon G \to F(-^*)$ and $\sigma \colon F(-^*) \times (-)^* \to F(-^*)$ by putting

$$\begin{aligned} &\lambda_X \colon GX \to F(X^*) \\ &\lambda_X(\mathrm{com}, p, \beta, u) = (\mathrm{com}, p, \beta'), && \text{where } \beta'(\not{d}) = [\beta(\not{d})] \text{ and } \beta'(*) = u \\ &\lambda_X(a, \beta, u) = (a, \beta'), && \text{where } \beta'(s) = \beta(s) : u \end{aligned}$$

and

$$
\begin{aligned}
&\sigma_X \colon F(X^*) \times X^* \longrightarrow F(X^*) && \\
&\sigma_X((\mathrm{com}, p, \beta), v) = (\mathrm{com}, p, \beta'), && \text{where } \beta'(\not{d}) = \beta(\not{d}) \text{ and } \beta'(*) = \beta(*) + v \\
&\sigma_X((a, \beta), u) = (a, \beta'), && \text{where } \beta'(s) = \beta(s) + v.
\end{aligned}
$$

It is straightforward to show that these definitions are natural in X. Using the transformations λ and σ, the determinisation functor is defined on coalgebras $c\colon X \longrightarrow GX$ by

$$
\begin{aligned}
&D(c)\colon X^* \longrightarrow F(X^*), \\
&D(c)(\varepsilon) = (\mathrm{end}, f_{\emptyset}), \\
&D(c)(x : v) = \sigma_X(\lambda_X(c(x)), v),
\end{aligned}
$$

and on homomorphisms $h\colon (X, c) \longrightarrow (Y, c')$ by $D(h) = h^*$. A routine calculation shows that $D(h)$ is a homomorphism from $D(c)$ to $D(c')$ by using naturality of λ and σ. That D preserves identities and composition is immediate by $(-)^*$ being a functor. □

Both Theorem 7.7 and 7.10 follow the outline provided to construct traces and to determinise stack session coalgebras by using coalgebraic modal logic, as it can be found in the coalgebra literature [37]. However, there are some discrepancies that we are unable to resolve:

(1) Theorem 7.7 is typically proven by providing a certain natural transformation. However, it seems unclear how it could be constructed in our case.
(2) Similarly, Theorem 7.10 arises typically from a natural transformation and is, again, unclear how it could be given in our case.

These issues may suggest that we either may have to use a different setup, for example, by replacing the category of sets with the category of monoids to deal with lists, or that the framework of coalgebraic modal logic and trace semantics needs to be extended. We leave this as future work. Theorem 7.7 and 7.10 provide in any case a clear picture akin to functorial semantics.

8 CONCLUDING REMARKS

We have developed a new language-independent foundation for session types by relying on coalgebras. We introduced session coalgebras, which elegantly capture all communication structures of session types, both linear and unrestricted, without committing to a specific syntactic formulation for processes and types. Session coalgebras allow us to rediscover language-independent coinductive definitions for duality, subtyping, and type equivalence. A key idea is to assimilate channel types to the states of a session coalgebra; we demonstrated this insight by deriving a session type system for the π-calculus, unlocking decidability results and algorithmic type checking.

Interesting strands for future work include extending our coalgebraic toolbox to give a language-independent justification to other session-type systems, such as multiparty session types [31] or nested session types [14]. Another line concerns extending our coalgebraic view to include *language-dependent* issues and properties that require a global analysis on session behaviours. Salient examples are *liveness* properties such as (dead)lock-freedom and progress: advanced type systems [6, 13, 38, 45, 46] typically couple types with advanced mechanisms (such as priority-based annotations and strict partial orders), which provide a global insight to rule out the circular dependencies between sessions that are at the heart of stuck processes.

Last, the whole area of coalgebra now becomes available to explore session types. One possible direction is to make use of final coalgebras and modal logic, which would allow us to analyse the behaviour of session coalgebras. This would be particularly powerful in combination with composition operations for session coalgebras to break down protocols and type checking. Another direction is to use session coalgebras to verify other coalgebras that take on the role of the syntactic π-calculus [18, 44] and thereby allowing also for the exploration of other semantics like manifest sharing [5, 6] without resorting to a specific syntax.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their useful remarks and suggestions.

REFERENCES

[1] Jiří Adámek, Stefan Milius, and Lawrence S. Moss. 2021. Initial Algebras, Terminal Coalgebras, and the Theory of Fixed Points of Functors. Retrieved from http://www.stefan-milius.eu.

[2] Jirí Adámek, Stefan Milius, and Jiri Velebil. 2006. Iterative algebras at work. *Math. Struct. Comput. Sci.* 16, 6 (2006), 1085–1131. https://doi.org/10.1017/S0960129506005706

[3] J. Adamek and J. Rosicky. 1994. *Locally Presentable and Accessible Categories*. Cambridge University Press, Cambridge. https://doi.org/10.1017/CBO9780511600579

[4] Steve Awodey. 2010. *Category Theory* (2nd ed.). Number 52 in Oxford Logic Guides. Oxford University Press.

[5] Stephanie Balzer and Frank Pfenning. 2017. Manifest sharing with session types. *Proc. ACM Program. Lang.* 1, ICFP (2017), 37:1–37:29. https://doi.org/10.1145/3110281

[6] Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types. In *Proceedings of the 28th European Symposium on Programming (ESOP'19)*. 611–639. https://doi.org/10.1007/978-3-030-17184-1_22

[7] Giovanni Bernardi and Matthew Hennessy. 2016. Using higher-order contracts to model session types. *Log. Methods Comput. Sci.* 12, 2 (2016). https://doi.org/10.2168/LMCS-12(2:10)2016

[8] Francis Borceux. 2008. *Handbook of Categorical Algebra: Volume 1, Basic Category Theory*. Cambridge University Press.

[9] Mario Bravetti and Gianluigi Zavattaro. 2007. Towards a unifying theory for choreography conformance and contract compliance. In *Proceedings of the 6th International Symposium on Software Composition (SC@ETAPS'07) (Lecture Notes in Computer Science, Vol. 4829)*, Markus Lumpe and Wim Vanderperren (Eds.). Springer, 34–50. https://doi.org/10.1007/978-3-540-77351-1_4

[10] Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. 2013. Behavioral polymorphism and parametricity in session-based communication. In *Proceedings of the 22nd European Symposium on Programming (ESOP'13) (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 330–349. https://doi.org/10.1007/978-3-642-37036-6_19

[11] Samuele Carpineti, Giuseppe Castagna, Cosimo Laneve, and Luca Padovani. 2006. A formal account of contracts for web services. In *Proceedings of the 3rd Conference on Web Services and Formal Methods (WS-FM'06) (Lecture Notes in Computer Science, Vol. 4184)*, Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro (Eds.). Springer, 148–162. https://doi.org/10.1007/11841197_10

[12] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. 2009. A theory of contracts for web services. *ACM Trans. Program. Lang. Syst.* 31, 5 (2009), 19:1–19:61. https://doi.org/10.1145/1538917.1538920

[13] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. 2016. Global progress for dynamically interleaved multiparty sessions. *Math. Struct. Comput. Sci.* 26, 2 (2016), 238–302. https://doi.org/10.1017/S0960129514000188

[14] Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. 2021. Nested session types. In *Proceedings of the 30th European Symposium on Programming (ESOP'21), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS'21) (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 178–206. https://doi.org/10.1007/978-3-030-72019-3_7

[15] Luca de Alfaro and Thomas A. Henzinger. 2001. Interface automata. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE'01)*, A. Min Tjoa and Volker Gruhn (Eds.). ACM, 109–120. https://doi.org/10.1145/503209.503226

[16] Pierre-Malo Deniélou and Nobuko Yoshida. 2012. Multiparty session types meet communicating automata. In *Proceedings of the 21st European Symposium on Programming (ESOP'12) (Lecture Notes in Computer Science, Vol. 7211)*, Helmut Seidl (Ed.). Springer, 194–213. https://doi.org/10.1007/978-3-642-28869-2_10

[17] Pierre-Malo Deniélou and Nobuko Yoshida. 2013. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming (ICALP'13) (LNCS, Vol. 7966)*, Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg (Eds.). Springer, 174–186. https://doi.org/10.1007/978-3-642-39212-2_18

[18] Clovis Eberhart, Tom Hirschowitz, and Thomas Seiller. 2015. An Intensionally Fully-abstract Sheaf Model for Pi. In *Proceedings of the Conference on Algebra and Coalgebra in Computer Science (CALCO'15)*. 86–100. https://doi.org/10.4230/LIPIcs.CALCO.2015.86

[19] Cédric Fournet, C. A. R. Hoare, Sriram K. Rajamani, and Jakob Rehof. 2004. Stuck-free conformance. In *Proceedings of the 16th International Conference on Computer-Aided Verification (CAV'04) (Lecture Notes in Computer Science, Vol. 3114)*, Rajeev Alur and Doron A. Peled (Eds.). Springer, 242–254. https://doi.org/10.1007/978-3-540-27813-9_19

[20] Clément Fumex, Neil Ghani, and Patricia Johann. 2011. Indexed Induction and Coinduction, Fibrationally. In *Proceedings of the Conference on Algebra and Coalgebra in Computer Science (CALCO'11) (Lecture Notes in Computer Science, Vol. 6859)*. Springer, 176–191. https://doi.org/10.1007/978-3-642-22944-2_13

[21] Nicola Gambino and Joachim Kock. 2009. Polynomial functors and polynomial monads. *Math. Proc. Cambridge Philos. Soc.* 154 (June 2009). https://doi.org/10.1017/S0305004112000394

[22] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of typestate-oriented programming. *ACM Trans. Program. Lang. Syst.* 36, 4 (2014), 12:1–12:44. https://doi.org/10.1145/2629609

[23] Simon J. Gay. 2016. Subtyping supports safe session substitution. In *A List of Successes That Can Change the World—Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 9600)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.). Springer, 95–108. https://doi.org/10.1007/978-3-319-30936-1_5

[24] Simon J. Gay and Malcolm Hole. 2005. Subtyping for session types in the pi calculus. *Acta Info.* 42, 2/3 (2005), 191–225.

[25] Simon J. Gay, Peter Thiemann, and Vasco T. Vasconcelos. 2020. Duality of session types: The final cut. *Electr. Proc. Theor. Comput. Sci.* 314 (Apr. 2020), 23–33. https://doi.org/10.4204/eptcs.314.3

[26] Sergey Goncharov, Stefan Milius, and Alexandra Silva. 2014. Towards a Coalgebraic Chomsky Hierarchy. In *Theoretical Computer Science*, Josep Diaz, Ivan Lanese, and Davide Sangiorgi (Eds.). Number 8705 in LNCS. Springer, 265–280. https://doi.org/10.1007/978-3-662-44602-7_21

[27] Daniel Hausmann, Till Mossakowski, and Lutz Schröder. 2006. A coalgebraic approach to the semantics of the ambient calculus. *Theor. Comput. Sci.* 366, 1-2 (2006), 121–143. https://doi.org/10.1016/j.tcs.2006.07.006

[28] Claudio Hermida and Bart Jacobs. 1997. Structural Induction and Coinduction in a Fibrational Setting. *Info. Comput.* 145 (1997), 107–152. https://doi.org/10.1006/inco.1998.2725

[29] Kohei Honda. 1993. Types for dyadic interaction. In *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR'93) (Lecture Notes in Computer Science, Vol. 715)*, Eike Best (Ed.). Springer, 509–523. https://doi.org/10.1007/3-540-57208-2_35

[30] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Proceedings of the European Symposium on Programming (ESOP'98) (LNCS, Vol. 1381)*, Chris Hankin (Ed.). Springer, 122–138. https://doi.org/10.1007/BFb0053567

[31] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *Proceedings of the 35th Symposium on Principles of Programming Languages (POPL'08)*, George C. Necula and Philip Wadler (Eds.). ACM, 273–284. https://doi.org/10.1145/1328438.1328472

[32] Jesse Hughes and Bart Jacobs. 2004. Simulations in coalgebra. *Theor. Comput. Sci.* 327, 1–2 (2004), 71–108. https://doi.org/10.1016/j.tcs.2004.07.022

[33] Bart Jacobs. 2016. *Introduction to Coalgebra: Towards Mathematics of States and Observation.* Number 59 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press. https://doi.org/10.1017/CBO9781316823187

[34] Alex C. Keizer, Henning Basold, and Jorge A. Pérez. 2021. Session coalgebras: A coalgebraic view on session types and communication protocols. In *Proceedings of the 30th European Symposium on Programming (ESOP'21) (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 375–403. https://doi.org/10.1007/978-3-030-72019-3_14

[35] Bartek Klin. 2005. The Least Fibred Lifting and the Expressivity of Coalgebraic Modal Logic. In *Proceedings of the Conference on Algebra and Coalgebra in Computer Science (CALCO'05)*. 247–262. https://doi.org/10.1007/11548133_16

[36] Bartek Klin. 2007. Coalgebraic Modal Logic Beyond Sets. *Electr. Notes Theor. Comput. Sci.* 173 (2007), 177–201. https://doi.org/10.1016/j.entcs.2007.02.034

[37] Bartek Klin and Jurriaan Rot. 2016. Coalgebraic trace semantics via forgetful logics. *Logic. Methods Comput. Sci.* 12, 4 (2016). https://doi.org/10.2168/LMCS-12(4:10)2016

[38] Naoki Kobayashi. 2006. A new type system for deadlock-free processes. In *Proceedings of the International Conference on Concurrency Theory (CONCUR'06) (Lecture Notes in Computer Science, Vol. 4137)*, Christel Baier and Holger Hermanns (Eds.). Springer, 233–247. https://doi.org/10.1007/11817949_16

[39] Sam Lindley and J. Garrett Morris. 2016. Talking bananas: Structural recursion for session types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP'16)*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 434–447. https://doi.org/10.1145/2951913.2951921

[40] Étienne Lozes and Jules Villard. 2011. Reliable contracts for unreliable half-duplex communications. In *Proceedings of the 8th International Workshop on Web Services and Formal Methods (WS-FM'11) (Lecture Notes in Computer Science, Vol. 7176)*, Marco Carbone and Jean-Marc Petit (Eds.). Springer, 2–16. https://doi.org/10.1007/978-3-642-29834-9_2

[41] Stefan Milius. 2010. A Sound and Complete Calculus for Finite Stream Circuits. In *Proceedings of the Symposium on Logic in Computer Science (LICS'10)*. 421–430. https://doi.org/10.1109/LICS.2010.11

[42] Stefan Milius, Marcello M. Bonsangue, Robert S. R. Myers, and Jurriaan Rot. 2013. Rational Operational Models. In *Proceedings of the 29th Conference on the Mathematical Foundations of Programming Semantics (MFPS'13)*. 257–282. https://doi.org/10.1016/j.entcs.2013.09.017

[43] Robin Milner, Joachim Parrow, and David Walker. 1992. A calculus of mobile processes, I. *Info. Comput.* 100, 1 (1992), 1–40. https://doi.org/10.1016/0890-5401(92)90008-4

[44] Ugo Montanari and Marco Pistore. 2005. Structured coalgebras and minimal HD-automata for the pi-calculus. *Theor. Comput. Sci.* 340, 3 (2005), 539–576. https://doi.org/10.1016/j.tcs.2005.03.014

[45] Luca Padovani. 2014. Deadlock and lock freedom in the linear π-calculus. In *Proceedings of the Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science (CSL-LICS'14)*, Thomas A. Henzinger and Dale Miller (Eds.). ACM, 72:1–72:10. https://doi.org/10.1145/2603088.2603116

[46] Luca Padovani, Vasco Thudichum Vasconcelos, and Hugo Torres Vieira. 2014. Typing liveness in multiparty communicating systems. In *Proceedings of the 16th IFIP WG 6.1 International Conference on Coordination Models and Languages (COORDINATION'14), Held as Part of the 9th International Federated Conferences on Distributed Computing Techniques (DisCoTec'14) (Lecture Notes in Computer Science, Vol. 8459)*, Eva Kühn and Rosario Pugliese (Eds.). Springer, 147–162. https://doi.org/10.1007/978-3-662-43376-8_10

[47] Paolo Perrone. 2021. Notes on Category Theory with Examples from Basic Mathematics. Retrieved from https://arxiv:1912.10642.

[48] Damien Pous. 2007. Complete Lattices and Up-To Techniques. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS'07) (LNCS, Vol. 4807)*, Zhong Shao (Ed.). Springer, 351–366. https://doi.org/10.1007/978-3-540-76637-7_24

[49] Jan Rutten. 2000. Universal Coalgebra: A Theory of Systems. *Theor. Comput. Sci.* 249, 1 (2000), 3–80. https://doi.org/10.1016/S0304-3975(00)00056-6

[50] Jan Rutten. 2019. *The Method of Coalgebra: Exercises in Coinduction*. CWI, Amsterdam. Retrieved from http://persistent-identifier.org/?identifier=urn:nbn:nl:ui:18-28550.

[51] Davide Sangiorgi and David Walker. 2001. *The Pi-Calculus—A Theory of Mobile Processes*. Cambridge University Press.

[52] Lutz Schröder. 2008. Expressivity of coalgebraic modal logic: The limits and beyond. *Theor. Comput. Sci.* 390, 2–3 (2008), 230–247. https://doi.org/10.1016/j.tcs.2007.09.023

[53] Alexandra Silva, Marcello M. Bonsangue, and Jan Rutten. 2010. Non-Deterministic Kleene Coalgebras. *Logic. Methods Comput. Sci.* 6, 3 (2010) arXiv:1007.3769. https://doi.org/10.2168/LMCS-6

[54] Sam Staton. 2011. Relating coalgebraic notions of bisimulation. *Logic. Methods Comput. Sci.* 7, 1 (2011), 1–21. https://doi.org/10.2168/LMCS-7(1:13)2011

[55] Robert E. Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.* 12, 1 (1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929

[56] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. 2011. First-class state change in Plaid. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'11)*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 713–732. https://doi.org/10.1145/2048066.2048122

[57] Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.* 5, 2 (1955), 285–309. https://projecteuclid.org:443/euclid.pjm/1103044538.

[58] Peter Thiemann and Vasco T. Vasconcelos. 2016. Context-free session types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP'16)*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 462–475. https://doi.org/10.1145/2951913.2951926

[59] Bernardo Toninho and Nobuko Yoshida. 2019. Polymorphic session processes as morphisms. In *The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy—Essays Dedicated to Catuscia Palamidessi on the Occasion of Her 60th Birthday (Lecture Notes in Computer Science, Vol. 11760)*, Mário S. Alvim, Kostas Chatzikokolakis, Carlos Olarte, and Frank Valencia (Eds.). Springer, 101–117. https://doi.org/10.1007/978-3-030-31175-9_7

[60] Benno van den Berg and Federico de Marchi. 2007. Non-well-founded trees in categories. *Ann. Pure Appl. Logic* 146, 1 (Apr. 2007), 40–59. arXiv:math/0409158 https://doi.org/10.1016/j.apal.2006.12.001

[61] Vasco T. Vasconcelos. 2012. Fundamentals of session types. *Info. Comput.* 217 (2012), 52–70.

[62] James Worrell. 2005. On the final sequence of a finitary set functor. *Theor. Comput. Sci.* 338, 1–3 (2005), 184–199. https://doi.org/10.1016/j.tcs.2004.12.009

Received April 2021; revised January 2022; accepted January 2022