Minimal Session Types

Alen Arslanagić
University of Groningen, The Netherlands

Jorge A. Pérez
University of Groningen, The Netherlands
http://www.jperez.nl/

Erik Voogd
University of Groningen, The Netherlands

Abstract

Session types are a type-based approach to the verification of message-passing programs. They have been much studied as type systems for the π-calculus and for languages such as Java. A session type specifies what and when should be exchanged through a channel. Central to session-typed languages are constructs in types and processes that specify *sequencing* in protocols.

Here we study *minimal session types*, session types without sequencing. This is arguably the simplest form of session types. By relying on a core process calculus with sessions and higher-order concurrency (abstraction-passing), we prove that every process typable with standard (non minimal) session types can be compiled down into a process typed with minimal session types. This means that having sequencing constructs in both processes and session types is redundant; only sequentiality in processes is indispensable, as it can precisely codify sequentiality in types.

Our developments draw inspiration from work by Parrow on behavior-preserving decompositions of untyped processes. By casting Parrow's results in the realm of typed processes, our results reveal a conceptually simple formulation of session types and a principled avenue to the integration of session types into languages without sequencing in types.

2012 ACM Subject Classification Theory of computation → Type structures; Theory of computation → Process calculi; Software and its engineering → Concurrent programming structures; Software and its engineering → Message passing

Keywords and phrases Session types, process calculi, π-calculus

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.23

Category Pearl

Related Version Extended version with omitted proofs: http://arxiv.org/abs/1906.03836.

Supplement Material ECOOP 2019 Artifact Evaluation approved artifact available at https://dx.doi.org/10.4230/DARTS.5.2.5

Funding Work partially supported by the Netherlands Organization for Scientific Research (NWO) under the VIDI Project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software).

Acknowledgements We are grateful to the anonymous reviewers for their remarks and questions. Pérez is also with CWI, Amsterdam and NOVA LINCS – the NOVA Laboratory for Computer Science and Informatics, Universidade Nova de Lisboa, Portugal (Ref. UID/CEC/04516/2019).

1 Introduction

Session types are a type-based approach to the verification of message-passing programs. A session type specifies what and when should be exchanged through a channel; this makes them a useful tool to enforce safety and liveness properties related to communication correctness. Session types have had a significant impact on the foundations of programming languages [15],

but also on their practice [1]. In particular, the interplay of session types and object-oriented languages has received much attention (cf. [8, 7, 14, 11, 2, 18, 25]). In this work, our goal is to understand to what extent session types can admit simpler, more fundamental formulations. This foundational question has concrete practical ramifications, as we discuss next.

In session-typed languages, *sequencing* constructs in types and processes specify the intended structure of message-passing protocols. In the session type $S = ?(\mathsf{Int}); ?(\mathsf{Int}); !\langle \mathsf{Bool} \rangle;$ `end`, sequencing (denoted ';') allows us to specify a protocol for a channel that *first* receives (?) two integers, *then* sends (!) a Boolean, and *finally* ends. As such, S could type a service that checks for integer equality. Sequencing in types goes hand-in-hand with sequencing in processes, which is specified using prefix constructs (denoted '.'). The π-calculus process $P = s?(x_1).s?(x_2).s!\langle b \rangle.\mathbf{0}$ is an implementation of the equality service: it *first* expects two values on name s, *then* outputs a Boolean on s, and *finally* stops. Thus, name s in P conforms to type S. Session types can also specify sequencing within labeled choices and recursion; these typed constructs are also in close match with their respective process expressions.

Originally developed on top of the π-calculus for the analysis of message-passing protocols between exactly two parties [12], session types have been extended in many directions. We find, for instance, multiparty session types [13] and extensions with dependent types, assertions, exceptions, and time (cf. [6, 15] for surveys). All these extensions seek to address natural research questions on the expressivity and applicability of session types theories.

Here we address a different, if opposite, question: *is there a minimal formulation of session types?* This is an appealing question from a theoretical perspective, but seems particularly relevant to the practice of session types: identifying the "core" of session types could enable their integration in languages whose type systems do not have advanced constructs present in session types (such as sequencing). For instance, the Go programming language offers primitive support for message-passing concurrency; it comes with a static verification mechanism which can only enforce that messages exchanged along channels correspond with their declared payload types – it cannot ensure essential correctness properties associated to the structure of protocols. This observation has motivated the development of advanced static verification tools based on session types for Go programs [22, 21].

This paper identifies an elementary formulation of session types and studies its properties. We call them *minimal session types*: these are session types without sequencing. That is, in session types such as '$!\langle U \rangle; S$' and '$?(U); S$', we decree that S can only correspond to `end`, the type of the terminated protocol.

Adopting this elementary formulation entails dispensing with sequencing, which is one of the most distinctive features of session types. While this may appear as a far too drastic restriction, it turns out that it is not: our main result is that for every process P that is well-typed under standard (non minimal) session types, there is a *process decomposition* $\mathcal{D}(P)$ that is well-typed using minimal session types. Intuitively, $\mathcal{D}(P)$ codifies the sequencing information given by the session types (protocols) of P using additional synchronizations. This shows that having sequencing in both types and processes is redundant; only sequencing at the level of processes is truly fundamental. To define $\mathcal{D}(P)$ we draw inspiration from a known result by Parrow [24], who proved that untyped π-calculus processes can be decomposed as a collection of *trios processes*, i.e., processes with at most three nested prefixes [24].

The question of how to relate session types with other type systems has attracted interest in the past. Session types have been encoded into generic types [10] and linear types [5, 3, 4]. As such, these prior studies concern the *relative expressiveness* of session types: where the expressivity of session types stands with respect to that of some other type system. In sharp contrast, we study the *absolute expressiveness* of session types: how session types can be explained in terms of themselves. To our knowledge, this is the first study of its kind.

The process language that we consider for decomposition into minimal session types is HO, the core process calculus for session-based concurrency studied by Kouzapas et al. [19, 20]. HO is a very small language: it supports abstraction-passing only and lacks name-passing and recursion; still, it is also very expressive, because both features can be expressed in it in a fully abstract way. As such, HO is an excellent candidate for a decomposition. Being a higher-order language, HO is very different from the (untyped, first-order) π-calculus considered by Parrow in [24]. Also, the session types of HO severely constrain the range and kind of conceivable decompositions. Therefore, our results are not an expected consequence of Parrow's: essential aspects of our decomposition into processes typable with minimal session types are only possible in a higher-order setting, not considered in [24].

Summing up, in this paper we make the following contributions:

1. We identify the class of *minimal session types* as a simple fragment of standard session types that retains its absolute expressiveness.
2. We show how to decompose processes typable with standard session types into processes typable with minimal session types. We prove that this decomposition satisfies a typability result for a rich typed language that includes labeled choices and recursive types.
3. We develop optimizations of our decomposition that bear witness to its robustness.

The rest of the paper is organized as follows. § 2 summarizes the syntax, semantics, and session type system for HO, the core process calculus for session-based concurrency. § 3 presents the decomposition of well-typed HO processes into minimal session types. The decomposition is presented incrementally, starting with a core fragment that is later extended with further features. § 4 presents optimizations of the decomposition. § 5 elaborates further on related works and § 6 concludes.

2 The Source Language

We recall the syntax, semantics, and type system for HO, the higher-order process calculus for session-based concurrency studied by Kouzapas et al. [19, 20].[1] HO is arguably the simplest language for session types: it supports passing of abstractions (functions from names to processes) but does not support name-passing nor process recursion. Still, HO is very expressive: it can encode name-passing, recursion, and polyadic communication via type-preserving encodings that are fully-abstract with respect to contextual equivalence [19].

2.1 Syntax and Semantics

The syntax of names, variables, values, and HO processes is defined as follows:

$$n, m \;::=\; a, b \;\mid\; s, \overline{s} \qquad u, w \;::=\; n \;\mid\; x, y, z \qquad V, W \;::=\; x, y, z \;\mid\; \lambda x.\, P$$

$$P, Q \;::=\; u!\langle V\rangle.P \;\mid\; u?(x).P \;\mid\; u \triangleleft l.P \;\mid\; u \triangleright \{l_i : P_i\}_{i\in I} \;\mid\; V\, u \;\mid\; P \mid Q \;\mid\; (\nu\, n)P \;\mid\; \mathbf{0}$$

We use $a, b, c, \ldots$ to range over *shared names*, and $s, \overline{s}, \ldots$ to range over *session names*. Shared names are used for unrestricted, non-deterministic interactions; session names are used for linear, deterministic interactions. We write n, m to denote session or shared names, and assume that the sets of session and shared names are disjoint. The *dual* of n is denoted $\overline{n}$; we define $\overline{\overline{s}} = s$ and $\overline{a} = a$, i.e., duality is only relevant for session names. Variables are

[1] We summarize the content from [19, 20] that concerns HO; the notions and results given in [19, 20] are given for HOπ, a super-calculus of HO.

denoted with $x, y, z, \ldots$. An abstraction $\lambda x.\, P$ is a process P with parameter x. *Values* $V, W, \ldots$ include variables and abstractions, but not names. A tuple of variables $(x_1, \ldots, x_k)$ is denoted $\widetilde{x}$ (and similarly for names and values). We use ϵ to denote the empty tuple.

Processes $P, Q, \ldots$ include usual π-calculus output and input prefixes, denoted $u!\langle V\rangle.P$ and $u?(x).P$, respectively. Processes $u \triangleleft l.P$ and $u \triangleright \{l_i : P_i\}_{i\in I}$ are selecting and branching constructs, respectively, commonly used in session calculi to express deterministic choices [12]. Process $V\, u$ is the application which substitutes name u on abstraction V. Constructs for inaction $\mathbf{0}$, parallel composition $P_1 \mid P_2$, and name restriction $(\nu\, n)P$ are standard. HO lacks name-passing and recursion, but they are expressible in the language (see Exam. 2.1 below).

We sometimes omit trailing $\mathbf{0}$'s, so we may write, e.g., $u!\langle V\rangle$ instead of $u!\langle V\rangle.\mathbf{0}$. Also, we write $u!\langle\rangle.P$ and $u?().P$ whenever the exchanged value is not relevant (cf. Rem. 3.7).

Session name restriction $(\nu\, s)P$ simultaneously binds session names s and $\overline{s}$ in P. Functions $\mathtt{fv}(P)$, $\mathtt{fn}(P)$, and $\mathtt{fs}(P)$ denote, respectively, the sets of free variables, names, and session names in P, and are defined as expected. If $\mathtt{fv}(P) = \emptyset$, we call P *closed*. We write $P\{^{u}/_{y}\}$ (resp., $P\{^{V}/_{y}\}$) for the capture-avoiding substitution of name u (resp., value V) for y in process P. We identify processes up to consistent renaming of bound names, writing $\equiv_\alpha$ for this congruence. We shall rely on Barendregt's variable convention, which ensures that free and bound names are different in every mathematical context.

The operational semantics of HO is defined in terms of a *reduction relation*, denoted $\longrightarrow$. Reduction is closed under *structural congruence*, denoted $\equiv$, which is defined as the smallest congruence on processes such that:

$$P \mid \mathbf{0} \equiv P \quad P_1 \mid P_2 \equiv P_2 \mid P_1 \quad P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3 \quad (\nu\, n)\mathbf{0} \equiv \mathbf{0}$$
$$P \mid (\nu\, n)Q \equiv (\nu\, n)(P \mid Q)\ (n \notin \mathtt{fn}(P)) \quad P \equiv Q \text{ if } P \equiv_\alpha Q$$

We assume the expected extension of $\equiv$ to values V. The reduction relation expresses the behavior of processes; it is defined as follows:

$$(\lambda x.\, P)\, u \longrightarrow P\{^{u}/_{x}\} \qquad \text{[App]}$$
$$n!\langle V\rangle.P \mid \overline{n}?(x).Q \longrightarrow P \mid Q\{^{V}/_{x}\} \qquad \text{[Pass]}$$
$$n \triangleleft l_j.Q \mid \overline{n} \triangleright \{l_i : P_i\}_{i\in I} \longrightarrow Q \mid P_j \quad (j \in I) \qquad \text{[Sel]}$$
$$P \longrightarrow P' \Rightarrow (\nu\, n)P \longrightarrow (\nu\, n)P' \qquad \text{[Res]}$$
$$P \longrightarrow P' \Rightarrow P \mid Q \longrightarrow P' \mid Q \qquad \text{[Par]}$$
$$P \equiv Q \longrightarrow Q' \equiv P' \Rightarrow P \longrightarrow P' \qquad \text{[Cong]}$$

Rule [App] defines name application. Rule [Pass] defines a shared or session interaction, depending on the nature of n. Rule [Sel] is the standard rule for labelled choice/selection. Other rules are standard π-calculus rules. We write $\longrightarrow^k$ for a k-step reduction, and $\longrightarrow^*$ for the reflexive, transitive closure of $\longrightarrow$.

We illustrate HO processes and their semantics by means of an example.

▶ **Example 2.1** (Encoding Name-Passing)**.** HO lacks name-passing, and so the reduction

$$n!\langle m\rangle.P \mid \overline{n}?(x).Q \longrightarrow P \mid Q\{^{m}/_{x}\} \tag{1}$$

is not supported by the language. Still, as explained in [19], name-passing can be encoded in a fully-abstract way using abstraction-passing, by "packing" the name m in an abstraction. Let $[\![\cdot]\!]$ be the encoding defined as

$$[\![n!\langle m\rangle.P]\!] = n!\big\langle \lambda z.\ z?(x).(x\, m)\big\rangle.[\![P]\!]$$
$$[\![n?(x).Q]\!] = n?(y).(\nu\, s)(y\, s \mid \overline{s}!\langle \lambda x.\, [\![Q]\!]\rangle)$$

and as an homomorphism for the other constructs. Reduction (1) can be mimicked as

$$\begin{aligned}
[\![n!\langle m\rangle.P \mid \overline{n}?(x).Q]\!] = \;& n!\big\langle\lambda z.\ z?(x).(x\,m)\big\rangle.[\![P]\!] \mid n?(y).(\nu\, s)(y\,s \mid \overline{s}!\langle\lambda x.\,[\![Q]\!]\rangle)\\
\longrightarrow\;& [\![P]\!] \mid (\nu\, s)(\lambda z.\ z?(x).(x\,m)\,s \mid \overline{s}!\langle\lambda x.\,[\![Q]\!]\rangle)\\
\longrightarrow\;& [\![P]\!] \mid (\nu\, s)(s?(x).(x\,m) \mid \overline{s}!\langle\lambda x.\,[\![Q]\!]\rangle)\\
\longrightarrow\;& [\![P]\!] \mid (\lambda x.\,[\![Q]\!])\,m\\
\longrightarrow\;& [\![P]\!] \mid [\![Q]\!]\{m/x\}
\end{aligned}$$

⌟

▶ Remark 2.2 (Polyadic Communication). HO as presented above allows only for *monadic communication*, i.e., the exchange of tuples of values with length 1. We will find it convenient to use HO with *polyadic communication*, i.e., the exchange of tuples of values $\widetilde{V}$, with length $k \geq 1$. In HO, polyadicity appears in session synchronizations and applications, but not in synchronizations on shared names. This entails having the following reduction rules:

$$\begin{aligned}
(\lambda\widetilde{x}.\,P)\,\widetilde{u} &\longrightarrow P\{\widetilde{u}/\widetilde{x}\}\\
s!\langle\widetilde{V}\rangle.P \mid \overline{s}?(\widetilde{x}).Q &\longrightarrow P \mid Q\{\widetilde{V}/\widetilde{x}\}
\end{aligned}$$

where the simultaneous substitutions $P\{\widetilde{u}/\widetilde{x}\}$ and $P\{\widetilde{V}/\widetilde{x}\}$ are as expected. This polyadic HO can be readily encoded into (monadic) HO [20]; for this reason, by a slight abuse of notation we will often write HO when we actually mean "polyadic HO".

2.2 Session Types for HO

We give essential definitions and properties for the session type system for HO, following [20].

▶ **Definition 2.3** (Session Types for HO [20]). *Let us write $\diamond$ to denote the process type. The syntax of types for* HO *is defined as follows:*

$$\begin{array}{rcl}
U & ::= & C \to \diamond \;\mid\; C \multimap \diamond\\
C & ::= & S \;\mid\; \langle U\rangle\\
S & ::= & \mathtt{end} \;\mid\; !\langle U\rangle;S \;\mid\; ?(U);S \;\mid\; \oplus\{l_i : S_i\}_{i\in I} \;\mid\; \&\{l_i : S_i\}_{i\in I} \;\mid\; \mu\mathtt{t}.S \;\mid\; \mathtt{t}
\end{array}$$

Value types U include $C \to \diamond$ and $C \multimap \diamond$, which denote *shared* and *linear* higher-order types, respectively. Shared channel types are denoted $\langle S\rangle$ and $\langle U\rangle$. Session types, denoted by S, follow the standard binary session type syntax [12]. Type `end` is the termination type. The *output type* $!\langle U\rangle;S$ first sends a value of type U and then follows the type described by S. Dually, $?(U);S$ denotes an *input type*. The *branching type* $\&\{l_i : S_i\}_{i\in I}$ and the *selection type* $\oplus\{l_i : S_i\}_{i\in I}$ are used to type the branching and selection constructs that define the labeled choice. We assume the *recursive type* $\mu\mathtt{t}.S$ is guarded, i.e., type $\mu\mathtt{t}.\mathtt{t}$ is not allowed.

In session types theories *duality* is a key notion: implementations derived from dual session types will respect their protocols at run-time, avoiding communication errors. Intuitively, duality is obtained by exchanging ! by ? (and vice versa) and $\oplus$ by & (and vice versa), including the fixed point construction. We write S dual T if session types S and T are dual according to this intuition; the formal definition is coinductive, and given in [20].

We consider shared, linear, and session *environments*, denoted Γ, Λ, and Δ, resp.:

$$\begin{array}{rcl}
\Gamma & ::= & \emptyset \;\mid\; \Gamma, x : C \to \diamond \;\mid\; \Gamma, u : \langle U\rangle \qquad \Lambda \;::=\; \emptyset \;\mid\; \Lambda, x{:}C \multimap \diamond\\
\Delta & ::= & \emptyset \;\mid\; \Delta, u{:}S
\end{array}$$

$$\text{(Prom)}\ \frac{\Gamma;\emptyset;\emptyset \vdash V \triangleright C \multimap \diamond}{\Gamma;\emptyset;\emptyset \vdash V \triangleright C \to \diamond} \qquad \text{(EProm)}\ \frac{\Gamma;\Lambda, x : C \multimap \diamond;\Delta \vdash P \triangleright \diamond}{\Gamma, x : C \to \diamond;\Lambda;\Delta \vdash P \triangleright \diamond} \qquad \text{(Abs)}\ \frac{\Gamma;\Lambda;\Delta_1 \vdash P \triangleright \diamond \quad \Gamma;\emptyset;\Delta_2 \vdash x \triangleright C}{\Gamma\backslash x;\Lambda;\Delta_1\backslash\Delta_2 \vdash \lambda x.\, P \triangleright C \multimap \diamond}$$

$$\text{(App)}\ \frac{\Gamma;\Lambda;\Delta_1 \vdash V \triangleright C \rightsquigarrow \diamond \quad \rightsquigarrow \in \{\multimap, \to\} \quad \Gamma;\emptyset;\Delta_2 \vdash u \triangleright C}{\Gamma;\Lambda;\Delta_1,\Delta_2 \vdash V\, u \triangleright \diamond}$$

$$\text{(Send)}\ \frac{u : S \in \Delta_1,\Delta_2 \quad \Gamma;\Lambda_1;\Delta_1 \vdash P \triangleright \diamond \quad \Gamma;\Lambda_2;\Delta_2 \vdash V \triangleright U}{\Gamma;\Lambda_1,\Lambda_2;((\Delta_1,\Delta_2) \setminus u : S), u :!\langle U\rangle; S \vdash u!\langle V\rangle.P \triangleright \diamond}$$

$$\text{(Rcv)}\ \frac{\Gamma;\Lambda_1;\Delta, u : S \vdash P \triangleright \diamond \quad \Gamma;\Lambda_2;\emptyset \vdash x \triangleright U}{\Gamma\backslash x;\Lambda_1\backslash\Lambda_2;\Delta, u :?(U); S \vdash u?(x).P \triangleright \diamond}$$

$$\text{(Req)}\ \frac{\Gamma;\emptyset;\emptyset \vdash u \triangleright \langle U\rangle \quad \Gamma;\Lambda;\Delta_1 \vdash P \triangleright \diamond \quad \Gamma;\emptyset;\Delta_2 \vdash V \triangleright U}{\Gamma;\Lambda;\Delta_1,\Delta_2 \vdash u!\langle V\rangle.P \triangleright \diamond} \qquad \text{(Acc)}\ \frac{\Gamma;\emptyset;\emptyset \vdash u \triangleright \langle U\rangle \quad \Gamma;\Lambda_1;\Delta \vdash P \triangleright \diamond \quad \Gamma;\Lambda_2;\emptyset \vdash x \triangleright U}{\Gamma\backslash x;\Lambda_1\backslash\Lambda_2;\Delta \vdash u?(x).P \triangleright \diamond}$$

Figure 1 Selected Typing Rules for HO. See [20] for a full account.

Γ maps variables and shared names to value types; Λ maps variables to linear higher-order types. Δ maps session names to session types. While Γ admits weakening, contraction, and exchange principles, both Λ and Δ are only subject to exchange. The domains of Γ, Λ, and Δ are assumed pairwise distinct. $\Delta_1 \cdot \Delta_2$ is the disjoint union of Δ_1 and Δ_2.

We write $\Gamma\backslash x$ to denote $\Gamma\backslash\{x : C\}$, i.e., the environment obtained from Γ by removing the assignment $x : C \to \diamond$, for some C. Notations $\Delta\backslash u$ and $\Gamma\backslash\widetilde{x}$ will have expected readings. With a slight abuse of notation, given a tuple of variables $\widetilde{x}$, we sometimes write $(\Gamma, \Delta)(\widetilde{x})$ to denote the tuple of types assigned to variables in $\widetilde{x}$.

The typing judgements for values V and processes P are denoted

$$\Gamma;\Lambda;\Delta \vdash V \triangleright U \qquad \text{and} \qquad \Gamma;\Lambda;\Delta \vdash P \triangleright \diamond$$

Fig. 1 shows selected typing rules; see [20] for a full account. The shared type $C \to \diamond$ is derived using Rule (Prom) only if the value has a linear type with an empty linear environment. Rule (EProm) allows us to freely use a shared type variable as linear. Abstraction values are typed with Rule (Abs). Application typing is governed by Rule (App): the type C of an application name u must match the type of the application variable x ($C \multimap \diamond$ or $C \to \diamond$). In Rule (Send), the type U of value V should appear as a prefix in the session type $!\langle U\rangle; S$ of u. Rule (Rcv) is its dual. Rules (Req) and (Acc) type interaction along shared names; the type of the sent/received object V (i.e., U) should match the type of the subject s ($\langle U\rangle$).

To state type soundness, we require two auxiliary definitions on session environments. First, a session environment Δ is *balanced* (written $\mathsf{balanced}(\Delta)$) if whenever $s : S_1, \overline{s} : S_2 \in \Delta$ then S_1 dual S_2. Second, we define the reduction relation $\longrightarrow$ on session environments as:

$$\begin{aligned} \Delta, s :!\langle U\rangle; S_1, \overline{s} :?(U); S_2 &\longrightarrow \Delta, s : S_1, \overline{s} : S_2 \\ \Delta, s : \oplus\{l_i : S_i\}_{i\in I}, \overline{s} : \&\{l_i : S'_i\}_{i\in I} &\longrightarrow \Delta, s : S_k, \overline{s} : S'_k\ (k \in I) \end{aligned}$$

▶ **Theorem 2.4** (Type Soundness [20]). Suppose $\Gamma;\emptyset;\Delta \vdash P \triangleright \diamond$ with balanced(Δ). Then $P \longrightarrow P'$ implies $\Gamma;\emptyset;\Delta' \vdash P' \triangleright \diamond$ and $\Delta = \Delta'$ or $\Delta \longrightarrow \Delta'$ with balanced(Δ').

▶ Remark 2.5 (Typed Polyadic Communication). When using processes with polyadic communication (cf. Rem. 2.2), we shall assume the extension of the type system defined in [20].

▶ **Notation 1** (Type Annotations). *We shall often annotate bound names and variables with their respective type. We will write, e.g.,* $(\nu\, s : S)P$ *to denote that the type of* s *in* P *is* S*. Similarly for values: we shall write* $\lambda u : C.\, P$*. Also, letting* $\rightsquigarrow \in \{\multimap, \to\}$*, we may write* $\lambda u : C^{\rightsquigarrow}.\, P$ *to denote that the value is linear (if* $\rightsquigarrow = \multimap$*) or shared (if* $\rightsquigarrow = \to$*). That is, we write* $\lambda u : C^{\rightsquigarrow}.\, P$ *if* $\Gamma;\Lambda;\Delta \vdash \lambda u.\, P \triangleright C \rightsquigarrow \diamond$*, for some* Γ*,* Λ*, and* Δ*.*

Having introduced the core session process language HO, we now move to detail its type-preserving decomposition into minimal session types.

3 Decomposing Session-Typed Processes

3.1 Key Ideas

Our goal is to transform an HO process P, typable with the session types in Def. 2.3, into another HO process, denoted $\mathcal{D}(P)$, typable using *minimal session types* (cf. Def. 3.1 below). By means of this transformation on processes, which we call a *decomposition*, the sequencing in session types for P is codified in $\mathcal{D}(P)$ by using additional actions. To ensure that this transformation on P is sound, we must also decompose its session types; our main result says that if P is well-typed under session types $S_1, \ldots, S_n$, then $\mathcal{D}(P)$ is typable using the minimal session types $\mathcal{G}(S_1), \ldots, \mathcal{G}(S_n)$, where $\mathcal{G}(\cdot)$ is a decomposition function that "slices" a session type (as in Def. 2.3) into a *list* of minimal session types (cf. Def. 3.2 below).

To define the decomposition $\mathcal{D}(P)$, in Def. 3.8 we rely on a *breakdown function* that translates P into a composition of *trios processes* (or simply *trios*). A trio is a process with exactly three nested prefixes. Roughly speaking, if P is a sequential process with k nested actions, then $\mathcal{D}(P)$ will contain k trios running in parallel: each trio in $\mathcal{D}(P)$ will enact exactly one prefix from P; the breakdown function must be carefully designed to ensure that trios trigger each other in such a way that $\mathcal{D}(P)$ preserves the prefix sequencing in P.

We borrow from Parrow [24] some useful terminology and notation on trios. The *context* of a trio is a tuple of variables $\widetilde{x}$, possibly empty, which makes variable bindings explicit. We use a reserved set of *propagator names* (or simply *propagators*), denoted with $c_k, c_{k+1}, \ldots$, to carry contexts and trigger the subsequent trio. A process with less than three sequential prefixes is called a *degenerate trio*. Also, a *leading trio* is the one that receives a context, performs an action, and triggers the next trio; a *control trio* only activates other trios.

The breakdown function works on both processes and values. The breakdown of process P is denoted by $\mathcal{B}^k_{\widetilde{x}}(P)$, where k is the index for the propagators c_k, and $\widetilde{x}$ is the context to be received by the previous trio. Similarly, the breakdown of a value V is denoted by $\mathcal{V}^k_{\widetilde{x}}(V)$.

We present the decomposition of well-typed HO processes (and its associated typability results) incrementally – this is useful to gradually illustrate our ideas and highlight the several ways in which our developments differ from Parrow's. In § 3.2, we consider a "core fragment" of HO, which contains output and input prefixes, application, restriction, parallel composition, and inaction. Hence, this fragment does not have labeled choice and recursion, nor recursive types. In § 3.3 we shall extend the decomposition functions with selection and branching; an extension that supports names with recursive types is presented in § 3.4.

3.2 The Core Fragment

We present our approach for a core fragment of HO. We start introducing some preliminary definitions, including the definition of breakdown function. Then we give our main result: Thm. 3.11 (Page 13) asserts that if process P is well-typed with standard session types, then $\mathcal{D}(P)$ is well-typed with minimal session types. This theorem relies crucially on Thm. 3.10 (Page 13), which specifies the way in which the breakdown function preserves typability.

3.2.1 Preliminaries

We start by introducing minimal session types as a fragment of Def. 2.3:

▶ **Definition 3.1** (Minimal Session Types). *The syntax of* minimal session types *for* HO *is defined as follows:*

$$\begin{array}{rcl} U & ::= & \widetilde{C} \to \diamond \;\mid\; \widetilde{C} \multimap \diamond \\ C & ::= & M \;\mid\; \langle U \rangle \\ M & ::= & \mathtt{end} \;\mid\; !\langle \widetilde{U} \rangle; \mathtt{end} \;\mid\; ?(\widetilde{U}); \mathtt{end} \end{array}$$

Clearly, this minimal type structure induces a reduced set of typable HO processes. We shall implicitly assume a type system for HO based on these minimal session types by considering the expected specializations of the notions, typing rules, and results summarized in § 2.2.

We now define how to "slice" a session type into a *list* of minimal session types.

▶ **Definition 3.2** (Decomposing Session Types). *Let S be a session type, U be a higher-order type, C be a name type, and $\langle U \rangle$ be a shared type, all as in Def. 2.3. The* type decomposition function $\mathcal{G}(\cdot)$ *is defined as:*

$$\mathcal{G}(!\langle U \rangle; S) = \begin{cases} !\langle \mathcal{G}(U) \rangle; \mathtt{end} & \text{if } S = \mathtt{end} \\ !\langle \mathcal{G}(U) \rangle; \mathtt{end}\,, \mathcal{G}(S) & \text{otherwise} \end{cases}$$

$$\mathcal{G}(?(U); S) = \begin{cases} ?(\mathcal{G}(U)); \mathtt{end} & \text{if } S = \mathtt{end} \\ ?(\mathcal{G}(U)); \mathtt{end}\,, \mathcal{G}(S) & \text{otherwise} \end{cases}$$

$$\begin{array}{rcl} \mathcal{G}(\mathtt{end}) & = & \mathtt{end} \\ \mathcal{G}(C \multimap \diamond) & = & \mathcal{G}(C) \multimap \diamond \\ \mathcal{G}(C \to \diamond) & = & \mathcal{G}(C) \to \diamond \\ \mathcal{G}(\langle U \rangle) & = & \langle \mathcal{G}(U) \rangle \\ \mathcal{G}(S_1, \ldots, S_n) & = & \mathcal{G}(S_1), \ldots, \mathcal{G}(S_n) \end{array}$$

Thus, intuitively, if a session type S contains k input/output actions, the list $\mathcal{G}(S)$ will contain k minimal session types. We write $|\mathcal{G}(S)|$ to denote the length of $\mathcal{G}(S)$.

▶ **Example 3.3.** Let $S = ?(\mathsf{Int}); ?(\mathsf{Int}); !\langle \mathsf{Bool} \rangle; \mathtt{end}$ be the session type given in § 1. Then $\mathcal{G}(S)$ is the list of minimal session types given by $?(\mathsf{Int}); \mathtt{end}\,, ?(\mathsf{Int}); \mathtt{end}\,, !\langle \mathsf{Bool} \rangle; \mathtt{end}$. ⌟

The breakdown function $\mathcal{B}^k_{\tilde{x}}(\,\cdot\,)$ will operate on processes with *indexed* names (cf. Def. 3.6). Indexes are relevant for session names: a name s_i will execute the i-th action in session s. For this reason, to extend the decomposition function $\mathcal{G}(\cdot)$ to typing environments, we consider names u_i in Γ and Δ. To define the decomposition of environments, we rely on the following notation. Given a tuple of names $\widetilde{s} = s_1, \ldots, s_n$ and a tuple of (session) types $\widetilde{S} = S_1, \ldots, S_n$ of the same length, we write $\widetilde{s} : \widetilde{S}$ to denote a list of typing assignments $s_1 : S_1, \ldots, s_n : S_n$.

▶ **Definition 3.4** (Decomposition of Environments). *Let Γ, Λ, and Δ be typing environments. We define $\mathcal{G}(\Gamma)$, $\mathcal{G}(\Lambda)$, and $\mathcal{G}(\Delta)$ inductively as follows:*

$$\begin{aligned}\mathcal{G}(\Delta, u_i : S) &= \mathcal{G}(\Delta), (u_i, \ldots, u_{i+|\mathcal{G}(S)|-1}) : \mathcal{G}(S)\\ \mathcal{G}(\Gamma, u_i : \langle U \rangle) &= \mathcal{G}(\Gamma), u_i : \mathcal{G}(\langle U \rangle)\\ \mathcal{G}(\Gamma, x : U) &= \mathcal{G}(\Gamma), x : \mathcal{G}(U)\\ \mathcal{G}(\Lambda, x : U) &= \mathcal{G}(\Lambda), x : \mathcal{G}(U)\\ \mathcal{G}(\emptyset) &= \emptyset\end{aligned}$$

In order to determine the required number of propagators $(c_k, c_{k+1}, \ldots)$ required in the breakdown of processes and values, we mutually define their *degree*:

▶ **Definition 3.5** (Degree of a Process and Value). *Let P be an* HO *process. The* degree *of P, denoted $|P|$, is inductively defined as follows:*

$$|P| = \begin{cases} |V| + |Q| + 1 & \text{if } P = u_i!\langle V \rangle.Q \\ |Q| + 1 & \text{if } P = u_i!\langle y \rangle.Q \text{ or } P = u_i?(y).Q \\ |V| + 1 & \text{if } P = V\, u_i \\ |P'| & \text{if } P = (\nu\, s : S)P' \\ |Q| + |R| + 1 & \text{if } P = Q \mid R \\ 1 & \text{if } P = y\, u_i \text{ or } P = \mathbf{0} \end{cases}$$

The degree *of a value V, denoted $|V|$, is defined as follows:*

$$|V| = \begin{cases} |P| & \text{if } V = \lambda x : C^{\multimap}.\, P \\ 0 & \text{if } V = \lambda x : C^{\rightarrow}.\, P \text{ or } V = y \end{cases}$$

We define an auxiliary function that "initializes" the indices of a tuple of names.

▶ **Definition 3.6** (Name and Process Initialization). *Let $\widetilde{u} = (a, b, s, s', \ldots)$ be a finite tuple of names. We shall write* $\mathsf{init}(\widetilde{u})$ *to denote the tuple $(a_1, b_1, s_1, s'_1, \ldots)$. We will say that a process has been* initialized *if all of its names have some index.*

▶ Remark 3.7. Recall that we write '$c_k?()$' and '$\overline{c_k}!\langle\rangle$' to denote input and output prefixes in which the value communicated along c_k is not relevant. While '$c_k?()$' stands for '$c_k?(x)$', '$\overline{c_k}!\langle\rangle$' stands for '$\overline{c_k}!\langle \lambda x.\, \mathbf{0} \rangle$'. Their corresponding minimal types are $?(\mathtt{end} \to \diamond); \mathtt{end}$ and $!\langle \mathtt{end} \to \diamond \rangle; \mathtt{end}$, which are denoted by $?(\cdot); \mathtt{end}$ and $!\langle \cdot \rangle; \mathtt{end}$, respectively.

Recall that P is *closed* if $\mathtt{fv}(P) = \emptyset$. We now define the *decomposition* of a process.

▶ **Definition 3.8** (Decomposing Processes). *Let P be a closed* HO *process such that $\widetilde{u} = \mathtt{fn}(P)$. The* decomposition *of P, denoted $\mathcal{D}(P)$, is defined as:*

$$\mathcal{D}(P) = (\nu\, \widetilde{c})\big(\overline{c_k}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^k_{\epsilon}\big(P\sigma\big)\big)$$

where: $k > 0$; $\widetilde{c} = (c_k, \ldots, c_{k+|P|-1})$; $\sigma = \{^{\mathsf{init}(\tilde{u})}/_{\widetilde{u}}\}$; and the breakdown function $\mathcal{B}^k_{\widetilde{x}}(\,\cdot\,)$, *where $\widetilde{x}$ is a tuple of variables, is defined inductively in § 3.2.2.*

The bulk of the decomposition of a process is given by the breakdown function, detailed next.

3.2.2 The Breakdown Function

Given a context $\widetilde{x}$ and a $k > 0$, the breakdown function $\mathcal{B}^k_{\widetilde{x}}(\,\cdot\,)$ is defined on the structure of initialized processes, relying on the breakdown function on values $\mathcal{V}^k_{\widetilde{y}}(\,\cdot\,)$. The definition relies on type information; we describe each of its cases next.

Output. The decomposition of $u_i!\langle V\rangle.Q$ is the most interesting case: an output prefix sends a value V (i.e., an abstracted process) that has to be broken down as well. We then have:

$$\mathcal{B}^k_{\widetilde{x}}\big(u_i!\langle V\rangle.Q\big) = c_k?(\widetilde{x}).u_i!\big\langle\mathcal{V}^{k+1}_{\widetilde{y}}\big(V\sigma\big)\big\rangle.\overline{c_{k+l+1}}!\langle\widetilde{z}\rangle \mid \mathcal{B}^{k+l+1}_{\widetilde{z}}\big(Q\sigma\big)$$

Process $\mathcal{B}^k_{\widetilde{x}}\big(u_i!\langle V\rangle.Q\big)$ consists of a leading trio that mimics an output action in parallel with the breakdown of the continuation Q. The context $\widetilde{x}$ must include the free variables of V and Q, denoted $\widetilde{y}$ and $\widetilde{z}$, respectively. These tuples are not necessarily disjoint: variables with shared types can appear free in both V and Q. The output object V is then broken down with parameters $\widetilde{y}$ and $k+1$; the latter serves to consistently generate propagators for the trios in the breakdown of V, denoted $\mathcal{V}^{k+1}_{\widetilde{y}}\big(V\sigma\big)$ (see below for its definition). The substitution σ increments the index of session names; it is applied to both V and Q before they are broken down. We then distinguish two cases:

- If name u_i is linear (i.e., it has a session type) then its future occurrences are renamed into u_{i+1}, and $\sigma = \{u_{i+1}/u_i\}$;
- Otherwise, if u_i is not linear, then $\sigma = \{\}$.

Note that if u_i is linear then it appears either in V or Q and σ affects only one of them. The last prefix in the leading trio activates the breakdown of Q with its corresponding context $\widetilde{z}$. To avoid name conflicts with the propagators used in the breakdown of V, we use $\overline{c_{k+l+1}}$, with $l = |V|$ as a trigger for the continuation.

We remark that the same breakdown strategy is used when V stands for a variable y. Since by definition $|y| = 0$, $\mathcal{V}^k_{\widetilde{y}}\big(y\big) = y$, and $y\sigma = y$, we have:

$$\mathcal{B}^k_{\widetilde{x}}\big(u_i!\langle y\rangle.Q\big) = c_k?(\widetilde{x}).u_i!\langle y\rangle.\overline{c_{k+1}}!\langle\widetilde{z}\rangle \mid \mathcal{B}^{k+1}_{\widetilde{z}}\big(Q\sigma\big)$$

We may notice that variable y is not propagated further if it does not appear in Q.

Input. The breakdown of an input prefix is defined as follows:

$$\mathcal{B}^k_{\widetilde{x}}\big(u_i?(y).Q\big) = c_k?(\widetilde{x}).u_i?(y).\overline{c_{k+1}}!\langle\widetilde{x}'\rangle \mid \mathcal{B}^{k+1}_{\widetilde{x}'}\big(Q\sigma\big)$$

where $\widetilde{x}' = \mathtt{fv}(Q)$. A leading trio mimics the input action and possibly extends the context with the received variable y. The substitution σ is defined as in the output case.

Application. The breakdown of $V\,u_i$ is as follows:

$$\mathcal{B}^k_{\widetilde{x}}\big(V\,u_i\big) = c_k?(\widetilde{x}).\mathcal{V}^{k+1}_{\widetilde{x}}\big(V\big)\,\widetilde{m}$$

A degenerate trio receives a context $\widetilde{x}$ and then proceeds with the application. We break down V with $\widetilde{x}$ as a context since these variables need to be propagated to the abstracted process. We use $k+1$ as a parameter to avoid name conflicts. Name u_i is decomposed into a tuple $\widetilde{m}$ using type information: if $u_i : C$ then $\widetilde{m} = (u_i, \ldots, u_{i+|\mathcal{G}(C)|-1})$ and so the length of $\widetilde{m}$ is $|\mathcal{G}(C)|$; each name in $\widetilde{m}$ will perform exactly one action. When V is a variable y, we have:

$$\mathcal{B}^k_{\widetilde{x}}\big(y\,u_i\big) = c_k?(y).y\,\widetilde{m}$$

Notice that by construction $\widetilde{x} = y$.

Restriction. We define the breakdown of a restricted process as follows:

$$\mathcal{B}^k_{\widetilde{x}}\big((\nu\, s : C)P'\big) = (\nu\, \widetilde{s} : \mathcal{G}(C))\, \mathcal{B}^k_{\widetilde{x}}\big(P'\sigma\big)$$

By construction, $\widetilde{x} = \mathtt{fv}(P')$. Similarly as in the decomposition of u_i into $\widetilde{m}$ discussed above, we use the type C of s to obtain the tuple $\widetilde{s}$ of length $|\mathcal{G}(C)|$. We initialize the index of s in P' by applying the substitution σ. This substitution depends on C: if it is a shared type then $\sigma = \{s_1/s\}$; otherwise, if C is a session type, then $\sigma = \{s_1\overline{s_1}/s\overline{s}\}$.

Composition. The breakdown of a process $Q \mid R$ is as follows:

$$\mathcal{B}^k_{\widetilde{x}}\big(Q \mid R\big) = c_k?(\widetilde{x}).\overline{c_{k+1}}!\langle\widetilde{y}\rangle.\overline{c_{k+l+1}}!\langle\widetilde{z}\rangle \mid \mathcal{B}^{k+1}_{\widetilde{y}}\big(Q\big) \mid \mathcal{B}^{k+l+1}_{\widetilde{z}}\big(R\big)$$

A control trio triggers the breakdowns of Q and R; it does not mimic any action of the source process. The tuple $\widetilde{y} \subseteq \widetilde{x}$ (resp. $\widetilde{z} \subseteq \widetilde{x}$) collects the free variables in Q (resp. R). To avoid name conflicts, the trigger for the breakdown of R is $\overline{c_{k+l+1}}$, with $l = |Q|$.

Inaction. To breakdown $\mathbf{0}$, we define a degenerate trio with only one input prefix that receives a context that by construction will always be empty, i.e., $\widetilde{x} = \epsilon$:

$$\mathcal{B}^k_{\widetilde{x}}\big(\mathbf{0}\big) = c_k?().\mathbf{0}$$

Value. In defining the breakdown function for values we distinguish two main cases:

- If $V = \lambda y : C^{\rightsquigarrow}.\, P$, where $\rightsquigarrow \in \{\multimap, \to\}$, then we have:

$$\mathcal{V}^k_{\widetilde{x}}\big(\lambda y : C^{\rightsquigarrow}.\, P\big) = \lambda\widetilde{y} : \mathcal{G}(C)^{\rightsquigarrow}.\, (\nu\, \widetilde{c})\big(\overline{c_k}!\langle\widetilde{x}\rangle \mid \mathcal{B}^k_{\widetilde{x}}\big(P\{y_1/y\}\big)\big)$$

  We use type C to decompose y into the tuple $\widetilde{y}$. We abstract over $\widetilde{y}$; the body of the abstraction is the composition of a control trio and the breakdown of P, with name index initialized with the substitution $\{y_1/y\}$. If $\rightsquigarrow = \to$ then we restrict the propagators $\widetilde{c} = (c_k, \ldots, c_{k+|P|-1})$: this enables us to type the value in a shared environment. When $\rightsquigarrow = \multimap$ we do not have to restrict the propagators, and $\widetilde{c} = \epsilon$.
- If $V = y$, then the breakdown function is the identity: $\mathcal{V}^k_{\widetilde{x}}\big(y\big) = y$.

Tab. 1 summarizes the definition of the breakdown, spelling out the side conditions involved. We illustrate it by means of an example:

▶ **Example 3.9** (Breaking Down Name-Passing)**.** Consider the following process P, in which a channel m is passed, through which a Boolean value is sent back:

$$P = (\nu\, u)(u!\langle m\rangle.\overline{m}?(b).\mathbf{0} \mid \overline{u}?(x).x!\langle\mathsf{true}\rangle.\mathbf{0})$$

P is not an HO process as it features name-passing. We then use the encoding described in Exam. 2.1 to construct its encoding into HO. We thus obtain $[\![P]\!] = (\nu\, u)(Q \mid R)$, where

$$Q = u!\langle V\rangle.\overline{m}?(y).(\nu\, s)(y\, s \mid \overline{s}!\langle\lambda b.\, \mathbf{0}\rangle.\mathbf{0}) \qquad V = \lambda z.\, z?(x).(x\, m)$$
$$R = \overline{u}?(y).(\nu\, s)(y\, s \mid \overline{s}!\langle W\rangle.\mathbf{0}) \qquad W = \lambda x.\, x!\langle W'\rangle.\mathbf{0} \text{ with } W' = \lambda z.\, z?(x).(x\, \mathsf{true})$$

By Exam. 2.1, we know that $[\![\cdot]\!]$ requires exactly four reduction steps to mimic a name-passing synchronization. We show here part of the reduction chain of $[\![P]\!]$:

$$[\![P]\!] \longrightarrow^4 [\![\overline{m}?(b).\mathbf{0} \mid m!\langle\mathsf{true}\rangle.\mathbf{0}]\!] \longrightarrow^4 \mathbf{0} \tag{2}$$

We will now investigate the decomposition of $[\![P]\!]$ and its reduction chain. First, we use Def. 3.5 to compute $|V| = |W'| = 2$, and so $|W| = 4$. Then $|Q| = |y\, s \mid \overline{s}!\langle\lambda b.\, \mathbf{0}\rangle.\mathbf{0}| + |V| + 2 = 9$, and similarly, $|R| = 9$. Therefore, $|[\![P]\!]| = 19$. Following Def. 3.8, we see that $\sigma = \{m_1\overline{m_1}/m\overline{m}\}$, which we silently apply. Using $k = 1$, we then have the decomposition shown in Tab. 2.

Table 1 The breakdown function for processes and values (core fragment).

P	$\mathcal{B}^k_{\widetilde{x}}(P)$	
$u_i!\langle V\rangle.Q$	$c_k?(\widetilde{x}).u_i!\langle\mathcal{V}^{k+1}_{\widetilde{y}}(V\sigma)\rangle.\overline{c_{k+l+1}}!\langle\widetilde{z}\rangle \mid \mathcal{B}^{k+l+1}_{\widetilde{z}}(Q\sigma)$	$\widetilde{y} = \mathtt{fv}(V),\ \widetilde{z} = \mathtt{fv}(Q)$ $l = \lvert V\rvert$ $\sigma = \begin{cases}\{u_{i+1}/u_i\} & \text{if } u_i : S\\ \{\} & \text{otherwise}\end{cases}$
$u_i?(y).Q$	$c_k?(\widetilde{x}).u_i?(y).\overline{c_{k+1}}!\langle\widetilde{x}'\rangle \mid \mathcal{B}^{k+1}_{\widetilde{x}'}(Q\sigma)$	$\widetilde{x}' = \mathtt{fv}(Q)$ $\sigma = \begin{cases}\{u_{i+1}/u_i\} & \text{if } u_i : S\\ \{\} & \text{otherwise}\end{cases}$
$V\,u_i$	$c_k?(\widetilde{x}).\mathcal{V}^{k+1}_{\widetilde{x}}(V)\,\widetilde{m}$	$u_i : C$ $\widetilde{x} = \mathtt{fv}(V)$ $\widetilde{m} = (u_i, \ldots, u_{i+\lvert\mathcal{G}(C)\rvert-1})$
$(\nu\, s : C)P'$	$(\nu\,\widetilde{s} : \mathcal{G}(C))\,\mathcal{B}^k_{\widetilde{x}}(P'\sigma)$	$\widetilde{x} = \mathtt{fv}(P')$ $\widetilde{s} = (s_1, \ldots, s_{\lvert\mathcal{G}(C)\rvert})$ $\sigma = \begin{cases}\{s_1\overline{s_1}/s\overline{s}\} & \text{if } C = S\\ \{s_1/s\} & \text{if } C = \langle U\rangle\end{cases}$
$Q \mid R$	$c_k?(\widetilde{x}).\overline{c_{k+1}}!\langle\widetilde{y}\rangle.\overline{c_{k+l+1}}!\langle\widetilde{z}\rangle \mid \mathcal{B}^{k+1}_{\widetilde{y}}(Q) \mid \mathcal{B}^{k+l+1}_{\widetilde{z}}(R)$	$\widetilde{y} = \mathtt{fv}(Q)$ $\widetilde{z} = \mathtt{fv}(R)$ $l = \lvert Q\rvert$
$\mathbf{0}$	$c_k?().\mathbf{0}$	
V	$\mathcal{V}^k_{\widetilde{x}}(V)$	
y	y	
$\lambda u : C^{\rightsquigarrow}.\,P$	$\lambda\widetilde{y} : \mathcal{G}(C)^{\rightsquigarrow}.\,(\nu\,\widetilde{c})\big(\overline{c_k}!\langle\widetilde{x}\rangle \mid \mathcal{B}^k_{\widetilde{x}}(P\{y_1/y\})\big)$ $\widetilde{c} = \begin{cases}\epsilon & \text{if } \rightsquigarrow = \multimap\\ (c_k, \ldots, c_{k+\lvert P\rvert-1}) & \text{if } \rightsquigarrow = \to\end{cases}$	$\widetilde{x} = \mathtt{fv}(V)$ $\widetilde{y} = (y_1, \ldots, y_{\lvert\mathcal{G}(C)\rvert})$

Tab. 2 we have omitted substitutions that have no effect and trailing **0**s. The first interesting process appears after synchronizations on c_1, c_2, and c_{11}. At that point, the process will be ready to mimic the first action that is performed by $[\![P]\!]$, i.e., u_1 will send $\mathcal{V}^3_\epsilon(V)$, the breakdown of V. Next, c_{12}, c_{13}, and c_{14} will synchronize, and $\mathcal{V}^3_\epsilon(V)$ is passed further along, until s_1 is ready to be applied to it in the breakdown of R. At this point, we know that $[\![P]\!] \longrightarrow^7 (\nu\,\widetilde{c})P'$, where $\widetilde{c} = (c_3, \ldots, c_{10}, c_{15}, \ldots, c_{19})$, and

$$\begin{aligned} P' = {} & \overline{c_5}!\langle\rangle.\mathbf{0} \mid c_5?().\overline{m_1}?(y).\overline{c_6}!\langle y\rangle.\mathbf{0} \\ & \mid (\nu\, s_1)(c_6?(y).\overline{c_7}!\langle y\rangle.\overline{c_8}!\langle\rangle.\mathbf{0} \mid c_7?(y).y\,s_1 \mid c_8?().\overline{s_1}!\langle\mathcal{V}^9_\epsilon(\lambda b.\,\mathbf{0})\rangle.\overline{c_{10}}!\langle\rangle.\mathbf{0} \mid c_{10}?().\mathbf{0}) \\ & \mid (\nu\, s_1)\big(\mathcal{V}^3_\epsilon(V)\,s_1 \mid \overline{s_1}!\langle\mathcal{V}^{15}_\epsilon(W)\rangle.\overline{c_{19}}!\langle\rangle.\mathbf{0} \mid c_{19}?().\mathbf{0}\big) \end{aligned}$$

After s_1 is applied, the trio guarded by c_3 will be activated, where z_1 has been substituted by s_1. Then $\overline{s_1}$ and s_1 will synchronize, and the breakdown of W is passed along. Then c_4 and c_{19} synchronize, and now m_1 is ready to be applied to $\mathcal{V}^{15}_\epsilon(W)$, which was the input

Table 2 The process decomposition discussed in Exam. 3.9.

$$
\begin{aligned}
\mathcal{D}([\![P]\!]) &= (\nu\, c_1,\ldots,c_{19})\Big(\overline{c_1}!\langle\rangle \mid (\nu\, u_1)\big(c_1?().\overline{c_2}!\langle\rangle.\overline{c_{11}}!\langle\rangle \mid \mathcal{B}^2_\epsilon(Q) \mid \mathcal{B}^{11}_\epsilon(R)\big)\Big)\\
\mathcal{B}^2_\epsilon(Q) &= c_2?().u_1!\langle\mathcal{V}^3_\epsilon(V)\rangle.\overline{c_5}!\langle\rangle \mid c_5?().\overline{m_1}?(y).\overline{c_6}!\langle y\rangle \mid\\
&\quad (\nu\, s_1)(c_6?(y).\overline{c_7}!\langle y\rangle.\overline{c_8}!\langle\rangle \mid c_7?(y).(y\, s_1) \mid c_8?().\overline{s_1}!\langle\mathcal{V}^9_\epsilon(\lambda b.\,\mathbf{0})\rangle.\overline{c_{10}}!\langle\rangle \mid c_{10}?())\\
\mathcal{B}^{11}_\epsilon(R) &= c_{11}?().\overline{u_1}?(y).\overline{c_{12}}!\langle y\rangle \mid\\
&\quad (\nu\, s_1)\big(c_{12}?(y).\overline{c_{13}}!\langle y\rangle.\overline{c_{14}}!\langle\rangle \mid c_{13}?(y).(y\, s_1) \mid c_{14}?().\overline{s_1}!\langle\mathcal{V}^{15}_\epsilon(W)\rangle.\overline{c_{19}}!\langle\rangle \mid c_{19}?()\big)\\
\mathcal{V}^3_\epsilon(V) &= \lambda z_1.\,(\overline{c_3}!\langle\rangle \mid c_3?().z_1?(x).\overline{c_4}!\langle x\rangle \mid c_4?(x).(x\, m_1))\\
\mathcal{V}^9_\epsilon(\lambda b.\,\mathbf{0}) &= \lambda b_1.\,(\overline{c_9}!\langle\rangle \mid c_9?())\\
\mathcal{V}^{15}_\epsilon(W) &= \lambda x_1.\,(\overline{c_{15}}!\langle\rangle \mid c_{15}?().x_1!\langle\mathcal{V}^{16}_\epsilon(W')\rangle.\overline{c_{18}}!\langle\rangle \mid c_{18}?())\\
\mathcal{V}^{16}_\epsilon(W') &= \lambda z_1.\,(\overline{c_{16}}!\langle\rangle \mid c_{16}?().z_1?(x).\overline{c_{17}}!\langle x\rangle \mid c_{17}?(x).(x\,\mathsf{true}))
\end{aligned}
$$

for c_4 in the breakdown of V. After this application, c_5 and c_{15} can synchronize with their duals, and we know that $(\nu\,\widetilde{c})P' \longrightarrow^8 (\nu\,\widetilde{c}')P''$, where $\widetilde{c}' = (c_6,\ldots,c_{10},c_{16},c_{17},c_{18})$, and

$$
\begin{aligned}
P'' = {} & \overline{m_1}?(y).\overline{c_6}!\langle y\rangle.\mathbf{0} \mid m_1!\langle\mathcal{V}^{15}_\epsilon(W')\rangle.\overline{c_{17}}!\langle\rangle.\mathbf{0} \mid c_{17}?().\mathbf{0}\\
& \mid (\nu\, s_1)(c_6?(y).\overline{c_7}!\langle y\rangle.\overline{c_8}!\langle\rangle.\mathbf{0} \mid c_7?(y).y\, s_1 \mid c_8?().\overline{s_1}!\langle\mathcal{V}^9_\epsilon(\lambda b.\,\mathbf{0})\rangle.\overline{c_{10}}!\langle\rangle.\mathbf{0} \mid c_{10}?().\mathbf{0})
\end{aligned}
$$

Remarkably, P'' is standing by to mimic the encoded exchange of value true. Indeed, the decomposition of the four-step reduced process in (2) will reduce in three steps to a process that is equal (up to $\equiv_\alpha$) to the process we obtained here. This strongly suggests a tight operational correspondence between a process and its decomposition. ⌟

We may now state our technical results:

▶ **Theorem 3.10** (Typability of Breakdown). *Let P be an initialized process and V be a value.*

1. *If* $\Gamma;\Lambda;\Delta \vdash P \triangleright \diamond$ *then*

$$\mathcal{G}(\Gamma_1);\emptyset;\mathcal{G}(\Delta),\Theta \vdash \mathcal{B}^k_{\widetilde{x}}(P) \triangleright \diamond \quad (k>0)$$

where: $\widetilde{x} = \mathtt{fv}(P)$; $\Gamma_1 = \Gamma \setminus \widetilde{x}$; *and* $\mathsf{balanced}(\Theta)$ *with* $dom(\Theta) = \{c_k, c_{k+1},\ldots,c_{k+|P|-1}\} \cup \{\overline{c_{k+1}},\ldots,\overline{c_{k+|P|-1}}\}$ *and* $\Theta(c_k) = ?(\widetilde{M});\mathtt{end}$, *where* $\widetilde{M} = (\mathcal{G}(\Gamma),\mathcal{G}(\Lambda))(\widetilde{x})$.

2. *If* $\Gamma;\Lambda;\Delta \vdash V \triangleright C \multimap \diamond$ *then*

$$\mathcal{G}(\Gamma);\mathcal{G}(\Lambda);\mathcal{G}(\Delta),\Theta \vdash \mathcal{V}^k_{\widetilde{x}}(V) \triangleright \mathcal{G}(C) \multimap \diamond \quad (k>0)$$

where: $\widetilde{x} = \mathtt{fv}(V)$; *and* $\mathsf{balanced}(\Theta)$ *with* $dom(\Theta) = \{c_k,\ldots,c_{k+|V|-1}\} \cup \{\overline{c_k},\ldots,\overline{c_{k+|V|-1}}\}$ *and* $\Theta(c_k) = ?(\widetilde{M});\mathtt{end}$ *and* $\Theta(\overline{c_k}) = !\langle\widetilde{M}\rangle;\mathtt{end}$, *where* $\widetilde{M} = (\mathcal{G}(\Gamma),\mathcal{G}(\Lambda))(\widetilde{x})$.

3. *If* $\Gamma;\emptyset;\emptyset \vdash V \triangleright C \to \diamond$ *then* $\mathcal{G}(\Gamma);\emptyset;\emptyset \vdash \mathcal{V}^k_{\widetilde{x}}(V) \triangleright \mathcal{G}(C) \to \diamond$, *where* $\widetilde{x} = \mathtt{fv}(V)$ *and* $k>0$.

Proof. By mutual induction on the structure of P and V. ◀

Using the above theorem, we can prove our main result:

▶ **Theorem 3.11** (Typability of the Decomposition). *Let P be a closed* HO *process with* $\widetilde{u} = \mathtt{fn}(P)$. *If* $\Gamma;\emptyset;\Delta \vdash P \triangleright \diamond$ *then* $\mathcal{G}(\Gamma\sigma);\emptyset;\mathcal{G}(\Delta\sigma) \vdash \mathcal{D}(P) \triangleright \diamond$, *where* $\sigma = \{^{\mathsf{init}(\widetilde{u})}/_{\widetilde{u}}\}$.

Proof. Direct from the definitions, using Thm. 3.10. ◀

3.3 Extensions (I): Select and Branching

We now show how to extend the decomposition to handle select and branch processes, which implement labeled (deterministic) choice in session protocols, as well as their corresponding session types. As we will see, in formalizing this extension we shall appeal to the expressive power of abstraction-passing. We start by extending the syntax of minimal session types:

▶ **Definition 3.12** (Minimal Session Types (with Labeled Choice)). *The syntax of* minimal session types *for* HO *is defined as follows:*

$$M \;::=\; \mathtt{end} \;\mid\; !\langle\widetilde{U}\rangle;\mathtt{end} \;\mid\; ?(\widetilde{U});\mathtt{end} \;\mid\; \oplus\{l_i : M_i\}_{i\in I} \;\mid\; \&\{l_i : M_i\}_{i\in I}$$

where U and C are defined as in Def. 3.1.

We may then extend Def. 3.2 to branch and select types as follows:

▶ **Definition 3.13** (Decomposing Session Types, Extended (I)). *The decomposition function on types as given in Def. 3.2 is extended as follows:*

$$\mathcal{G}(\&\{l_i : S_i\}_{i\in I}) = \&\{l_i : !\langle\mathcal{G}(S_i) \multimap \diamond\rangle;\mathtt{end}\}_{i\in I}$$
$$\mathcal{G}(\oplus\{l_i : S_i\}_{i\in I}) = \oplus\{l_i : ?(\mathcal{G}(\overline{S_i}) \multimap \diamond);\mathtt{end}\}_{i\in I}$$

The above definition for decomposed types already suggests our strategy to breakdown branching and selection processes: we will exploit abstraction-passing to exchange one abstraction per each branch of the labeled choice. This intuition will become clearer shortly.

We now extend the definition of the degree of a process/value (cf. Def. 3.5) to account for branch and select processes:

▶ **Definition 3.14** (Degree of a Process, Extended). *The* degree *of a process P, denoted $|P|$, is as given in Def. 3.5, extended as follows:*

$$|P| = \begin{cases} 1 & \text{if } P = u_i \triangleright \{l_j : P_j\}_{j\in I} \\ |P'| + 2 & \text{if } P = u_i \triangleleft l_j.P' \end{cases}$$

The definition of process decomposition (cf. Def. 3.8) does not require modifications; it relies on the extended definition of the breakdown function for processes $\mathcal{B}^k_{\widetilde{x}}(\,\cdot\,)$ that combines the definitions in Tab. 1 with those in Tab. 3 (see below). The breakdown of values $\mathcal{V}^k_{\widetilde{x}}(\,\cdot\,)$ is as before, and relies on the extended definition of $\mathcal{B}^k_{\widetilde{x}}(\,\cdot\,)$.

We now present and describe the breakdown of branching and selection processes:

Branching. The breakdown of a branching process $u_i \triangleright \{l_j : P_j\}_{j\in I}$ is as follows:

$$\mathcal{B}^k_{\widetilde{x}}\big(u_i \triangleright \{l_j : P_j\}_{j\in I}\big) = c_k?(\widetilde{x}).u_i \triangleright \{l_j : u_i!\big\langle N_{u,j}\big\rangle\}_{j\in I}$$
$$\text{where}\;\; N_{u,j} = \lambda\widetilde{y}^u_j : \mathcal{G}(S_j).\,(\nu\,\widetilde{c}_j)\big(\overline{c_{k+1}}!\langle\widetilde{x}\rangle \mid \mathcal{B}^{k+1}_{\widetilde{x}}\big(P_j\{y^u_1/u_i\}\big)\big)$$

The first prefix receives the context $\widetilde{x}$. The next two prefixes are along u_i: the first one mimics the branching action of P, whereas the second outputs an abstraction $N_{u,j}$. This output does not have a counterpart in P; it is meant to synchronize with an input in the breakdown of the corresponding selection process (see below). $N_{u,j}$ encapsulates the breakdown of subprocess P_j. It has the same structure as the breakdown of a value $\lambda y : C^{\to}.\,P$ in Tab. 1: it is a composition of a control trio and the breakdown of P_j; the generated propagators, denoted $\widetilde{c}_j$, are restricted. We use types to define $N_{u,j}$: we assume S_j is the session type of u_i in the j-th branch of P. We abstract over $\widetilde{y}^u_j = (y^u_1, \ldots, y^u_{|\mathcal{G}(S_j)|})$. We substitute u_i with y^u_1 in P_j before breaking it down: this way, u_i is decomposed and bound by abstraction.

Table 3 The breakdown function for processes (extension with selection and branching).

$\mathcal{B}^k_{\widetilde{x}}\big(u_i \triangleright \{l_j{:}P_j\}_{j\in I}\big)$	
$c_k?(\widetilde{x}).u_i \triangleright \{l_j : u_i!\langle N_{u,j}\rangle\}_{j\in I}$ where: $N_{u,j} = \lambda\widetilde{y}^u_j : \mathcal{G}(S_j).\,(\nu\,\widetilde{c}_j)\big(\overline{c_{k+1}}!\langle\widetilde{x}\rangle \mid \mathcal{B}^{k+1}_{\widetilde{x}}\big(P_j\{y^u_1/u_i\}\big)\big)$	$\widetilde{y}^u_j = (y^u_1,\ldots,y^u_{\lvert\mathcal{G}(S_j)\rvert})$ $\widetilde{c}_j = (c_{k+1},\ldots,c_{k+\lvert P_j\rvert})$
$\mathcal{B}^k_{\widetilde{x}}\big(u_i \triangleleft l_j.P'\big)$	
$c_k?(\widetilde{x}).\overline{c_{k+1}}!\big\langle M_j\big\rangle \mid$ $(\nu\,\widetilde{u} : \mathcal{G}(S_j))\big(c_{k+1}?(y).y\,\widetilde{\overline{u}} \mid \mathcal{B}^{k+2}_{\widetilde{x}}\big(P'\{u_{i+1}/u_i\}\big)\big)$ where: $M_j = \lambda\widetilde{y}.\,u_i \triangleleft l_j.u_i?(z).\overline{c_{k+2}}!\langle\widetilde{x}\rangle.z\,\widetilde{y}$	$\widetilde{y} = (y_1,\ldots,y_{\lvert\mathcal{G}(S_j)\rvert})$ $\widetilde{u} = (u_{i+1},\ldots,u_{i+\lvert\mathcal{G}(S_j)\rvert})$ $\widetilde{\overline{u}} = (\overline{u_{i+1}},\ldots,\overline{u_{i+\lvert\mathcal{G}(S_j)\rvert}})$

Selection. The breakdown of a selection process $u_i \triangleleft l_j.P'$ is as follows:

$$\mathcal{B}^k_{\widetilde{x}}\big(u_i \triangleleft l_j.P'\big) = c_k?(\widetilde{x}).\overline{c_{k+1}}!\big\langle M_j\big\rangle \mid (\nu\,\widetilde{u} : \mathcal{G}(S_j))(c_{k+1}?(y).y\,\widetilde{\overline{u}} \mid \mathcal{B}^{k+2}_{\widetilde{x}}\big(P'\{u_{i+1}/u_i\}\big))$$
$$\text{where } M_j = \lambda\widetilde{y}.\,u_i \triangleleft l_j.u_i?(z).\overline{c_{k+2}}!\langle\widetilde{x}\rangle.z\,\widetilde{y}$$

After receiving the context $\widetilde{x}$, the abstraction M_j is sent along $\overline{c_{k+1}}$, and is to be received by the second subprocess in the composition. This sequence of actions allows us to preserve the intended trio structure. We use S_j, the type of u_i in P', to construct a corresponding tuple $\widetilde{u}$, with type $\mathcal{G}(S_j)$. We apply the abstraction M_j, received along c_{k+1}, to $\widetilde{\overline{u}}$ (the duals of $\widetilde{u}$). At this point, the selection action in P can be mimicked, and so label l_j is chosen from the breakdown of a corresponding branching process. As discussed above, such a breakdown will send an abstraction $N_{u,j}$ with type $\overline{S_j} \multimap \diamond$, which encapsulates the breakdown of the chosen subprocess. Before running $N_{u,j}$ with names $\widetilde{\overline{u}}$, we trigger the breakdown of P' with an appropriate substitution.

Summing up, our strategy for breaking down labeled choices exploits higher-order concurrency to uniformly handle the fact that the subprocesses of a branching process have a different session type and degree. Interestingly, it follows the intuition that branching and selection correspond to a form of output and input actions involving labels, respectively.

▶ Remark 3.15. Theorems 3.10 and 3.11 hold also for the extension with selection and branching .

▶ **Example 3.16** (Breaking down Selection and Branching)**.** We illustrate the breaking down of selection and branching processes by considering a basic mathematical server Q that allows clients to add or subtract two integers. The server contains two branches: one sends an abstraction V_+ that implements integer addition, the other sends an abstraction V_- implementing subtraction. A client R selects the first option to add integers 16 and 26:

$$Q \triangleq u \triangleright \{\mathsf{add} : u!\langle V_+\rangle.\mathbf{0}, \mathsf{sub} : u!\langle V_-\rangle.\mathbf{0}\}$$
$$R \triangleq \overline{u} \triangleleft \mathsf{add}.\overline{u}?(x).x\,(16{,}26)$$

The composition $P \triangleq (\nu\,u)(Q \mid R)$ reduces in two steps to a process $V_+\,(16{,}26)$:

$$P \longrightarrow (\nu\,u)(u!\langle V_+\rangle.\mathbf{0} \mid \overline{u}?(x).x\,(16{,}26)) \longrightarrow V_+\,(16{,}26) \tag{3}$$

We will investigate the decomposition of P, and its reduction chain. First, by Def. 3.5 and Def. 3.14, we have: $|Q| = 1$, $|R| = 4$, and $|P| = 6$. Following the extension of Def. 3.8, using $k = 1$, and observing that $\sigma_1 = \{\}$, we obtain:

$$\mathcal{D}(P) = (\nu\,c_1\ldots c_6)\big(\overline{c_1}!\langle\rangle.\mathbf{0} \mid (\nu\,u_1)(c_1?().\overline{c_2}!\langle\rangle.\overline{c_3}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^2_\epsilon\big(Q\sigma_2\big) \mid \mathcal{B}^3_\epsilon\big(R\sigma_2\big))\big)$$

where $\sigma_2 = \{^{u_1\overline{u_1}}/_{u\overline{u}}\}$. The breakdown of Q is obtained by applying the first rule in Tab. 3:

$$\begin{aligned}\mathcal{B}^2_\epsilon\big(Q\sigma_2\big) = c_2?().u_1 \triangleright \{&\mathsf{add} : (\nu\, c_3c_4)u_1!\big\langle\lambda y_1.\,\overline{c_3}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^3_\epsilon\big(u_1!\langle V_+\rangle.\mathbf{0}\{^{y_1}/_{u_1}\}\big)\big\rangle.\mathbf{0},\\ &\mathsf{sub} : (\nu\, c_3c_4)u_1!\big\langle\lambda y_1.\,\overline{c_3}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^3_\epsilon\big(u_1!\langle V_-\rangle.\mathbf{0}\{^{y_1}/_{u_1}\}\big)\big\rangle.\mathbf{0}\}\end{aligned}$$

The breakdown of R is obtained by applying the second rule in Tab. 3:

$$\begin{aligned}\mathcal{B}^3_\epsilon\big(R\sigma_2\big) = c_3?().\overline{c_4}!\langle\lambda y_1.\,\overline{u_1} \triangleleft \mathsf{add}.\overline{u_1}?(z).\overline{c_5}!\langle\rangle.z\, y_1\rangle.\mathbf{0}\\ \mid (\nu\, u_2)(c_4?(y).y\,\overline{u_2} \mid \mathcal{B}^5_\epsilon\big(\overline{u_1}?(x).x\,(16{,}26)\{^{u_2}/_{\overline{u_1}}\}\big))\end{aligned}$$

We will now follow the chain of reductions of the process $\mathcal{D}(P)$. First, c_1, c_2, and c_3 will synchronize, after which c_4 will pass the abstraction. Let $\mathcal{D}(P) \longrightarrow^4 P'$, then we know:

$$\begin{aligned}P' = (\nu\, c_5c_6)(\nu\, u_1)\big(u_1 \triangleright \{&\mathsf{add} : (\nu\, c_3c_4)u_1!\big\langle\lambda y_1.\,\overline{c_3}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^3_\epsilon\big(y_1!\langle V_+\rangle.\mathbf{0}\big)\big\rangle.\mathbf{0},\\ &\mathsf{sub} : (\nu\, c_3c_4)u_1!\big\langle\lambda y_1.\,\overline{c_3}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^3_\epsilon\big(y_1!\langle V_-\rangle.\mathbf{0}\big)\big\rangle.\mathbf{0}\}\\ \mid (\nu\, u_2)(\lambda y_1.\,\overline{u_1} \triangleleft \mathsf{add}.&\overline{u_1}?(z).\overline{c_5}!\langle\rangle.z\, y_1\,\overline{u_2} \mid \mathcal{B}^5_\epsilon\big(u_2?(x).x\,(16{,}26)\big)\big)\big)\end{aligned}$$

In P', $\overline{u_2}$ will be applied to the abstraction with variable y_1. After that, the choice for the process labeled by add is made. Process P' will reduce further as $P' \longrightarrow^2 P'' \longrightarrow^2 P'''$, where:

$$\begin{aligned}P'' = (\nu\, c_5c_6)(\nu\, u_1)\big((\nu\, c_3c_4)u_1!\big\langle\lambda y_1.\,\overline{c_3}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^3_\epsilon\big(y_1!\langle V_+\rangle.\mathbf{0}\big)\big\rangle.\mathbf{0}\\ \mid (\nu\, u_2)(\overline{u_1}?(z).\overline{c_5}!\langle\rangle.z\,\overline{u_2} \mid \mathcal{B}^5_\epsilon\big(u_2?(x).x\,(16{,}26)\big)\big)\big)\\ P''' = (\nu\, c_3c_4c_5c_6)\big((\nu\, u_2)\overline{c_5}!\langle\rangle.\overline{c_3}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^3_\epsilon\big(u_2!\langle V_+\rangle.\mathbf{0}\big) \mid \mathcal{B}^5_\epsilon\big(u_2?(x).x\,(16{,}26)\big)\big)\end{aligned}$$

Interestingly, P''' strongly resembles a decomposition of the one-step reduced process in (3). This advocates the operational correspondence between a process and its decomposition. ⌟

3.4 Extensions (II): Recursion

We extend the decomposition to handle HO processes in which names can be typed with recursive session types $\mu\mathsf{t}.S$. We consider recursive types which are *simple* and *contractive*, i.e., in $\mu\mathsf{t}.S$, the body $S \neq \mathsf{t}$ does not contain recursive types. Unless stated otherwise, we shall handle *tail-recursive* session types such as, e.g., $S = \mu\mathsf{t}.?(\mathsf{Int});?(\mathsf{Bool});!\langle\mathsf{Bool}\rangle;\mathsf{t}$. Non-tail-recursive session types such as $\mu\mathsf{t}.?((\widetilde{T},\mathsf{t})\to\diamond);\mathtt{end}$, which is essential in the fully abstract encoding of HOπ into HO [19], can also be accommodated; see Rem. 3.29 below.

We start by extending minimal session types (Def. 3.1) with minimal recursive types:

▶ **Definition 3.17** (Minimal Recursive Session Types)**.** *The syntax of* minimal recursive session types *for* HO *is defined as follows:*

$$\begin{aligned}M &::= \gamma \mid !\langle\widetilde{U}\rangle;\gamma \mid ?(\widetilde{U});\gamma \mid \mu\mathsf{t}.M\\ \gamma &::= \mathtt{end} \mid \mathsf{t}\end{aligned}$$

Thus, types such as $\mu\mathsf{t}.!\langle U\rangle;\mathsf{t}$ and $\mu\mathsf{t}.?(U);\mathsf{t}$ are minimal recursive session types: in fact they are tail-recursive session types with exactly one session prefix. We extend Def. 3.2 as follows:

▶ **Definition 3.18** (Decomposing Session Types, Extended (II)). *Let $\mu\mathsf{t}.S$ be a recursive session type. The decomposition function given in Def. 3.2 is extended as:*

$$\mathcal{G}(\mathsf{t}) = \mathsf{t} \qquad \mathcal{G}(\mu\mathsf{t}.S) = \begin{cases} \mathcal{R}(S) & \text{if } \mu\mathsf{t}.S \text{ is tail-recursive} \\ \mu\mathsf{t}.\mathcal{G}(S) & \text{otherwise} \end{cases}$$

$$\mathcal{R}(\mathsf{t}) = \epsilon \qquad \mathcal{R}(!\langle U\rangle;S) = \mu\mathsf{t}.!\langle\mathcal{G}(U)\rangle;\mathsf{t},\mathcal{R}(S)$$
$$\mathcal{R}(?(U);S) = \mu\mathsf{t}.?(\mathcal{G}(U));\mathsf{t},\mathcal{R}(S)$$

We shall also use the function $\mathcal{R}^\star(\cdot)$, which is defined as follows:

$$\mathcal{R}^\star(?(U);S) = \mathcal{R}^\star(S) \qquad \mathcal{R}^\star(!\langle U\rangle;S) = \mathcal{R}^\star(S) \qquad \mathcal{R}^\star(\mu\mathsf{t}.S) = \mathcal{R}(S)$$

Hence, $\mathcal{G}(\mu\mathsf{t}.S)$ is a list of minimal recursive session types, obtained using the auxiliary function $\mathcal{R}(\cdot)$ on S: if S has k prefixes then the list $\mathcal{G}(\mu\mathsf{t}.S)$ will contain k minimal recursive session types. The auxiliary function $\mathcal{R}^\star(\cdot)$ decomposes *guarded* recursive session types: it skips session prefixes until a type of form $\mu\mathsf{t}.S$ is encountered; when that occurs, the recursive type is decomposed using $\mathcal{R}(\cdot)$. We illustrate Def. 3.18 with two examples:

▶ **Example 3.19** (Decomposing a Recursive Type). Let $S = \mu\mathsf{t}.S'$ be a recursive session type, with $S' = ?(\mathsf{Int});?(\mathsf{Bool});!\langle\mathsf{Bool}\rangle;\mathsf{t}$. By Def. 3.18, since S is tail-recursive, $\mathcal{G}(S) = \mathcal{R}(S')$. Further, $\mathcal{R}(S') = \mu\mathsf{t}.?(\mathcal{G}(\mathsf{Int}));\mathsf{t},\mathcal{R}(?(\mathsf{Bool});!\langle\mathsf{Bool}\rangle;\mathsf{t})$. By definition of $\mathcal{R}(\cdot)$, we obtain $\mathcal{G}(S) = \mu\mathsf{t}.?(\mathsf{Int});\mathsf{t},\mu\mathsf{t}.?(\mathsf{Bool});\mathsf{t},\mu\mathsf{t}.!\langle\mathsf{Bool}\rangle;\mathsf{t},\mathcal{R}(t)$ (using $\mathcal{G}(\mathsf{Int}) = \mathsf{Int}$ and $\mathcal{G}(\mathsf{Bool}) = \mathsf{Bool}$). Since $\mathcal{R}(\mathsf{t}) = \epsilon$, we obtain $\mathcal{G}(S) = \mu\mathsf{t}.?(\mathsf{Int});\mathsf{t},\mu\mathsf{t}.?(\mathsf{Bool});\mathsf{t},\mu\mathsf{t}.!\langle\mathsf{Bool}\rangle;\mathsf{t}$. ⌟

▶ **Example 3.20** (Decomposing an Unfolded Recursive Type). Let $T = ?(\mathsf{Bool});!\langle\mathsf{Bool}\rangle;S$ be a derived unfolding of S from Exam. 3.19. Then, by Def. 3.18, $\mathcal{R}^\star(T)$ is the list of minimal recursive types obtained as follows: first, $\mathcal{R}^\star(T) = \mathcal{R}^\star(!\langle\mathsf{Bool}\rangle;\mu\mathsf{t}.S')$ and after one more step, $\mathcal{R}^\star(!\langle\mathsf{Bool}\rangle;\mu\mathsf{t}.S') = \mathcal{R}^\star(\mu\mathsf{t}.S')$. Finally, we have $\mathcal{R}^\star(\mu\mathsf{t}.S') = \mathcal{R}(S')$. We get the same list of minimal types as in Exam. 3.19: $\mathcal{R}^\star(T) = \mu\mathsf{t}.?(\mathsf{Int});\mathsf{t},\mu\mathsf{t}.?(\mathsf{Bool});\mathsf{t},\mu\mathsf{t}.!\langle\mathsf{Bool}\rangle;\mathsf{t}$. ⌟

We now explain how to decompose processes whose names are typed with recursive types. In the core fragment, we decompose a name u into a sequence of names $\tilde{u} = (u_1,\ldots,u_n)$: each $u_i \in \tilde{u}$ is used exactly by one trio to perform exactly one action; the session associated to u_i ends after its single use, as prescribed by its minimal session type. The situation is different when names can have recursive types, for the names $\tilde{u}$ should be propagated in order to be used infinitely many times. As a simple example, consider the process

$$R = r?(x).r!\langle x\rangle.V\ r$$

where name r has type $S = \mu\mathsf{t}.?(\mathsf{Int});!\langle\mathsf{Int}\rangle;\mathsf{t}$ and the higher-order type of V is $S \to \diamond$. Processes of this form are key in the encoding of recursion given in [19]. A naive decomposition of R, using the approach we defined for processes without recursive types, would result into

$$\mathcal{B}^1_\epsilon\big(R\big) = c_1?().r_1?(x).\overline{c_2}!\langle x\rangle.\mathbf{0} \mid c_2?(x).r_2!\langle x\rangle.\overline{c_3}!\langle\rangle.\mathbf{0} \mid c_3?().\mathcal{V}_\epsilon\big(V\big)\,(r_3,r_4)$$

There are several issues with this breakdown. One of them is typability: we have that $r_1 : \mu\mathsf{t}.?(\mathsf{Int});\mathsf{t}$, but subprocess $\overline{c_2}!\langle x\rangle.\mathbf{0}$ is not typable under a linear environment containing such a judgment. Another, perhaps more central, issue concerns $\tilde{r}$: the last trio (which mimics application) should apply to the sequence of names (r_1,r_2), rather than to (r_3,r_4). We address both issues by devising a mechanism that propagates names with recursive types (such as (r_1,r_2)) among the trios that use some of them. This entails decomposing R in such a way that the first two trios propagate r_1 and r_2 after they have used them; the trio simulating $V\ r$ should then have a way to access the propagated names (r_1,r_2).

We illustrate the key insights underpinning our solution by means of two examples. The first one illustrates how to break down input and output actions on names with recursive types (the "first part" of R). The second example shows how to break down an application where a value is applied to a tuple of names with recursive names (the "second part" of R).

▶ **Example 3.21** (Decomposing Processes with Recursive Names (I))**.** Let $P = r?(x).r!\langle x\rangle.P'$ be a process where r has type $S = \mu \mathsf{t}.?(\mathsf{Int});!\langle \mathsf{Int}\rangle;\mathsf{t}$ and $r \in \mathtt{fn}(P')$. To define $\mathcal{B}^1_\epsilon\big(P\big)$ in a compositional way, names (r_1, r_2) should be provided to its first trio; they cannot be known beforehand. To this end, we introduce a new control trio that will hold these names:

$$c^r?(b).b\,(r_1, r_2)$$

where the *shared* name c^r provides a decomposition of the (recursive) name r. The intention is that each name with a recursive type r will get its own dedicated propagator channel c^r. Since there is only one recursive name in P, its decomposition will be of the following form:

$$\mathcal{D}(P) = (\nu\, \tilde{c})(\nu\, c^r)\big(c^r?(b).b\,(r_1, r_2) \mid \overline{c_1}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^1_\epsilon\big(P\big)\big)$$

The new control trio can be seen as a server that provides names: each trio that mimics some action on r should request the sequence $\tilde{r}$ from the server on c^r. This request will be realized by a higher-order communication: trios should send an abstraction to the server; such an abstraction will contain further actions of a trio and it will be applied to the sequence $\tilde{r}$. Following this idea, we may refine the definition of $\mathcal{D}(P)$ by expanding $\mathcal{B}^1_\epsilon\big(P\big)$:

$$\mathcal{D}(P) = (\nu\, \tilde{c})(\nu\, c^r)\big(c^r?(b).b\,(r_1, r_2) \mid \overline{c_1}!\langle\rangle.\mathbf{0} \mid c_1?().c^r!\langle N_1\rangle.\mathbf{0} \mid c_2?(y).c^r!\langle N_2\rangle.\mathbf{0} \mid \mathcal{B}^3_\epsilon\big(P'\big)\big)$$

The trios involving names with recursive types have now a different shape. After being triggered by a previous trio, rather than immediately mimicking an action, they will send an abstraction to the server available on c^r. The abstractions N_1 and N_2 are defined as follows:

$$N_1 = \lambda(z_1, z_2).\, z_1?(x).\overline{c_2}!\langle x\rangle.c^r?(b).b\,(z_1, z_2) \quad N_2 = \lambda(z_1, z_2).\, z_2!\langle x\rangle.\overline{c_3}!\langle\rangle.c^r?(b).b\,(z_1, z_2)$$

Hence, the formal arguments for these values are meant to correspond to $\tilde{r}$. The server on name c^r will appropriately instantiate these names. Notice that all names in $\tilde{r}$ are propagated, even if the abstractions only use some of them. For instance, N_1 only uses r_1, whereas N_2 uses r_2. After simulating an action on r_i and activating the next trio, these values reinstate the server on c^r for the benefit of future trios mimicking actions on r. ⌟

▶ **Example 3.22** (Decomposing Processes with Recursive Names (II))**.** Let $S = \mu \mathsf{t}.?(\mathsf{Int});!\langle \mathsf{Int}\rangle;\mathsf{t}$ and $T = \mu \mathsf{t}.?(\mathsf{Bool});!\langle \mathsf{Bool}\rangle;\mathsf{t}$, and define $Q = V\,(u, v)$ as a process where $u : S$ and $v : T$, where V is some value of type $(S, T) \to \diamond$. The decomposition of Q is as in the previous example, except that now we need two servers, one for u and one for v:

$$\mathcal{D}(Q) = (\nu\, c_1\tilde{c})(\nu\, c^u c^v)\big(c^u?(b).b\,(u_1, u_2) \mid c^v?(b).b\,(v_1, v_2) \mid \overline{c_1}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^1_\epsilon\big(Q\big)\big)$$

where $\tilde{c} = (c_2, \ldots, c_{|Q|})$. We should break down Q in such a way that it could communicate with both servers to collect sequences $\tilde{u}$ and $\tilde{v}$. To this end, we define a process in which abstractions are nested using output prefixes and whose innermost process is an application. After successive communications with multiple servers this innermost application will have collected all names in $\tilde{u}$ and $\tilde{v}$. We apply this idea to breakdown Q:

$$\mathcal{B}^1_\epsilon\big(Q\big) = c_1?().c^u!\big\langle\lambda(x_1, x_2).\, c^v!\langle\lambda(y_1, y_2).\, \mathcal{V}^2_\epsilon\big(V\big)\,(x_1, x_2, y_1, y_2)\rangle.\mathbf{0}\big\rangle.\mathbf{0}$$

Observe that we use two nested outputs, one for each name with recursive types in Q. We now look at the reductions of $\mathcal{D}(Q)$ to analyze how the communication of nested abstractions allows us to collect all name sequences needed. After the first reduction along c_1 we have:

$$\begin{aligned}\mathcal{D}(Q) \longrightarrow &(\nu\,\tilde{c})(\nu\,c^u c^v)\big(c^u?(b).b\,(u_1,u_2) \mid c^v?(b).b\,(v_1,v_2) \mid \\ &c^u!\langle\lambda(x_1,x_2).\,c^v!\langle\lambda(y_1,y_2).\,\mathcal{V}_\epsilon^2\big(V\big)\,(x_1,x_2,y_1,y_2)\rangle.\mathbf{0}\rangle.\mathbf{0} = R^1\end{aligned}$$

From R^1 we have a synchronization along name c^u:

$$\begin{aligned}R^1 \longrightarrow &(\nu\,\tilde{c})(\nu\,c^u c^v)\big(\lambda(x_1,x_2).\,c^v!\langle\lambda(y_1,y_2).\,\mathcal{V}_\epsilon^2\big(V\big)\,(x_1,x_2,y_1,y_2)\rangle.\mathbf{0}\,(u_1,u_2) \mid \\ &c^v?(b).b\,(v_1,v_2)\big) = R^2\end{aligned}$$

Upon receiving the value, the server applies it to (u_1,u_2) obtaining the following process:

$$R^2 \longrightarrow(\nu\,\tilde{c})(\nu\,c^u c^v)\big(c^v!\langle\lambda(y_1,y_2).\,\mathcal{V}_\epsilon^2\big(V\big)\,(u_1,u_2,y_1,y_2)\rangle.\mathbf{0} \mid c^v?(b).b\,(v_1,v_2)\big) = R^3$$

Up to here, we have partially instantiated name variables of a value with the sequence $\tilde{u}$. Next, the first trio in R^3 can communicate with the server on name c^v:

$$\begin{aligned}R^3 &\longrightarrow(\nu\,\tilde{c})(\nu\,c^u c^v)\big(\lambda(y_1,y_2).\,\mathcal{V}_\epsilon^2\big(V\big)\,(u_1,u_2,y_1,y_2)\,(v_1,v_2)\big) \\ &\longrightarrow(\nu\,\tilde{c})(\nu\,c^u c^v)\big(\mathcal{V}_\epsilon^2\big(V\big)\,(u_1,u_2,v_1,v_2)\big)\end{aligned}$$

This completes the instantiation of name variables with appropriate sequences of names with recursive types. At this point, $\mathcal{D}(Q)$ can proceed to mimic the application in Q. ⌟

These two examples illustrate the main ideas of the decomposition of processes that involve names with recursive types. Tab. 4 presents a formal account of the extension of the definition of process decomposition given in Def. 3.8. Before explaining the table in detail, we require an auxiliary definition.

Given an unfolded recursive session type S, the auxiliary function $f(S)$ returns the position of the top-most prefix of S within its body. (Whenever $S = \mu\mathsf{t}.S'$, we have $f(S) = 1$.)

▶ **Definition 3.23** (Index function). *Let S be an (unfolded) recursive session type. The function $f(S)$ is defined as follows:*

$$f(S) = \begin{cases} f_0'(S'\{^S/_{\mathsf{t}}\}) & \text{if } S = \mu\mathsf{t}.S' \\ f_0'(S) & \text{otherwise}\end{cases}$$

where: $f_l'(\mu\mathsf{t}.S) = |\mathcal{R}(S)| - l + 1, \qquad f_l'(!\langle U\rangle;S) = f_{l+1}'(S), \qquad f_l'(?(U);S) = f_{l+1}'(S).$

▶ **Example 3.24.** Let $S' = ?(\mathsf{Bool});!\langle\mathsf{Bool}\rangle;S$ where S is as in Exam. 3.19. Then $f(S') = 2$ since the top-most prefix of S' ('$?(\mathsf{Bool});$') is the second prefix in the body of S. ⌟

Given a typed process P, we write $\mathtt{rn}(P)$ to denote the set of free names of P whose types are recursive. As mentioned above, for each $r \in \mathtt{rn}(P)$ with $r : S$ we shall rely on a control trio of the form $c^r?(b).b\,\widetilde{r}$, where $\widetilde{r} = r_1,\ldots,r_{|\mathcal{G}(S)|}$.

▶ **Definition 3.25** (Decomposition of a Process with Recursive Session Types). *Let P be a closed* HO *process with $\widetilde{u} = \mathtt{fn}(P)$ and $\widetilde{v} = \mathtt{rn}(P)$. The* decomposition *of P, denoted $\mathcal{D}(P)$, is defined as:*

$$\mathcal{D}(P) = (\nu\,\widetilde{c})(\nu\,\widetilde{c_r})\Big(\prod_{r\in\widetilde{v}} c^r?(b).b\,\widetilde{r} \mid \overline{c_k}!\langle\rangle.\mathbf{0} \mid \mathcal{B}_\epsilon^k\big(P\sigma\big)\Big)$$

where: $k > 0$; $\widetilde{c} = (c_k,\ldots,c_{k+|P|-1})$; $\widetilde{c_r} = \bigcup_{r\in\widetilde{v}} c^r$; $\sigma = \{^{\mathsf{init}(\widetilde{u})}/_{\widetilde{u}}\}$.

Table 4 The breakdown function for processes and values (extension with recursive types).

$\mathcal{B}^k_{\widetilde{x}}\big(r!\langle V\rangle.Q\big)$	
$c_k?(\widetilde{x}).c^r!\big\langle N_V\big\rangle \mid \mathcal{B}^{k+l+1}_{\widetilde{w}}\big(Q\big)$ where: $N_V = \lambda\widetilde{z}.\, z_{f(S)}!\big\langle\mathcal{V}^{k+1}_{\widetilde{y}}\big(V\big)\big\rangle.$ $\overline{c_{k+l+1}}!\langle\widetilde{w}\rangle.c^r?(b).(b\,\widetilde{z})$	$r : S \wedge \mathsf{tr}(S)$ $\widetilde{y} = \mathtt{fv}(V),\ \widetilde{w} = \mathtt{fv}(Q)$ $l = \lvert V\rvert$ $\widetilde{z} = (z_1, \ldots, z_{\lvert\mathcal{R}^\star(S)\rvert})$
$\mathcal{B}^k_{\widetilde{x}}\big(r?(y).Q\big)$	
$c_k?(\widetilde{x}).c^r!\big\langle N_y\big\rangle \mid \mathcal{B}^{k+1}_{\widetilde{x}'}\big(Q\big)$ where: $N_y = \lambda\widetilde{z}.\, z_{f(S)}?(y).\overline{c_{k+1}}!\langle\widetilde{x}'\rangle.c^r?(b).(b\,\widetilde{z})$	$r : S \wedge \mathsf{tr}(S)$ $\widetilde{x}' = \mathtt{fv}(Q)$ $\widetilde{z} = (z_1, \ldots, z_{\lvert\mathcal{R}^\star(S)\rvert})$
$\mathcal{B}^k_{\widetilde{x}}\big(V\,(\widetilde{r}, u_i)\big)$	
$c_k?(\widetilde{x}).\overbrace{c^{r_1}!\big\langle\lambda\widetilde{z}_1.c^{r_2}!\langle\lambda\widetilde{z}_2.\cdots.c^{r_n}!\langle\lambda\widetilde{z}_n.Q\rangle\big\rangle}^{n=\lvert\tilde{r}\rvert}\big\rangle$ where: $Q = \mathcal{V}^{k+1}_{\widetilde{x}}\big(V\big)\,(\widetilde{z}_1, \ldots, \widetilde{z}_n, \widetilde{m})$	$\forall r_i \in \widetilde{r}.(r_i : S_i \wedge \mathsf{tr}(S_i) \wedge$ $\widetilde{z_i} = (z^i_1, \ldots, z^i_{\lvert\mathcal{R}^\star(S_i)\rvert}))$ $u_i : C$ $\widetilde{m} = (u_i, \ldots, u_{i+\lvert\mathcal{G}(C)\rvert-1})$
$\mathcal{B}^k_{\widetilde{x}}\big((\nu\, s : \mu\mathrm{t}.S)P'\big)$	
$(\nu\,\widetilde{s} : \mathcal{R}(S))(\nu\, c^s)c^s?(b).(b\,\widetilde{s}) \mid$ $(\nu\, c^{\widetilde{s}})c^{\widetilde{s}}?(b).(b\,\widetilde{\overline{s}}) \mid \mathcal{B}^k_{\widetilde{x}}\big(P'\big)$	$\mathsf{tr}(\mu\mathrm{t}.S)$ $\widetilde{s} = (s_1, \ldots, s_{\lvert\mathcal{R}(S)\rvert})$ $\widetilde{\overline{s}} = (\overline{s_1}, \ldots, \overline{s_{\lvert\mathcal{R}(S)\rvert}})$
$\mathcal{V}^k_{\widetilde{x}}\big(\lambda(\widetilde{y}, z) : (\widetilde{S}, C)^{\rightsquigarrow}.P\big)$	
$\lambda(\widetilde{y^1}, \ldots, \widetilde{y^n}, \widetilde{z}) : (\widetilde{T})^{\rightsquigarrow}.N$ where: $\widetilde{T} = (\mathcal{G}(S_1), \ldots, \mathcal{G}(S_n), \mathcal{G}(C))$ $N = (\nu\,\widetilde{c})\prod_{i\in\lvert\widetilde{y}\rvert}(c^{y_i}?(b).(b\,\widetilde{y}^i)) \mid \overline{c_k}!\langle\widetilde{x}\rangle \mid$ $\mathcal{B}^k_{\widetilde{x}}\big(P\{^{z_1}/_z\}\big)$	$\forall y_i \in \widetilde{y}.(y_i : S_i \wedge \mathsf{tr}(S_i) \wedge$ $\widetilde{y^i} = (y^i_1, \ldots, y^i_{\lvert\mathcal{G}(S_i)\rvert}))$ $\widetilde{z} = (z_1, \ldots, z_{\lvert\mathcal{G}(C)\rvert})$ $\widetilde{c} = \begin{cases}\epsilon & \text{if } \rightsquigarrow = \multimap \\ (c_k, \ldots, c_{k+\lvert P\rvert-1}) & \text{if } \rightsquigarrow = \to\end{cases}$

We now describe the required extensions for the function $\mathcal{B}^k_{\widetilde{x}}\big(\,\cdot\,\big)$. We will use predicate $\mathsf{tr}(S)$ on types to indicate that S is a tail-recursive session type. Tab. 4 describes the breakdown of prefixes whose type is recursive; all other prefixes can be treated as in Tab. 1.

Output. The breakdown of process $r!\langle V\rangle.Q$, when r has a recursive type S, is as follows:

$$\begin{aligned}\mathcal{B}^k_{\widetilde{x}}\big(r!\langle V\rangle.Q\big) = \;& c_k?(\widetilde{x}).c^r!\big\langle N_V\big\rangle \mid \mathcal{B}^{k+l+1}_{\widetilde{w}}\big(Q\big)\\ & \text{where } N_V = \lambda\widetilde{z}.\, z_{f(S)}!\big\langle\mathcal{V}^{k+1}_{\widetilde{y}}\big(V\big)\big\rangle.\overline{c_{k+l+1}}!\langle\widetilde{w}\rangle.c^r?(b).(b\,\widetilde{z})\end{aligned}$$

The decomposition consists of a leading trio that mimics the output action running in parallel with the breakdown of Q. After receiving the context $\widetilde{x}$, the leading trio sends an abstraction N_V along c^r. Value N_V performs several tasks. First, it collects the sequence $\tilde{r}$; then, it mimics the output action of P along one of them ($r_{f(S)}$) and triggers the next trio, with context $\widetilde{w}$; finally, it reinstates the server on c^r for the next trio that

uses r. Notice that differently from what is done in Tab. 1, indexing is not relevant when breaking down names with recursive types. Also, since by definition $\mathcal{V}_{\widetilde{y}}^{k}(y) = y$, $y\sigma = y$, and $|y| = 0$, when the communicated value V is a variable y we obtain the following:

$$\mathcal{B}_{\widetilde{x}}^{k}(r!\langle y\rangle.Q) = c_k?(\widetilde{x}).c^{r}!\langle\lambda\widetilde{z}.\, z_{f(S)}!\langle y\rangle.\overline{c_{k+1}}!\langle\widetilde{w}\rangle.c^{r}?(b).(b\,\widetilde{z})\rangle \mid \mathcal{B}_{\widetilde{w}}^{k+1}(Q)$$

Input. The breakdown of process $r?(y).Q$, when r has recursive session type S, is as follows:

$$\mathcal{B}_{\widetilde{x}}^{k}(r?(y).Q) = c_k?(\widetilde{x}).c^{r}!\langle\lambda\widetilde{z}.\, z_{f(S)}?(y).\overline{c_{k+1}}!\langle\widetilde{x}'\rangle.c^{r}?(b).(b\,\widetilde{z})\rangle \mid \mathcal{B}_{\widetilde{x}'}^{k+1}(Q)$$

The breakdown follows the lines of the output case, but also of the linear case in Tab. 1, with additional structure needed to implement the reception of $\tilde{r}$, using one of the received names ($r_{f(S)}$) as a subject for the input action and propagating those names further.

Application. For simplicity we consider applications $V\,(\widetilde{r}, u_i)$, where names in $\widetilde{r}$ have recursive types and only name u_i has a non-recursive type; the general case involving different orders in names and multiple names with non-recursive types is as expected. We have:

$$\mathcal{B}_{\widetilde{x}}^{k}(V\,(\widetilde{r}, u_i)) = c_k?(\widetilde{x}).\overbrace{c^{r_1}!\langle\lambda\widetilde{z}_1.c^{r_2}!\langle\lambda\widetilde{z}_2.\cdots.c^{r_n}!\langle\lambda\widetilde{z}_n.}^{n=|\tilde{r}|}\mathcal{V}_{\widetilde{x}}^{k+1}(V)\,(\widetilde{z}_1,\ldots,\widetilde{z}_n,\widetilde{m})\rangle\rangle$$

We rely on types to decompose every name in $(\widetilde{r}, u_i)$. Letting $|\tilde{r}| = n$ and $i \in \{1,\ldots,n\}$, for each $r_i \in \widetilde{r}$ (with $r_i : S_i$) we generate a sequence $\widetilde{z}_i = (z_1^i,\ldots,z_{|\mathcal{R}^\star(S_i)|}^i)$ as in the output case. Since name u_i has a non-recursive session type, we decompose it as in Tab. 1. Subsequently, we define an output action on propagator c^{r_1} that sends a value containing n abstractions that occur nested within output prefixes: for each $j \in \{1,\ldots,n-1\}$, each abstraction binds $\widetilde{z}_j$ and sends the next abstraction along $c^{r_{j+1}}$. The innermost abstraction abstracts over $\widetilde{z}_n$ and encapsulates process $\mathcal{V}_{\widetilde{x}}^{k+1}(V)\,(\widetilde{z}_1,\ldots,\widetilde{z}_n,\widetilde{m})$, which mimics the application in the source process. By this abstraction nesting we bind all variables $\widetilde{z}_i$ in Q. This structure can be seen as an encoding of partial application: by virtue of a single synchronization on c^{r_i} part of variables (i.e., $\widetilde{z}_i$) will be instantiated.

The breakdown of a value application of the form $y\,(\widetilde{r}, u_i)$ results into a specific form of the breakdown:

$$\mathcal{B}_{\widetilde{x}}^{k}(y\,(\widetilde{r}, u_i)) = c_k?(\widetilde{x}).\overbrace{c^{r_1}!\langle\lambda\widetilde{z}_1.c^{r_2}!\langle\lambda\widetilde{z}_2.\cdots.c^{r_n}!\langle\lambda\widetilde{z}_n.}^{n=|\tilde{r}|}y\,(\widetilde{z}_1,\ldots,\widetilde{z}_n,\widetilde{m})\rangle\rangle$$

Restriction. The restriction process $(\nu\, s : \mu\mathrm{t}.S)P'$ is translated as follows:

$$\mathcal{B}_{\widetilde{x}}^{k}((\nu\, s : \mu t.S)P') = (\nu\,\widetilde{s} : \mathcal{R}(S))(\nu\, c^{s})c^{s}?(b).(b\,\widetilde{s}) \mid (\nu\, c^{\overline{s}})c^{\overline{s}}?(b).(b\,\widetilde{\overline{s}}) \mid \mathcal{B}_{\widetilde{x}}^{k}(P')$$

We decompose s into $\widetilde{s} = (s_1,\ldots,s_{|\mathcal{R}(S)|})$ and $\overline{s}$ into $\widetilde{\overline{s}} = (\overline{s_1},\ldots,\overline{s_{|\mathcal{R}(S)|}})$. The breakdown introduces two servers in parallel with the breakdown of P'; these servers provide names for s and $\overline{s}$ along c^{s} and $c^{\overline{s}}$, respectively. The server on c^{s} (resp. $c^{\overline{s}}$) receives a value and applies it to the sequence $\widetilde{s}$ (resp. $\widetilde{\overline{s}}$). We restrict over $\widetilde{s}$ and propagators c^{s} and $c^{\overline{s}}$.

Value. The polyadic value $\lambda(\widetilde{y}, z) : (\widetilde{S}, C)^{\rightsquigarrow}.P$, where $\rightsquigarrow \in \{\multimap, \to\}$, is decomposed as follows:

$$\begin{aligned}\mathcal{V}_{\widetilde{x}}^{k}(\lambda(\widetilde{y}, z) : (\widetilde{S}, C)^{\rightsquigarrow}.P) = &\lambda(\widetilde{y^1},\ldots,\widetilde{y^n},\widetilde{z}) : (\mathcal{G}(S_1),\ldots,\mathcal{G}(S_n),\mathcal{G}(C))^{\rightsquigarrow}.N\\ &\text{where: } N = (\nu\,\widetilde{c})\prod_{i\in|\widetilde{y}|}(c^{y_i}?(b).(b\,\widetilde{y}^{i})) \mid \overline{c_k}!\langle\widetilde{x}\rangle \mid \mathcal{B}_{\widetilde{x}}^{k}(P\{z_1/z\})\end{aligned}$$

We assume variables in $\widetilde{y}$ have recursive session types $\widetilde{S}$ and variable z has some non-recursive session type C; the general case involving different orders in variables and multiple variables with non-recursive types is as expected. Every variable y_i (with $y_i : S_i$)

is decomposed into $\widetilde{y}^i = (y_1, \ldots, y_{|\mathcal{G}(S_i)|})$. Variable z is decomposed as in Tab. 1. The breakdown is similar to the (monadic) shared value given in Tab. 1. In this case, for every $y_i \in \widetilde{y}$ there is a server $c^{y_i}?(b).(b\,\widetilde{y}^i)$ as a subprocess in the abstracted composition. The rationale for these servers is as described in previous cases.

To sum up, each trio using a name with a recursive session type first receives a sequence of names; then, it uses one of such names to mimic the appropriate action; finally, it propagates the entire sequence by reinstating a server defined as a control trio. Interestingly, this scheme for name propagation follows the implementation of the encoding of name-passing in HO.

▶ **Example 3.26** (Breakdown of Recursion Encoding)**.** Consider the recursive process $P = \mu X.a?(m).a!\langle m\rangle.X$, which is not an HO process. P can be encoded into HO as follows [19]:

$$[\![P]\!] = a?(m).a!\langle m\rangle.(\nu\, s)(V\,(a,s) \mid s!\langle V\rangle.\mathbf{0})$$

where the value V is an abstraction that potentially reduces to $[\![P]\!]$:

$$V = \lambda(x_a, y_1).\, y_1?(z_x).x_a?(m).x_a!\langle m\rangle.(\nu\, s)(z_x\,(x_a,s) \mid \overline{s}!\langle z_x\rangle.\mathbf{0})$$

We compose $[\![P]\!]$ with an appropriate client process to illustrate the encoding of recursion:

$$\begin{aligned}
&[\![P]\!] \mid a!\langle W\rangle.a?(b).R\\
&\longrightarrow^2 (\nu\, s)(V\,(a,s) \mid s!\langle V\rangle.\mathbf{0}) \mid R\\
&\longrightarrow (\nu\, s)(s?(z_x).a?(m).a!\langle m\rangle.(\nu\, s')(z_x\,(a,s') \mid \overline{s'}!\langle z_x\rangle.\mathbf{0}) \mid s!\langle V\rangle.\mathbf{0}) \mid R\\
&\longrightarrow a?(m).a!\langle m\rangle.(\nu\, s')(V\,(a,s') \mid \overline{s'}!\langle V\rangle.\mathbf{0}) \mid R = [\![P]\!] \mid R
\end{aligned}$$

where R is some unspecified process such that $a \in \mathtt{rn}(R)$. We now analyze $\mathcal{D}([\![P]\!])$ and its reduction chain. By Def. 3.5, we have $|[\![P]\!]| = 7$, and $|V| = 0$. Then, we choose $k = 1$ and observe that $\sigma = \{^{a_1\overline{a_1}}/_{a\overline{a}}\}$. Following Def. 3.25, we get:

$$\begin{aligned}
\mathcal{D}([\![P]\!]) &= (\nu\, c_1, \ldots, c_7)(\nu\, c^a)(c^a?(b).b\,(a_1,a_2) \mid \overline{c_1}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^1_\epsilon\big([\![P]\!]\sigma\big))\\
\mathcal{B}^1_\epsilon\big([\![P]\!]\big) &= c_1?().c^a!\langle\lambda(z_1,z_2).\, z_1?(m).\overline{c_2}!\langle m\rangle.c^a?(b).b\,(z_1,z_2)\rangle.\mathbf{0}\\
&\quad \mid c_2?(m).c^a!\langle\lambda(z_1,z_2).\, z_2!\langle m\rangle.\overline{c_3}!\langle\rangle.c^a?(b).b\,(z_1,z_2)\rangle.\mathbf{0}\\
&\quad \mid (\nu\, s_1)\big(c_3?().\overline{c_4}!\langle\rangle.\overline{c_5}!\langle\rangle.\mathbf{0} \mid c_4?().\overline{c^a}!\langle\lambda(z_1,z_2).\, \mathcal{V}^5_\epsilon\big(V\big)\,(z_1,z_2,s_1)\rangle.\mathbf{0}\\
&\qquad \mid c_5?().\overline{s}_1!\langle\mathcal{V}^6_\epsilon\big(V\big)\rangle.\overline{c_7}!\langle\rangle.\mathbf{0} \mid c_7?().\mathbf{0}\big)
\end{aligned}$$

The decomposition relies twice on the breakdown of value V, so we give $\mathcal{V}^k_\epsilon\big(V\big)$ here for arbitrary $k > 0$. For this, we observe that V is an abstraction of a process Q with $|Q| = 7$.

$$\begin{aligned}
\mathcal{V}^k_\epsilon\big(V\big) &= \lambda(x_{a_1}, x_{a_2}, y_1).\,(\nu\, c_k, \ldots, c_{k+6})(c^{x_a}?(b).b\,(x_{a_1},x_{a_2}) \mid \overline{c_k}!\langle\rangle.\mathbf{0} \mid \mathcal{B}^k_\epsilon\big(Q\big))\\
\mathcal{B}^k_\epsilon\big(Q\big) &= c_k?().y_1?(z_x).\overline{c_{k+1}}!\langle z_x\rangle.\mathbf{0}\\
&\quad \mid c_{k+1}?(z_x).c^a_1!\langle\lambda(z_1,z_2).\, z_1?(m).\overline{c_{k+2}}!\langle z_x, m\rangle.c^a_2?(b).b\,(z_1,z_2)\rangle.\mathbf{0}\\
&\quad \mid c_{k+2}?(z_x).c^a_2!\langle\lambda(z_1,z_2).\, z_2!\langle m\rangle.\overline{c_{k+3}}!\langle z_x\rangle.c^a_3?(b).b\,(z_1,z_2)\rangle.\mathbf{0}\\
&\quad \mid (\nu\, s_1)\big(c_{k+3}?(x_z).\overline{c_{k+4}}!\langle z_x\rangle.\overline{c_{k+5}}!\langle z_x\rangle.\mathbf{0}\\
&\quad \mid c_{k+4}?(z_x).c^a_3!\langle\lambda(z_1,z_2).\, z_x\,(z_1,z_2,s_1)\rangle.\mathbf{0} \mid c_{k+5}?(z_x).\overline{s_1}!\langle z_x\rangle.\overline{c_{k+6}}!\langle\rangle.\mathbf{0} \mid c_{k+6}?().\mathbf{0}\big)
\end{aligned}$$

We follow the reduction chain on $\mathcal{D}([\![P]\!])$ until it is ready to mimic the first action with channel a, which is an input. First, c_1 will synchronize, after which c^a sends the abstraction

to which then (a_1, a_2) is applied. We obtain $\mathcal{D}([\![P]\!]) \longrightarrow^3 (\nu\, c_2, \ldots, c_7, c^a)P'$, where

$$\begin{aligned} P' = {} & a_1?(m).\overline{c_2}!\langle m\rangle.c^a?(b).b\,(a_1, a_2) \\ & \mid c_2?(m).c^a!\langle\lambda(z_1, z_2).\, z_2!\langle m\rangle.\overline{c_3}!\langle\rangle.c^a?(b).b\,(z_1, z_2)\rangle.\mathbf{0} \\ & \mid (\nu\, s_1)\big(c_3?().\overline{c_4}!\langle\rangle.\overline{c_5}!\langle\rangle.\mathbf{0} \mid c_4?().\overline{c^a}!\langle\lambda(z_1, z_2).\, \mathcal{V}_\epsilon^5\big(V\big)\,(z_1, z_2, s_1)\rangle.\mathbf{0} \\ & \qquad \mid c_5?().\overline{s}_1!\langle\mathcal{V}_\epsilon^6\big(V\big)\rangle.\overline{c_7}!\langle\rangle.\mathbf{0} \mid c_7?().\mathbf{0}\big) \end{aligned}$$

Note that this process is awaiting an input on channel a_1, after which c_2 can synchronize with its dual. At that point, c^a is ready to receive another abstraction that mimics an input on a_1. This strongly suggests a tight operational correspondence between a process P and its decomposition in the case where P performs higher-order recursion. ⌟

Below we write Δ_μ to denote a session environment that concerns only recursive types. We state our main results:

▶ **Theorem 3.27** (Typability of Breakdown)**.** *Let P be an initialized* HO *process and V be a value.*

1. *If $\Gamma; \Lambda; \Delta, \Delta_\mu \vdash P \triangleright \diamond$ then $\mathcal{G}(\Gamma_1), \Phi; \emptyset; \mathcal{G}(\Delta), \Theta \vdash \mathcal{B}_{\widetilde{x}}^k\big(P\big) \triangleright \diamond$ where: $\widetilde{r} = dom(\Delta_\mu)$; $\Phi = \prod_{r\in\widetilde{r}} c^r : \langle\mathcal{R}^\star(\Delta_\mu(r)) \multimap \diamond\rangle$; $\widetilde{x} = \mathtt{fv}(P)$; $k > 0$; $\Gamma_1 = \Gamma \setminus \widetilde{x}$; and* $\mathsf{balanced}(\Theta)$ *with $dom(\Theta) = \{c_k, \ldots, c_{k+|P|-1}\} \cup \{\overline{c_{k+1}}, \ldots, \overline{c_{k+|P|-1}}\}$ such that $\Theta(c_k) = ?(\widetilde{M}); \mathtt{end}$, where $\widetilde{M} = (\mathcal{G}(\Gamma), \mathcal{G}(\Lambda))(\widetilde{x})$.*
2. *If $\Gamma; \Lambda; \Delta \vdash V \triangleright C \multimap \diamond$ then $\mathcal{G}(\Gamma); \mathcal{G}(\Lambda); \mathcal{G}(\Delta), \Theta \vdash \mathcal{V}_{\widetilde{x}}^k\big(V\big) \triangleright \mathcal{G}(C) \multimap \diamond$, where: $\widetilde{x} = \mathtt{fv}(V)$; $k > 0$; and* $\mathsf{balanced}(\Theta)$ *with $dom(\Theta) = \{c_k, \ldots, c_{k+|V|-1}\} \cup \{\overline{c_k}, \ldots, \overline{c_{k+|V|-1}}\}$ such that $\Theta(c_k) = ?(\widetilde{M}); \mathtt{end}$ and $\Theta(\overline{c_k}) = !\langle\widetilde{M}\rangle; \mathtt{end}$, where $\widetilde{M} = (\mathcal{G}(\Gamma), \mathcal{G}(\Lambda))(\widetilde{x})$.*
3. *If $\Gamma; \emptyset; \emptyset \vdash V \triangleright C \to \diamond$ then $\mathcal{G}(\Gamma); \emptyset; \emptyset \vdash \mathcal{V}_{\widetilde{x}}^k\big(V\big) \triangleright \mathcal{G}(C) \to \diamond$ where $\widetilde{x} = \mathtt{fv}(V)$ and $k > 0$.*

Proof. By mutual induction on the structure of P and V. ◀

▶ **Theorem 3.28** (Typability of the Decomposition with Recursive Types)**.** *Let P be a closed* HO *process with $\widetilde{u} = \mathtt{fn}(P)$ and $\widetilde{v} = \mathtt{rn}(P)$. If $\Gamma; \emptyset; \Delta, \Delta_\mu \vdash P \triangleright \diamond$, where Δ_μ only involves recursive session types, then $\mathcal{G}(\Gamma\sigma); \emptyset; \mathcal{G}(\Delta\sigma), \mathcal{G}(\Delta_\mu\sigma) \vdash \mathcal{D}(P) \triangleright \diamond$, where $\sigma = \{^{\mathsf{init}(\widetilde{u})}/_{\widetilde{u}}\}$.*

Proof. Directly from the definitions, using Thm. 3.27. ◀

▶ Remark 3.29 (Non-Tail-Recursive Session Types). Our definitions and results apply to tail-recursive session types. We can accommodate the non-tail-recursive type $\mu\mathsf{t}.?((\widetilde{T}, \mathtt{t}) \to \diamond); \mathtt{end}$ into our approach: in Def. 3.18, we need to have $\mathcal{G}(\mu\mathsf{t}.S) = \mu\mathsf{t}.\mathcal{G}(S)$ if $\mu\mathsf{t}.S$ is non-tail-recursive. The decomposition functions for non-recursive session types suffice in this case.

4 Optimizations of the Decomposition

Here we briefly discuss two optimizations of the decompositions. They simplify the structure of trios and the underlying communication discipline. Interestingly, they are both enabled by the higher-order nature of HO. In fact, they hinge on *thunk processes*, i.e., inactive processes that can be activated upon reception. We write $\{\{P\}\}$ to stand for the thunk process $\lambda x : \langle\mathtt{end} \to \diamond\rangle.\, P$, with $x \notin \mathtt{fn}(P)$. We write $\mathtt{run}\,\{\{P\}\}$ to denote the application of a thunk to a (dummy) name of type $\mathtt{end} \to \diamond$. This way, we have $\mathtt{run}\,\{\{P\}\} \longrightarrow P$.

From Trios to Duos. We can simplify the breakdown functions by replacing trios with *duos*, i.e., processes with exactly two sequential prefixes. The idea is to transform trios such as $c_k?(\widetilde{x}).u!\langle V\rangle.c_{k+1}!\langle\widetilde{y}\rangle$ into the composition of a duo with a control trio:

$$c_k?(\widetilde{x}).c_{k+1}!\big\langle\{\!\{u!\langle V\rangle.c_{k+2}!\langle\widetilde{z}\rangle\}\!\}\big\rangle \mid c_{k+1}?(b).(\mathtt{run}\,b) \tag{4}$$

The first action is as before; the two remaining prefixes are encapsulated into a thunk. This thunk is sent via a propagator to the control trio that activates it upon reception. This transformation involves an additional propagator, denoted c_{k+2} above. This requires minor modifications in the definition of the degree function $|\cdot|$ (cf. Def. 3.5).

In some cases, the breakdown function in § 3.2 already produces duos. Breaking down input and output prefixes and parallel composition involves proper trios; following the scheme illustrated by (4), we can define a map $\{\!|\cdot|\!\}$ to transform these trios into duos:

$$\begin{aligned}
\{\!|c_k?(\widetilde{x}).u_i!\langle V\rangle.\overline{c_{k+1}}!\langle\widetilde{z}\rangle|\!\} &= c_k?(\widetilde{x}).\overline{c_{k+1}}!\big\langle\{\!\{u_i!\langle V\rangle.\overline{c_{k+2}}!\langle\widetilde{z}\rangle\}\!\}\big\rangle \mid c_{k+1}?(b).(\mathtt{run}\,b)\\
\{\!|c_k?(\widetilde{x}).u_i?(y).\overline{c_{k+1}}!\langle\widetilde{x}'\rangle|\!\} &= c_k?(\widetilde{x}).\overline{c_{k+1}}!\big\langle\{\!\{u_i?(y).\overline{c_{k+2}}!\langle\widetilde{x}'\rangle\}\!\}\big\rangle \mid c_{k+1}?(b).(\mathtt{run}\,b)\\
\{\!|c_k?(\widetilde{x}).\overline{c_{k+1}}!\langle\widetilde{y}\rangle.\overline{c_{k+l+1}}!\langle\widetilde{z}\rangle|\!\} &= c_k?(\widetilde{x}).\overline{c_{k+1}}!\big\langle\{\!\{\overline{c_{k+2}}!\langle\widetilde{y}\rangle.\overline{c_{k+l+2}}!\langle\widetilde{z}\rangle\}\!\}\big\rangle \mid c_{k+1}?(b).(\mathtt{run}\,b)
\end{aligned}$$

In the breakdown given in § 3.3 there is a proper trio, which can be transformed as follows:

$$\{\!|u_i \triangleleft l_j.u_i?(z).\overline{c_k}!\big\langle\widetilde{x}\big\rangle.(z\,\widetilde{y})|\!\} = u_i \triangleleft l_j.\overline{c_k}!\big\langle\{\!\{u_i?(z).\overline{c_{k+1}}!\langle\widetilde{x}\rangle.z\,\widetilde{y}\}\!\}\big\rangle \mid c_k?(b).(\mathtt{run}\,b)$$

Similarly, in the breakdown function extended with recursion (cf. § 3.4) there is only one trio pattern, which can be transformed into a duo following the very same idea.

From Polyadic to Monadic Communication. Since we consider *closed* HO processes, we can dispense with polyadic communication in the breakdown function. We can define a *monadic decomposition*, $\mathsf{D}(P)$, that simplifies Def. 3.8 as follows:

$$\mathsf{D}(P) = (\nu\,\widetilde{c})\big(c_k?(b).(\mathtt{run}\,b) \mid \mathsf{B}^k\big(P\sigma\big)\big)$$

where $k > 0$, $\widetilde{c} = (c_k, \ldots, c_{k+|P|-1})$, and σ is as in Def. 3.8. Process $c_k?(b).(\mathtt{run}\,b)$ activates a thunk received from $\mathsf{B}^k\big(\,\cdot\,\big)$, the *monadic* breakdown function that simplifies the one in Tab. 1 by using only one parameter, namely k:

$$\begin{aligned}
\mathsf{B}^k\big(u_i?(x).Q\big) &= (\nu\,c_x)\big(\overline{c_k}!\big\langle\{\!\{u_i?(x).c_{k+1}?(b).(c_x!\langle x\rangle \mid (\mathtt{run}\,b))\}\!\}\big\rangle \mid \mathsf{B}^{k+1}\big(Q\sigma\big)\big)\\
\mathsf{B}^k\big(u_i!\langle x\rangle.Q\big) &= \overline{c_k}!\big\langle\{\!\{c_x?(x).u_i!\langle x\rangle.c_{k+1}?(b).(\mathtt{run}\,b)\}\!\}\big\rangle \mid \mathsf{B}^{k+1}\big(Q\sigma\big)\\
\mathsf{B}^k\big(u_i!\langle V\rangle.Q\big) &= \overline{c_k}!\big\langle\{\!\{u_i!\langle \mathsf{V}^{k+1}\big(V\sigma\big)\rangle.c_{k+1}?(b).(\mathtt{run}\,b)\}\!\}\big\rangle \mid \mathsf{B}^{k+1}\big(Q\sigma\big)\\
\mathsf{B}^k\big(x\,u\big) &= \overline{c_k}!\big\langle\{\!\{c_x?(x).(x\,\widetilde{u})\}\!\}\big\rangle\\
\mathsf{B}^k\big(V\,u\big) &= \overline{c_k}!\big\langle\{\!\{\mathsf{V}^{k+1}\big(V\big)\,\widetilde{u})\}\!\}\big\rangle\\
\mathsf{B}^k\big((\nu\,s)P'\big) &= (\nu\,\widetilde{s})\mathsf{B}^k\big(P'\sigma\big)\\
\mathsf{B}^k\big(Q \mid R\big) &= \overline{c_k}!\langle\{\!\{c_{k+1}?(b).\mathtt{run}\,b \mid c_{k+|Q|+1}?(b).\mathtt{run}\,b\}\!\}\rangle \mid \mathsf{B}^{k+1}\big(Q\big) \mid \mathsf{B}^{k+|Q|+1}\big(R\big)
\end{aligned}$$

Above, σ is as in Tab. 1. $\mathsf{B}^k\big(\,\cdot\,\big)$ propagates values using thunks and a dedicated propagator c_x for each variable x. We describe only the definition of $\mathsf{B}^k\big(u_i?(x).Q\big)$: it illustrates key ideas common to all other cases. It consists of an output of a thunk on c_k composed in parallel with $\mathsf{B}^{k+1}\big(Q\sigma\big)$. The thunk will be activated by a process $c_k?(b).(\mathtt{run}\,b)$ at the top-level; this activation triggers the input action on u_i, and prepares the activation for the next thunk (exchanged on name c_{k+1}). Upon reception, such a thunk is activated in

parallel with $c_x!\langle x\rangle$, which propagates the value received on u_i. The scope of c_x is restricted to include input actions on c_x in $\mathsf{B}^{k+1}\big(Q\sigma\big)$; such actions are the first in the thunks present in, e.g., $\mathsf{B}^{k}\big(u_i!\langle x\rangle.Q\big)$. We also need to revise the breakdown function for values $\mathcal{V}^{k}_{\tilde{x}}\big(\,\cdot\,\big)$. The breakdown functions given in § 3.3 and § 3.4 (cf. Tables 3 and 4) can be made monadic following similar lines.

These two optimizations can be combined by transforming the trios of the monadic breakdown into duos, following the key idea of the first optimization (cf. (4)).

5 Related Work

Our developments are related to results by Parrow [24], who showed that every process in the untyped, summation-free π-calculus with replication is weakly bisimilar to its decomposition into trios processes (i.e., $P \approx \mathcal{D}(P)$). We draw inspiration from insights developed in [24], but pursuing different goals in a different technical setting: our decomposition treats processes from a calculus without name-passing but with higher-order concurrency (abstraction-passing), supports labeled choices, and accommodates recursive types. Our goals are different than those in [24] because trios processes are relevant to our work in that they allow us to formally justify minimal session types; however, they are not an end in themselves. Still, we opted to retain the definitional style and terminology for trios from [24], which are elegant and clear.

Our main result connects the typability of a process with that of its decomposition; this is a *static guarantee*. Based on our examples, we conjecture that the *behavioral guarantee* given by $P \approx \mathcal{D}(P)$ in [24] holds in our setting too, under an appropriate *typed* weak bisimilarity. An obstacle here is that known notions of typed bisimilarity for session-typed processes, such as those given by Kouzapas et al. [20], are not adequate: they only relate processes typed under the *same* typing environments. We need a relaxed equivalence that (i) relates processes typable under different environments (e.g., Δ and $\mathcal{G}(\Delta)$) and (ii) admits that actions along s from P can be matched by $\mathcal{D}(P)$ using actions along s_k, for some k (and viceversa). Defining this notion precisely and studying its properties goes beyond the scope of this paper.

Our approach is broadly related to works that relate session types with other type systems for the π-calculus (cf. [17, 3, 4, 5, 10]). Kobayashi [17] encoded a finite session π-calculus into a π-calculus with linear types with usages (without sequencing); this encoding uses a continuation-passing style to codify a session name using multiple linear channels. Dardha et al. [3, 4] formalize and extend Kobayashi's approach. They use two separate encodings, one for processes and one for types. The former uses a freshly generated linear name to mimic each session action; this fresh name becomes an additional argument in communications. Polyadicity is thus an essential ingredient in [3, 4], whereas in our work it is convenient but not indispensable (cf. § 4). The encoding of types in [3, 4] codifies sequencing in session types by nesting payload types. In contrast, we "slice" the n actions occurring in a session s along indexed names $s_1, \ldots, s_n$ with minimal session types, i.e., slices of the type for s. All in all, an approach based on minimal session types appears simpler than that in [3, 4]. Works by Padovani [23] and Scalas et al. [25] is also related: they rely on [3, 4] to develop verification techniques based on session types for OCaml and Scala programs, respectively.

Gay et al. [10] formalize how to encode a monadic π-calculus, equipped with a finite variant of the binary session types of [9], into a polyadic π-calculus with an instance of the generic process types of [16]. The work of Demangeon and Honda [5] encodes a session π-calculus into a linear/affine π-calculus with subtyping based on choice and selection types. Our developments differ from these previous works in an important respect: we relate two

formulations of session types, namely standard session types and minimal session types. Indeed, while [17, 3, 4, 5, 10] target the *relative expressiveness* of session-typed process languages, our work emerges as the first study of *absolute expressiveness* in this context.

Finally, we elaborate further on our choice of HO as source language. HO is a sub-calculus of HOπ, whose basic theory and expressivity were studied by Kouzapas et al. [19, 20] as a hierarchy of session-typed calculi based on relative expressiveness. Our developments enable us to include HO with minimal session types within this hierarchy. Still, our approach does not rely on having HO as source language, and can be adapted to other typed frameworks based on session types, such as the type discipline for first-order π-calculus processes in [26].

6 Concluding Remarks

Session types are a class of *behavioral types* for message-passing programs. We presented a *decomposition* of session-typed processes in [19, 20] using *minimal* session types, in which there is no sequencing. The decomposition of a process P, denoted $\mathcal{D}(P)$, is a collection of *trios processes* that trigger each other mimicking its sequencing. We prove that typability of P using standard session types implies the typability of $\mathcal{D}(P)$ with minimal session types. Our results hold for all session types constructs, including labeled choices and recursive types.

Our contributions can be interpreted in three ways. *First*, from a foundational standpoint, our study of minimal session types is a conceptual contribution to the theory of behavioral types, in that we precisely identify sequencing as a source of redundancy in all preceding session types theories. As remarked in § 1, there are many session types variants, and their expressivity often comes at the price of an involved underlying theory. Our work contributes in the opposite direction, as we identified a simple yet expressive fragment of an already minimal session-typed framework [19, 20], which allows us to justify session types in terms of themselves. Understanding further the underlying theory of minimal session types (e.g., notions such as type-based compatibility) is an exciting direction for future work.

Second, our work can be seen as a new twist on Parrow's decomposition results in the *untyped* setting [24]. While Parrow's work indeed does not consider types, in fairness we must observe that when Parrow's work appeared (1996) the study of types for the π-calculus was rather incipient (for instance, binary session types appeared in 1998 [12]). That said, we should stress that our results are not merely an extension of Parrow's with session types, for types in our setting drastically narrow down the range of conceivable decompositions. Also, we exploit features not supported in [24], most notably higher-order concurrency.

Last but not least, from a practical standpoint, we believe that our approach paves a new avenue to the integration of session types in programming languages whose type systems lack sequencing, such as Go. It is natural to envision program analysis tools which, given a message-passing program that should conform to protocols specified as session types, exploit our decomposition as an intermediate step in the verification of communication correctness. Remarkably, our decomposition lends itself naturally to an implementation – in fact, we generated our examples automatically using MISTY, an associated artifact written in Haskell.

References

1 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral Types in Programming Languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016. doi:10.1561/2500000031.

2 Stephanie Balzer and Frank Pfenning. Objects as session-typed processes. In Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos Varela, editors, *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015, Pittsburgh, PA, USA, October 26, 2015*, pages 13–24. ACM, 2015. doi:10.1145/2824815.2824817.

3 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *Proc. of PPDP 2012*, pages 139–150. ACM, 2012. doi:10.1145/2370776.2370794.

4 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Inf. Comput.*, 256:253–286, 2017. doi:10.1016/j.ic.2017.06.002.

5 Romain Demangeon and Kohei Honda. Full Abstraction in a Subtyped pi-Calculus with Linear Types. In *Proc. of CONCUR 2011*, volume 6901 of *LNCS*, pages 280–296. Springer, 2011. doi:10.1007/978-3-642-23217-6_19.

6 Mariangiola Dezani-Ciancaglini and Ugo de' Liguoro. Sessions and Session Types: an Overview. In *WS-FM'09*, volume 6194 of *LNCS*, pages 1–28. Springer, 2010. URL: http://www.di.unito.it/~dezani/papers/sto.pdf.

7 Mariangiola Dezani-Ciancaglini, Elena Giachino, Sophia Drossopoulou, and Nobuko Yoshida. Bounded Session Types for Object Oriented Languages. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*, volume 4709 of *Lecture Notes in Computer Science*, pages 207–245. Springer, 2006. doi:10.1007/978-3-540-74792-5_10.

8 Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session Types for Object-Oriented Languages. In Dave Thomas, editor, *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, volume 4067 of *Lecture Notes in Computer Science*, pages 328–352. Springer, 2006. doi:10.1007/11785477_20.

9 Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42:191–225, 2005. doi:10.1007/s00236-005-0177-z.

10 Simon J. Gay, Nils Gesbert, and António Ravara. Session Types as Generic Process Types. In Johannes Borgström and Silvia Crafa, editors, *Proceedings Combined 21st International Workshop on Expressiveness in Concurrency and 11th Workshop on Structural Operational Semantics, EXPRESS 2014, and 11th Workshop on Structural Operational Semantics, SOS 2014, Rome, Italy, 1st September 2014.*, volume 160 of *EPTCS*, pages 94–110, 2014. doi:10.4204/EPTCS.160.9.

11 Simon J. Gay, Vasco Thudichum Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 299–312. ACM, 2010. doi:10.1145/1706299.1706335.

12 Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

13 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.

14 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In Jan Vitek, editor, *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, volume 5142 of *Lecture Notes in Computer Science*, pages 516–541. Springer, 2008. doi:10.1007/978-3-540-70592-5_22.

15 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.*, 49(1):3, 2016. doi:10.1145/2873052.

16 Atsushi Igarashi and Naoki Kobayashi. A generic type system for the Pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004. doi:10.1016/S0304-3975(03)00325-6.

17 Naoki Kobayashi. Type Systems for Concurrent Programs. In *Formal Methods at the Crossroads*, volume 2757 of *LNCS*, pages 439–453. Springer, 2003. doi:10.1007/978-3-540-40007-3_26.

18 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and StMungo. In James Cheney and Germán Vidal, editors, *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, pages 146–159. ACM, 2016. doi:10.1145/2967973.2968595.

19 Dimitrios Kouzapas, Jorge A. Pérez, and Nobuko Yoshida. On the Relative Expressiveness of Higher-Order Session Processes. In Peter Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 446–475. Springer, 2016. Extended version to appear in Information and Computation (Elsevier). doi:10.1007/978-3-662-49498-1_18.

20 Dimitrios Kouzapas, Jorge A. Pérez, and Nobuko Yoshida. Characteristic bisimulation for higher-order session processes. *Acta Inf.*, 54(3):271–341, 2017. doi:10.1007/s00236-016-0289-7.

21 Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A static verification framework for message passing in Go using behavioural types. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1137–1148. ACM, 2018. doi:10.1145/3180155.3180157.

22 Nicholas Ng and Nobuko Yoshida. Static deadlock detection for concurrent go by global session graph synthesis. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 174–184. ACM, 2016. doi:10.1145/2892208.2892232.

23 Luca Padovani. A Simple Library Implementation of Binary Sessions. *Journal of Functional Programming*, 27, 2017. doi:10.1017/S0956796816000289.

24 Joachim Parrow. Trios in concert. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 623–638. The MIT Press, 2000. Online version, dated July 22, 1996, available at http://user.it.uu.se/~joachim/trios.pdf.

25 Alceste Scalas and Nobuko Yoshida. Lightweight Session Programming in Scala. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 21:1–21:28. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.21.

26 Vasco T. Vasconcelos. Fundamentals of session types. *Inf. Comput.*, 217:52–70, 2012. doi:10.1016/j.ic.2012.05.002.