Message-Observing Sessions

RYAN KAVANAGH, Université du Québec à Montréal, Canada
BRIGITTE PIENTKA, McGill University, Canada

We present Most, a process language with message-observing session types. Message-observing session types extend binary session types with type-level computation to specify communication protocols that vary based on messages observed on other channels. Hence, Most allows us to express global invariants about processes, rather than just local invariants, in a bottom-up, compositional way. We give Most a semantic foundation using *traces with binding*, a semantic approach for compositionally reasoning about traces in the presence of name generation. We use this semantics to prove type soundness and compositionality for Most processes. We see this as a significant step towards capturing message-dependencies and providing more precise guarantees about processes.

CCS Concepts: • **Theory of computation** → **Program specifications**; *Type structures*; Denotational semantics; • **Computing methodologies** → *Concurrent programming languages.*

Additional Key Words and Phrases: session types, program specification, trace semantics, dependent types

ACM Reference Format:
Ryan Kavanagh and Brigitte Pientka. 2024. Message-Observing Sessions. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 142 (April 2024), 29 pages. https://doi.org/10.1145/3649859

1 INTRODUCTION

Session types [Honda 1993; Honda, Vasconcelos, et al. 1998; Takeuchi et al. 1994] allow us to specify and statically verify that processes communicate according to prescribed protocols. Hence, they rule out a wide class of communication-related bugs before executing a given protocol.

A *binary* session type specifies the communication protocol as seen from the point of view of one of the two participants. From a Curry-Howard perspective, it corresponds to the standard sequent calculus proof system for dual intuitionistic linear logic (see [Caires and Pfenning 2010; Caires, Pfenning, and Toninho 2016]), thereby building a logical foundation for specifying and reasoning about concurrent communications. Over the past decade, there have been different approaches to extend it and capture ever richer protocols: value-dependent session types [Toninho, Caires, et al. 2011; Toninho and Yoshida 2018] allow protocols to vary based on previously transmitted values; label-dependent session types [Thiemann and Vasconcelos 2019] describe different communication behaviour depending on labels being observed on a given channel; manifest sharing [Balzer and Pfenning 2017] allows (binary) session types to capture shared communication channels.

In this paper, we describe Most, a language for protocols using *message-observing session types.* A message-observing session type specifies how communication evolves taking into account messages that can be observed on other channels in a process's environment. This is in contrast to both value and label-dependent session types, where a channel's type can only depend on messages that the

Authors' addresses: Ryan Kavanagh, Département d'informatique, Université du Québec à Montréal, 201 Du Président-Kennedy Avenue, Montréal, QC, H2X 3Y7, Canada, kavanagh.ryan@uqam.ca; Brigitte Pientka, School of Computer Science, McGill University, 3480 University Street, Montréal, QC, H3A 0E9, Canada, bpientka@cs.mcgill.ca.

ACM 2475-1421/2024/4-ART142
https://doi.org/10.1145/3649859

channel previously carried, but not on messages on other channels in the process's environment. Hence, Most allows us to express global invariants about processes rather than just local invariants.

To specify session types that vary based on messages on other channels, we extend binary session types with *type-level processes*. This allows us to capture a wider range of correctness guarantees than is currently possible. To illustrate, we take a closer look at the identity process id that provides an implementation for the session type $A \coloneqq \oplus\{\, \mathsf{left} : \mathbf{1}, \mathsf{right} : \mathbf{1} \,\}$. This session type specifies that communication is a message containing either label left or right, followed by a message signalling the end of communication. We can implement id between channels a and b of type A as follows:

$$\mathsf{case}\ a\ \{\, \mathsf{left} \Rightarrow b.\mathsf{left};\ \mathsf{wait}\ a;\ \mathsf{close}\ b \mid \mathsf{right} \Rightarrow b.\mathsf{right};\ \mathsf{wait}\ a;\ \mathsf{close}\ b \,\}.$$

Operationally, this process waits to receive a label on a, selects the corresponding branch, and sends the label over b. It then waits for the channel a to close before closing b. However, the session type for channel a and channel b does not capture this precisely. In particular, the type of b does not rule out the following erroneous implementation that swaps labels:

$$\mathsf{case}\ a\ \{\, \mathsf{left} \Rightarrow b.\mathsf{right};\ \mathsf{wait}\ a;\ \mathsf{close}\ b \mid \mathsf{right} \Rightarrow b.\mathsf{left};\ \mathsf{wait}\ a;\ \mathsf{close}\ b \,\}$$

The crux of the issue is that the messages we wish to allow on b depend on the messages that we observe on a. In Most, we specify such a protocol for b using type-level processes as follows:

$$b : \mathsf{CASE}\ a\ \{\, \mathsf{left} \Rightarrow \oplus\{\, \mathsf{left} : \mathbf{1} \,\} \mid \mathsf{right} \Rightarrow \oplus\{\, \mathsf{right} : \mathbf{1} \,\} \,\}.$$

Intuitively, it specifies that the type of b is the unary choice $\oplus\{\, \mathsf{left} : \mathbf{1} \,\}$ if a label left is observed on a, and symmetrically if right is observed on a. A message-observing session type is necessarily open: it names the channel whose messages it observes. This is analogous to processes, which contain the channel names on which they communicate. The particular channel a observed by the type of b is determined by the process specification in which it appears. For example, the following concrete syntax specifies id by specifying the types of its channels, and it specifies that b's type observes a channel $a : A$ used by id:

$$\mathsf{id} :: \{\,\}\ [a : A]\ (b : \mathsf{CASE}\ a\ \{\, \mathsf{left} \Rightarrow \oplus\{\, \mathsf{left} : \mathbf{1} \,\} \mid \mathsf{right} \Rightarrow \oplus\{\, \mathsf{right} : \mathbf{1} \,\} \,\}) = \mathsf{case}\ a\ \{ \cdots \}.$$

We will revisit this concrete syntax in section 2.

Most provides a "bottom-up" approach to specifying processes and their composition. We still specify the communication protocol from the point of view of one of the two participants, but take into account the messages observed on other channels in the environment. An alternative approach for specifying multi-channel communication patterns is *multiparty* session types. Multiparty session types [Honda, Yoshida, et al. 2016] specify interactions between a static number of participants using a *global type* and project these specifications onto individual participants for typechecking. Despite recent work [Deniélou and Yoshida 2011; Stolze et al. 2023], multiparty session types are an inherently closed world or "top-down" approach, where the entire system must be designed before it can be implemented. In contrast, we can compose protocols in Most from the bottom-up, while still being able to capture some of the richer interactions of multiparty session types.

Concretely, we make the following contributions:

(1) We introduce a **message-observing binary session type system**, called Most. A message-observing session type specifies how communication evolves taking into account communications that can be observed on other channels in our environment. This is achieved using **type-level processes** in session types. These type-level processes further restrict the eligible processes and their actions. We motivate this extension through a sequence of examples.
(2) We give processes and their specifications a **semantics using *traces with binding***, a semantic foundation inspired by nominal sequences [Gabbay and Ghica 2012]. A frequent challenge when we define a trace semantics or ordered semantics for concurrency is that

we must account for fresh channel name generation and propagate names involved in higher-order communications. Further, compositionality often requires infinite trace sets or infinite collections of semantic objects, which complicates implementation and obscures the computational content. Instead, we use name binding to quantify over all fresh channel in a manner reminiscent of higher-order abstract syntax, resulting in a more compact presentation. This abstraction allows us to retain compositionality.

(3) We give a **semantically sound typechecking algorithm**. Process denotations characterize all possible communication behaviours, while specifications specify the processes traces they allow subject to constraints on ambient communications. Soundness ensures that well-typed processes only exhibit behaviours (traces) permitted by their specifications.

We see this work as a significant step towards providing more precise guarantees about processes.

Omitted typing rules and proofs are available in the preprint [Kavanagh and Pientka 2024].

2 MOTIVATION

We introduce the main ideas of Most through a sequence of examples. This will allow us to showcase both the power of message-observing session types and the design decisions.

2.1 A Quick Guide to Session Types and Specifying Processes

Session types $A, B, \ldots$ specify the communication protocols on named channels $a, b, \ldots$. Processes in our system are organized according to a client-server architecture. This architecture is reflected in our concrete syntax for process specifications:

$$\mathsf{proc} :: \{a_1 : A_1, \ldots, a_n : A_n\}\ [b_1 : B_1, \ldots, b_n : B_n]\ (c_0 : C_0)$$

It states that the process proc is a **server** for (or **provides**) a distinguished service C_0 over a channel c_0. Dually, it is a **client** of (or **uses**) zero or more services $B_1, \ldots, B_n$ on channels named $b_1, \ldots, b_n$, respectively. Finally, its types may refer to zero or more ambient communication channels $a_i : A_i$. Ambient channels are instantiated either by composing proc with processes that use or provide them, or by defining proc as a composition of processes that communicate on these channels. Ambient channels will permit compositional process specifications. To keep our subsequent examples simple, most will feature empty internal and ambient contexts. Each type can refer other channel names in the specification, allowing for mutual dependency between types.

For our first examples, we consider sending a sequence of n bits on a channel a. This is accomplished by sending a sequence of n labels drawn from set $\mathsf{Bit} = \{\, \mathtt{0}, \mathtt{1} \,\}$. Such a sequence is specified by the session type Bits_n defined by induction on n using the following pair of equations:[1]

$$\mathsf{Bits}_0 = \mathbf{1} \qquad\qquad \mathsf{Bits}_{n+1} = \oplus\{\, l : \mathsf{Bits}_n \,\}_{l \in \mathsf{Bit}}.$$

The empty sequence of bits is captured by the unit type $\mathbf{1}$: it signals the end of communication on the channel. To define a bit sequence of length $n + 1$, we specify the possible labels (elements) that can be transmitted followed by the sequence of length n. To accomplish this, we use the internal choice type $\oplus\{\, l : \mathsf{Bits}_n \,\}_{l \in \mathsf{Bit}}$.[2] Concretely, we can use the definition to generate the type for bit sequences of length 2 as follows:

$$\mathsf{Bits}_2 = \oplus\{\, \mathtt{0} : \oplus\{\, \mathtt{0} : \mathbf{1}, \mathtt{1} : \mathbf{1} \,\}, \mathtt{1} : \oplus\{\, \mathtt{0} : \mathbf{1}, \mathtt{1} : \mathbf{1} \,\} \,\}$$

It specifies a sequence of two bits, i.e., a sequence of two labels, $\mathtt{0}$ and $\mathtt{1}$ followed by termination.

[1] Most omits language-level recursion and relies on meta-level induction to define inductive types. This simplifies Most's presentation and avoids obscuring its key contributions. We nevertheless sketch how to extend Most with recursive session types and processes in section 8.4.

[2] An internal choice $\oplus\{\, l : A_l \,\}_{l \in L}$ specifies that a label $l \in L$ will be transmitted, and then communication will satisfy type A_l.

We can then specify a family of identity processes, id_n, that use a channel $a : \mathsf{Bits}_n$ and provide an output channel b of the same type by the following pair of equations:

$$\begin{aligned}
\mathsf{id}_0 &:: \{\,\}\ [a : \mathsf{Bits}_0\ ]\ (b : \mathsf{Bits}_0\ ) = \mathsf{wait}\ a;\ \mathsf{close}\ b\\
\mathsf{id}_{n+1} &:: \{\,\}\ [a : \mathsf{Bits}_{n+1}]\ (b : \mathsf{Bits}_{n+1}) = \mathsf{case}\ a\ \{\, l \Rightarrow b.l;\ \mathsf{id}_n\,\}_{l \in \mathsf{Bit}}
\end{aligned}$$

In the base case, id_0, we wait to receive a close message on a that signals the end of communication, and we immediately send it on b before terminating. In the recursive step, id_{n+1}, we perform a case analysis on the label received over a. If we receive a label l, then we take the corresponding branch and send the label l over b. At this point, a and b both have type Bits_n, so we continue as id_n.

2.2 Specifying Identity Processes for Real Using Message-Observing Session Types

The session type specification of the identity process id_n in the previous section fails to ensure that we copy the sequence of bits unchanged from a to b. In Most, we can give a more precise type to the output channel by taking into account the message that we already have received on a given channel using message-observing sessions. In particular, we can refine the type of b to the following more precise family of types $\mathsf{IdBitSeq}_n\ a$ that observe messages on channel a. We give the definition of the identity process next to it, to highlight the close correspondence between the process and the message-observing session type:

Process	Message-Observing Session Type
$\mathsf{id}_0 = \mathsf{wait}\ a;\ \mathsf{close}\ b$	$\mathsf{IdBitSeq}_0\ \ a = \mathsf{CASE}\ a\ \{\, \mathsf{close} \Rightarrow \mathbf{1}\,\}$
$\mathsf{id}_{n+1} = \mathsf{case}\ a\ \{\, l \Rightarrow b.l;\ \mathsf{id}_n\,\}_{l\in\mathsf{Bit}}$	$\mathsf{IdBitSeq}_{n+1}\ a = \mathsf{CASE}\ a\ \{\, l \Rightarrow \oplus\{\, l : \mathsf{IdBitSeq}_n\ a\,\}\,\}_{l\in\mathsf{Bit}}$

In the base case, the type $\mathsf{CASE}\ a\ \{\,\mathsf{close} \Rightarrow \mathbf{1}\,\}$ specifies that we can only signal termination on b after we have observed termination on a. In the recursive step, $\mathsf{IdBitSeq}_{n+1}\ a$ performs a case analysis on the label sent on a. If a label l is sent or received on a, then the type reduces to the unary internal choice $\oplus\{\, l : \mathsf{IdBitSeq}_n\ a\,\}$ that only allows the label l to be sent, and continues as $\mathsf{IdBitSeq}_n\ a$. We can then refine the specification of id to use this more precise protocol:

$$\begin{aligned}
\mathsf{id}_0 &:: \{\}\ [a : \mathsf{Bits}_0\ ]\ (b : \mathsf{IdBitSeq}_0\ \ a) = \mathsf{wait}\ a;\ \mathsf{close}\ b\\
\mathsf{id}_{n+1} &:: \{\}\ [a : \mathsf{Bits}_{n+1}]\ (b : \mathsf{IdBitSeq}_{n+1}\ a) = \mathsf{case}\ a\ \{\, l \Rightarrow b.l;\ \mathsf{id}_n\,\}_{l\in\mathsf{Bit}}
\end{aligned}$$

Note that the process and the type of the used channel a remain the same as in the previous section: the only change is to the type of the channel b, which now observes messages on a. At the high-level, the type $\mathsf{IdBitSeq}_n\ a$ ensures that id_n can only output on b what it receives on a.

We call the type $\mathsf{IdBitSeq}_n\ a$ **message observing** because it observes communications on a to determine the range of permitted communications. It can be thought of as a type-level process, as it mimics choices and behaviour specified in the process id_n on the type-level. This type reduces as messages are observed on the channel a. These observations and reductions exist only in the typechecking algorithm (types are not present at runtime), which tracks messages sent or received by the specified process and reduces types accordingly. For example, to type check id_{n+1} against its specification, we check each of its branches "$b.l;\ \mathsf{id}_n$" (to which it steps after receiving a label l) against the specification "$\{\,\}\ [a : \mathsf{Bits}_n]\ (b : \oplus\{\, l : \mathsf{IdBitSeq}_n\ a\,\})$", which is obtained by reducing all types by the label l received on a.

This example illustrates two key points in Most: (1) message-observing session types can express dependencies on messages on other channels, and (2) message-observing types behave as type-level processes and incorporate a notion of concurrent computation on the type-level.

On a practical level, this allows us to express more precise safety guarantees that cannot be expressed by prior work. On a more theoretical level, we see message-observing session types as an essential step towards a type theory of session types.

2.3 Expressing Mutual Observation in Message-Observing Session Types

Session-typed languages often feature a higher-order session type $A \otimes B$, which specifies a server that sends a channel of type A (call it a) and then communicates according to B.

It is useful in applications for B to observe messages on a. To do so, we introduce the syntax $(a : A) \otimes B$, which binds a name a for the transmitted channel of type A in B. This bound name is eventually instantiated by the actual name of the transmitted channel. This syntax then lets us specify, e.g., sending a pair of identical bit streams: $(a : \mathsf{Bits}_{\,n}) \otimes \big(\mathsf{IdBitSeq}_n\ a\big)$. A server of this type sends a channel a carrying n bits and then communicating according to the protocol $\mathsf{IdBitSeq}_n\ a$, which specifies transmitting a bit stream equal to the one carried by a.

Communication on both channels occurs independently, so it is natural to ask whether A can also observe messages on the channel of type B (call it b). From the dependent-types perspective in the sequential setting, such mutual dependency may seem unusual, but it captures the concurrent behaviour where the processes providing A and B run concurrently. To allow for mutual observation, we use the syntax $(a : A) \otimes (b : B)$ for higher-order sessions, where name a is bound in B and name b is bound in A. It allows the type B to observe messages on the transmitted channel $a : A$, while also allowing the type A to observe messages on the channel $b : B$. In practice, b will always be instantiated with the name of the channel of type $(a : A) \otimes (b : B)$; we treat it as bound in the type only to ensure that the type makes sense independently of the channel that it types. As a result, we can specify mutually observing higher-order protocols.

To illustrate mutual observation, we consider a load-balanced list service. This service provides a pair of lists, and then forces the client to alternate between lists when receiving elements. It is given by $(a : \mathsf{LBList}_n\ b) \otimes (b : \mathsf{LBList}_n\ a)$, where $\mathsf{LBList}_n\ c$ is inductively defined by:

$$\mathsf{LBList}_0\ c = \mathbf{1} \qquad\qquad \mathsf{LBList}_{n+1}\ c = \oplus\{\, x : \mathsf{CASE}\ c\ \{\, y \Rightarrow \mathsf{LBList}_n\ c\,\}_{y \in \mathsf{Bit}}\,\}_{x \in \mathsf{Bit}}.$$

Intuitively, a channel $a : \mathsf{LBList}_{n+1}\ c$ provides a list element $x \in \mathsf{Bit}$, and then requires the client to observe some element y on c before it can observe another list element from a.

To see how the protocol $(a : \mathsf{LBList}_{n+1}\ b) \otimes (b : \mathsf{LBList}_{n+1}\ a)$ enforces alternation, consider one of its clients. After receiving the channel provided by this service, the client has two channels: channels $a : \mathsf{LBList}_{n+1}\ b$ and $b : \mathsf{LBList}_{n+1}\ a$. Assume, without loss of generality, that it first receives a label on a. The internal choice type specifies that communication on a then satisfies $\mathsf{CASE}\ b\ \{\, y \Rightarrow \mathsf{LBList}_n\ b\,\}_{y \in \mathsf{Bit}}$. In particular, the client cannot communicate on a until it reduces the CASE analysis. To do so, the client must receive a label on b. It can do so, for the type of b is an internal choice $\oplus\{\, x : \mathsf{LBList}_n\ a\,\}_{x \in \mathsf{Bit}}$ (it was reduced by the label received on a). After receiving a label on b, the type of a reduces to $\mathsf{LBList}_n\ b$, while the type of b becomes $\mathsf{LBList}_n\ a$.

For convenience, we write $A \otimes (b : B)$ for $(a : A) \otimes (b : B)$ when a does not appear in B. The abbreviations $(a : A) \otimes B$ and $A \otimes B$ are analogous.

2.4 Depending on Higher-Order Sessions

Protocols can depend on higher-order sessions using an elimination form reminiscent of positive product elimination in functional languages. We illustrate this by specifying an AND logic gate, given by the process AND:

$$\begin{aligned}
&\mathsf{AND} :: \{\,\}\ [b : \mathsf{Bits}_{\,1} \otimes \mathsf{Bits}_{\,1}]\ (c : \mathsf{Bits}_{\,1}) = \\
&\qquad a \leftarrow \mathsf{receive}\ b;\ \mathsf{case}\ a\ \{\, 1 \Rightarrow \mathsf{case}\ b\ \{\, 1 \Rightarrow c.1;\ \mathsf{end} \mid \mathtt{0} \Rightarrow c.\mathtt{0};\ \mathsf{end}\,\} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mid \mathtt{0} \Rightarrow \mathsf{case}\ b\ \{\, x \Rightarrow c.0;\ \mathsf{end}\,\}_{x \in \mathsf{Bit}}\,\}
\end{aligned}$$

where $\mathsf{end} :: \{\,\}\ [a : \mathbf{1}, b : \mathbf{1}]\ (g : \mathbf{1}) = (\mathsf{wait}\ a;\ \mathsf{wait}\ b;\ \mathsf{close}\ g)$. From its specification, we see that AND receives a pair of bits on b and sends a bit on c. Its implementation starts by receiving a channel of type Bits_1 over b and binding it to the name a (syntax: "$a \leftarrow \text{receive } b;\ \cdots$"). At this point, the process is a client of $a : \mathsf{Bits}_1$ and $b : \mathsf{Bits}_1$ and provides $c : \mathsf{Bits}_1$. The process then observes the bits on a and b, sends their AND over c, and terminates.

Observe that the above specification does not specify that AND correctly implements an AND gate. However, we can use message observation to specify this functionality at the type level. To do so, we refine the above specification to

$$\mathsf{AND} :: \{\,\}\ [b : \mathsf{Bits}_1 \otimes \mathsf{Bits}_1]\ (c : \mathsf{andOf}\ b) = \cdots$$

where the following type specifies an AND gate:

$$\begin{aligned}\mathsf{andOf}\ b = \mathsf{CASE}\ b\ \{\,\langle a\rangle \Rightarrow \mathsf{CASE}\ a\ \{\,1 \Rightarrow \mathsf{CASE}\ b\ \{\,1 \Rightarrow \oplus\{\,1 : \mathbf{1}\,\} \mid 0 \Rightarrow \oplus\{\,0 : \mathbf{1}\,\}\,\}\\ \mid 0 \Rightarrow \oplus\{\,0 : \mathbf{1}\,\}\,\}\,\}.\end{aligned}$$

This type-level process uses the new construct $\mathsf{CASE}\ b\ \{\,\langle a\rangle \Rightarrow C\,\}$, which binds the name of a channel transmitted over b to a in C. This construct reduces to $[\alpha/a]C$ when a channel α is transmitted on b. This reduction closely mimics the behaviour of the channel-receiving process construct, $a \leftarrow \text{receive } b;\ P$, which becomes $[\alpha/a]P$ after receiving α on b. In the case of "andOf b", it reduces to the internal choice $\oplus\{\,1 : \mathbf{1}\,\}$ if the transmitted channel and its carrier both carry a 1 bit, and otherwise it reduces to $\oplus\{\,0 : \mathbf{1}\,\}$.

2.5 A Fair Auction

In our examples so far, there has been a close correspondence between processes and the message-observing types in their specifications. In fact, this correspondence is partially by design as we take into account the process communication in defining a message-observing type. The next example shows that this correspondence between processes and message-observing types is not always trivial, and that processes can be more complex than the types specifying their communications.

Assume two bidders participating in an auction, where each may bid 0 (described by the label `0`) or 1 (described by the label 1). We want to ensure that the bidder with the highest bid wins. With traditional session types, we can specify the protocols for a bidder service as follows, where the external choice $\&\{\cdots\}$ means that the bidder *receives* the label:

$$\begin{aligned}\mathsf{bidder} &= \oplus\{\,0 : \mathsf{result}, 1 : \mathsf{result}\,\}\\ \mathsf{result} &= \&\{\,\mathtt{lost} : \mathbf{1}, \mathtt{tie} : \mathbf{1}, \mathtt{win} : \mathbf{1}\,\}\end{aligned}$$

A bidder is a process that provides a channel b of type bidder; the name b can freely be renamed, provided that it is kept distinct from other names in the process and its specification:

$$\mathsf{Bidder} :: \{\,\}\ [\,]\ (b : \mathsf{bidder}) = \cdots$$

An Auctioneer is a client of two bidders and signals the end of the auction on its provided channel c. We can implement it as follows, where $\text{end} :: \{\,\}\ [b_1 : \mathbf{1}, b_2 : \mathbf{1}]\ (c : \mathbf{1}) = (\text{wait } b_1;\ \text{wait } b_2;\ \text{close } c)$:

$$\begin{aligned}&\mathsf{Auctioneer} :: \{\}\ [b_1 : \mathsf{bidder}, b_2 : \mathsf{bidder}]\ (c : \mathbf{1}) =\\ &\qquad \text{case } b_1\ \{\,0 \Rightarrow \text{case } b_2\ \{\,0 \Rightarrow b_1.\mathtt{tie};\ b_2.\mathtt{tie};\ \text{end}\\ &\qquad\qquad\qquad\qquad\quad \mid 1 \Rightarrow b_1.\mathtt{lost};\ b_2.\mathtt{win};\ \text{end}\,\}\\ &\qquad\quad \mid 1 \Rightarrow \text{case } b_2\ \{\,0 \Rightarrow b_1.\mathtt{win};\ b_2.\mathtt{lost};\ \text{end}\\ &\qquad\qquad\qquad\qquad\quad \mid 1 \Rightarrow b_1.\mathtt{tie};\ b_2.\mathtt{tie};\ \text{end}\,\}\,\}\end{aligned}$$

Observe that the specification does not rule out an unfair auctioneer that privileges bidder b_1 and always give them the win, i.e., an auctioneer where each branch is $b_1.\mathtt{win};\ b_2.\mathtt{lost};\ \text{end}$.

To prohibit unfair auctioneers, we use a message-observing protocol that specifies the interactions for one bidder in terms of the actions of its opponent o:

$$\begin{aligned}\mathsf{fairAuct}\ o = \oplus\{\,\mathtt{0} : \mathsf{CASE}\ o\ \{\,\mathtt{0} \Rightarrow \&\{\,\mathtt{tie} : \mathbf{1}\,\} \mid \mathtt{1} \Rightarrow \&\{\,\mathtt{lost} : \mathbf{1}\,\}\,\},\\ \mathtt{1} : \mathsf{CASE}\ o\ \{\,\mathtt{0} \Rightarrow \&\{\,\mathtt{win} : \mathbf{1}\,\} \mid \mathtt{1} \Rightarrow \&\{\,\mathtt{tie} : \mathbf{1}\,\}\,\}\,\}\end{aligned}$$

The following specification then ensures that Auctioneer is fair:

$$\mathsf{Auctioneer} :: \{\}\ [b_1 : \mathsf{fairAuct}\ b_2, b_2 : \mathsf{fairAuct}\ b_1]\ (c : \mathbf{1}) = \cdots.$$

Indeed, the fair auction protocol fairAuct b_2 specifying communications on b_1 ensures that if b_1 sends a label `0`, then the auctioneer can only send `tie` or `lost` to b_1, and then only after a label `0` or `1` has been observed on b_2, respectively. The case when b_1 bids 1 is analogous.

This example illustrates that specifications do not simply lift process implementations to the type level. In particular, processes can specify more complex interactions than their types. This illustration is even more striking when we consider the following implementation of a bidder. It bids 0, and then terminates after having received the result of the auction:

$$\begin{aligned}&\mathsf{BidsZero} :: \{b_2 : \mathsf{fairAuct}\ b_1\}\ []\ (b_1 : \mathsf{fairAuct}\ b_2) =\\ &\quad b_1.\mathtt{0};\ \mathsf{case}\ b_1\ \{\,\mathsf{result} \Rightarrow \mathsf{close}\ a\,\}_{\mathsf{result} \in \{\,\mathtt{lost},\mathtt{tie},\mathtt{win}\,\}}.\end{aligned}$$

This example illustrates how we track the ambient channel b_2 : fairAuct b_1 in the specification to ensure that the type of b_1 remains well-scoped. When typechecking BidsZero against its specification, our typechecking algorithm generates constraints on messages that can appear on b_2 that are then checked when BidsZero is composed with a process implementing b_2.

The ambient context helps ensure that we know the types of all channels observed by types in process interfaces. As a result, we can ensure that types are well-formed relative to each other. It also means that we can modularly specify processes: we specify the types of local channels, but these can observe channels used by other processes in our execution environment.

2.6 Process Composition

To illustrate how Most typechecks compositions, we consider the composition of two bit-flipping processes. A bit-flipping process is a client of a bit that provides its negation:

$$\mathsf{neg} :: \{\,\}\ [i : \mathsf{Bits}_1]\ (o : \mathsf{Bits}_1) = \mathsf{case}\ i\ \{\,\mathtt{0} \Rightarrow o.\mathtt{1};\ \mathsf{end} \mid \mathtt{1} \Rightarrow o.\mathtt{0};\ \mathsf{end}\,\}$$

where end :: $\{\,\}\ [i : \mathbf{1}]\ (o : \mathbf{1}) = (\mathsf{wait}\ i;\ \mathsf{close}\ o)$. We can give neg more precise specifications. For example, we can specify that

$$\begin{aligned}&\mathsf{neg} :: \{\,\}\ [i : \mathsf{Bits}_1]\ (o : \mathsf{negBit}\ i) = \cdots\\ \text{or}\quad &\mathsf{neg} :: \{c : \mathsf{Bits}_1\}\ [i : \mathsf{negBit}\ c]\ (o : \mathsf{IdBitSeq}_1\ c) = \cdots,\\ \text{where}\quad &\mathsf{negBit}\ a = \mathsf{CASE}\ a\ \{\,\mathtt{0} \Rightarrow \oplus\{\,\mathtt{1} : \mathbf{1}\,\} \mid \mathtt{1} \Rightarrow \oplus\{\,\mathtt{0} : \mathbf{1}\,\}\,\}.\end{aligned}$$

The first specification ensures that neg sends the negation of the bit it receives on i over o. The second is more interesting: it specifies that if neg receives on i the negation of a bit sent on some ambient channel c, then it outputs that original bit on o.

Using these two specifications, we can ensure that flipping a bit twice behaves as the identity. To do so, we compose an implementation of neg with itself:

$$\mathsf{doubleNeg} :: \{c : \mathsf{negBit}\ i\}\ [i : \mathsf{Bits}_1]\ (o : \mathsf{IdBitSeq}_1\ i) = (i \leftrightarrow \mathsf{neg} \leftrightarrow c) \parallel_c (c \leftrightarrow \mathsf{neg} \leftrightarrow o).$$

The syntax $(i \leftrightarrow \mathsf{neg} \leftrightarrow c)$ instantiates the implementation of neg with the channel names i and c, while the syntax $P \parallel_c Q$ composes a server P with its client Q on c. The type of the composition channel c is determined by the ambient context $\{\, c : \mathsf{negBit}\ i \,\}$. The type checking algorithm checks each composed process against the types specified by the outer specification, i.e., that

$$\begin{aligned}&\mathsf{neg} :: \{\qquad\qquad\} \ [i : \mathsf{Bits}_1 \quad] \ (c : \mathsf{negBit}\ i \quad) = \cdots\\ &\mathsf{neg} :: \{i : \mathsf{Bits}_1\} \ [c : \mathsf{negBit}\ i] \ (o : \mathsf{IdBitSeq}_1\ i) = \cdots.\end{aligned}$$

We treat the channel i as ambient in the second specification (it is not accessible to its process). Typechecking the first process poses no difficulties; checking second process is interesting because it is a client of the channel c whose type immediately observes the ambient channel i. To check the second process, we do a case analysis on the messages that could appear on i, reduce the types of c and o accordingly in each case, and check that neg is well typed in the reduced specification. For instance, in the case where `0` appears on i, we check that:

$$\mathsf{neg} :: \{i : \mathbf{1}\} \ [c : \oplus\{\, \mathtt{1} : \mathbf{1} \,\}] \ (o : \oplus\{\, \mathtt{0} : \mathbf{1} \,\}) = \cdots.$$

As a side effect, we generate a constraint relating the label `0` on i to the behaviour of neg on c and o. After checking both processes independently, our algorithm checks that they impose mutually consistent constraints, and that they satisfy each other's constraints where applicable. In this example, the second process produces a constraint that neg receives `1` on c only if `0` appears on i; this constraint is clearly satisfied by the first negation process. Assuming that all constraints are satisfied, we conclude that the composition is well-typed.

The types of composition channels can be locally specified using the program syntax $\nu(a : A)\,.\,P$, which binds a private channel $a : A$ in P. This privacy also ensures that a cannot externally be observed. For example, we can hide the fact that doubleNeg is implemented as a composition and ensure that no other processes can observe its composition channel c by hiding $c : \mathsf{negBit}\ i$:

$$\mathsf{doubleNeg}' :: \{\,\} \ [i : \mathsf{Bits}_1] \ (o : \mathsf{IdBitSeq}_1\ i) = \nu(c : \mathsf{negBit}\ i)\,.\,\mathsf{doubleNeg}.$$

More generally, the syntax $\nu(a_1 : A_1, \ldots, a_n : A_n)\,.\,P$ binds channels $a_1, \ldots, a_n$ in P. To check $\nu\overrightarrow{a : A}\,.\,P$ against a specification, we check P against the same specification extended with the ambient channels $\overrightarrow{a : A}$. For example, checking doubleNeg′ entails checking the specification doubleNeg.

Our treatment of composition departs from many simply session typed systems [Caires and Pfenning 2010; Wadler 2014] that combine parallel composition with hiding, i.e., that define the composition of P and Q along a as $\nu(a : A)\,.\,P \parallel_a Q$. We distinguish these operations to more flexibly specify process compositions and to ensure that process composition is associative and partially commutative. Indeed, suppose we wished to chain together three processes:

$$\begin{aligned}&P_1 :: \{a_3 : A_3(a_2)\} \ [a_1 : \mathbf{1}] \ (a_2 : A_2(a_3))\\ &P_2 :: \{\,\} \ [a_2 : A_2(a_3)] \ (a_3 : A_3(a_2))\\ &P_3 :: \{a_2 : A_2(a_3)\} \ [a_3 : A_3(a_2)] \ (a_4 : \mathbf{1}),\end{aligned}$$

where we write $A_i(a_j)$ to mean A_i observes a_j, to form a process $P :: \{\,\} \ [a_1 : \mathbf{1}] \ (a_4 : \mathbf{1})$. The specification of P is clearly well-defined. However, if we always combine parallel composition with hiding, then neither of the following composition specifications is well-defined:

$$\begin{aligned}&P_{12} :: \{\,\} \ [a_1 : \mathbf{1}] \ (a_3 : A_2(a_3)) = \nu(a_2 : A_2(a_3))\,.\,\left(P_1 \parallel_{a_2} P_2\right)\\ &P_{23} :: \{\,\} \ [a_2 : A_2(a_3)] \ (a_4 : \mathbf{1}) = \nu(a_3 : A_3(a_2))\,.\,\left(P_2 \parallel_{a_3} P_3\right)\end{aligned}$$

This is because the types $A_2(a_3)$ and $A_3(a_2)$ are ill-scoped after hiding the channels a_2 and a_3. In contrast, if we hide a_2 and a_3 *after* the parallel composition, we can successfully define P as:

$$P :: \{\,\} \ [a_1 : \mathbf{1}] \ (a_4 : \mathbf{1}) = \nu\,(a_2 : A_2(a_3), a_3 : A_3(a_2))\,.\,\left(P_1 \parallel_{a_2} \left(P_2 \parallel_{a_3} P_3\right)\right).$$

Our examples show how Most modularly specifies processes while guaranteeing rich invariants.

3 A HIGHER-ORDER PROCESS LANGUAGE AND ITS TRACE SEMANTICS

Processes P are generated by the following grammar, where a and b range over channel names, L over sets of choice labels, l and k over labels, and A over types:

Process $P, Q ::= \mathsf{close}\ a$	End communication on a and terminate
$\mid\ \mathsf{wait}\ a;\ P$	Wait for communication to end on a; continue as P
$\mid\ a.k;\ P$	Send label k on a; continue as P
$\mid\ \mathsf{case}\ a\ \{\ l \Rightarrow P_l\ \}_{l \in L}$	Continue as P_l after receiving label $l \in L$ on a
$\mid\ \mathsf{send}\ a\ (b.P);\ Q$	Send a channel provided by P on a; continue as Q
$\mid\ b \leftarrow \mathsf{receive}\ a;\ P$	Receive channel b on a; continue as P
$\mid\ P \parallel_a Q$	Compose P and Q along channel a
$\mid\ \nu(a_1 : A_1, \dots, a_n : A_n)\,.\,P$	Introduce private channels a_i of type A_i in P

The channel name b is bound in P in $b \leftarrow \mathsf{receive}\ a;\ P$ and in $\mathsf{send}\ a\ (b.P);\ Q$. In $\mathsf{send}\ a\ (b.P);\ Q$, it represents the sent channel provided by P. Intuitively, this process forks P with b instantiated by a fresh name, and sends this name over a. The names $a_1, \dots, a_n$ are bound in P in $\nu(a_1 : A_1, \dots, a_n : A_n)\,.\,P$, and the type annotations exist only to simplify type checking. For the semantics to be reasonable, we require that a does not appear free in P in $\mathsf{wait}\ a;\ P$, that P and Q have disjoint sets of free names in $\mathsf{send}\ a\ (b.P);\ Q$, and that their free names intersect only in a in $P \parallel_a Q$. As is usual for session-typed processes, we consider *open* processes: the free names in a process name its used and provided channels.

A process denotes a set of traces that describe its possible executions. Traces are sequences of elements $(s; m)$, where m is a message on a channel and s is a tag indicating if m was sent or received, or if it appeared on an unhidden internal channel. To model sending and receiving channels (process constructs $\mathsf{send}\ a\ (b.P);\ Q$ and $b \leftarrow \mathsf{receive}\ a;\ P$), we extend traces with a binding structure similar to the coasbraction operator for nominal sequences [Gabbay and Ghica 2012]. To help ensure that our semantics is compositional, we bind b in the tail of a trace after a channel transmission message. This avoids free name clashes when interleaving traces in the denotation of process composition. Advantageously, our approach keeps the denotations of processes finite, a key property in ensuring that typechecking terminates. Explicitly, traces t, messages m, and signs s are given by the grammar:

$$\begin{array}{llll} \text{Observable Signs } o ::= \ ! \mid \ ? \mid \sigma & \text{Messages } m ::= \mathsf{close\ on}\ a & \text{Traces } t ::= \varepsilon \\ \text{Signs } s ::= o \mid ¿ & \quad\mid \mathsf{label}\ l\ \mathsf{on}\ a & \quad\mid (s;\ m) :: \vec{a}\,.\,t \\ & \quad\mid \mathsf{chan\ on}\ a & \end{array}$$

Observable **signs** specify actions performed by processes. They are ? (input), ! (output), and σ (internal synchronization). The constraint sign ¿ is not used in this section, but will be used in the semantics of message-observing types. It specifies a constraint on an ambient channel, i.e., that a message must be observed in the ambient environment before the trace can continue. **Messages** m describe possible messages on a given channel a. The message "close on a" means the end of communication, "label l on a" describes sending a label l, and "chan on a" captures sending a channel. We write $\mathsf{cc}(m)$ for the name of the channel carrying m; in the above cases, $\mathsf{cc}(m) = a$. Finally, **traces** follow a list-like structure. The empty trace is ε. The trace $(s;\ m) :: \vec{a}\,.\,t$ prefixes a message m with sign s onto trace t, with zero or more names $\vec{a}$ bound in t. We identify traces up to α-equivalence. The **free channel names** $\mathsf{fc}(t)$ in a trace t are defined by induction on the structure of t: $\mathsf{fc}(\varepsilon) = \emptyset$ and $\mathsf{fc}((s;\ m) :: \vec{a}\,.\,t) = \{\ \mathsf{cc}(m)\ \} \cup (\mathsf{fc}(t) \setminus \vec{a})$.

Processes denote sets of traces, where each trace denotes a possible interleaving of its actions. The denotation $[\![P]\!]$ of a process P is given by induction on its syntax. We explain the semantic clauses and the trace operators they use below:

$$
\begin{aligned}
[\![\mathsf{close}\ a]\!] &= \{\, (!;\ \mathsf{close\ on}\ a) :: \varepsilon \,\} && (1)\\
[\![\mathsf{wait}\ a;\ P]\!] &= (?;\ \mathsf{close\ on}\ a) :: [\![P]\!] && (2)\\
[\![a.k;\ P]\!] &= (!;\ \mathsf{label}\ k\ \mathsf{on}\ a) :: [\![P]\!] && (3)\\
[\![\mathsf{case}\ a\ \{\, l \Rightarrow P_l \,\}_{l\in L}]\!] &= \bigcup\nolimits_{l\in L} (?;\ \mathsf{label}\ l\ \mathsf{on}\ a) :: [\![P_l]\!] && (4)\\
[\![\mathsf{send}\ a\ (b.P);\ Q]\!] &= (!;\ \mathsf{chan\ on}\ a) :: b\,.\,([\![P]\!] \parallel [\![Q]\!]) && (5)\\
[\![b \leftarrow \mathsf{receive}\ a;\ P]\!] &= (?;\ \mathsf{chan\ on}\ a) :: b\,.\,[\![P]\!] && (6)\\
[\![P \parallel_a Q]\!] &= [\![P]\!] \parallel [\![Q]\!] && (7)\\
[\![\nu(a_1 : A_1, \dots, a_n : A_n)\,.\,P]\!] &= [\![P]\!] \setminus \{\, a_1, \dots, a_n \,\} && (8)\\
\text{where}\quad (s;\ m) :: \vec{a}\,.\,T &= \{\, (s;\ m) :: \vec{a}\,.\,t \mid t \in T \,\}\\
T_1 \parallel T_2 &= \{\, t \mid t_1 \in T_1,\ t_2 \in T_2,\ t \in (t_1 \parallel t_2) \,\}\\
T \setminus \{\, a_1, \dots, a_n \,\} &= \{\, t \setminus \{\, a_1, \dots, a_n \,\} \mid t \in T \,\}
\end{aligned}
$$

Equation (1) specifies that the only action performed by the process $(\mathsf{close}\ a)$ is sending a close message on a. Dually, eq. (2) specifies that every execution of the process $(\mathsf{wait}\ a;\ P)$ receives a close message on a before continuing as P.

Equation (3) similarly captures sending a label k on a. Equation (4) specifies that after receiving a label l, the process $P = (\mathsf{case}\ a\ \{\, l \Rightarrow P_l \,\}_{l\in L})$ continues executing as P_l. The meaning of the process P is then the union of all the traces for P_l prefixed by the received label l.

Equation (5) specifies sending a channel b provided by P over a and then continuing as Q. Processes P and Q execute independently, so the traces that follows sending b are interleavings of the executions of P and Q, i.e., elements of $[\![P]\!] \parallel [\![Q]\!]$. The interleaving operator $\parallel$ is defined in section 5.4. We assume without loss of generality that the bound name b is chosen distinct from any free name in Q. Because P and Q are assumed to have disjoint sets of free channel names, so will their traces. Therefore, $\parallel$ will effectively compute arbitrary interleavings of these two traces. Equation (6) specifies receiving a channel over a and binding it to the name b in P.

Equation (7) states that the parallel composition of processes is given by the synchronized interleavings of their traces. The synchronized interleavings of a pair of traces are defined in such a way that each receive action synchronizes with a send action when available. In particular, it captures a synchronous communication semantics. The process syntax $P\|_a Q$ specifies the channel name a along which P and Q are composed to specify the only name shared between P and Q and to guide typechecking. In contrast, the interleaving operator does not mention a. Our operator is sufficiently general to synchronize traces on multiple channels. This is required, e.g., to handle synchronization for channel transmission, where we must synchronize processes traces on two channels: the transmitted channel and the channel carrying it.

Finally, eq. (8) hides the bound channels a_i by deleting all messages appearing on a_i from the traces in $[\![P]\!]$. It uses the deletion operation $t \setminus \{\, a_1, \dots, a_n \,\}$ which is given in section 5.1. This deletion operator is defined in such a manner that it deletes not only actions on the a_i, but also actions appearing on the channels they carried, and so on and so forth.

To avoid proliferating definitions for trace operators, we postpone their formal definitions until we have discussed the slightly more general case of traces with constraints; readers wishing to skip ahead are invited to see section 5. Meanwhile, we consider a few intuition-building examples:

Example 3.1. We compute the denotation of the double negation process doubleNeg′ of section 2.6. We start by computing the denotation of the bit flipping process neg composing it:

$$\begin{aligned}[\![i \leftrightarrow \mathsf{neg} \leftrightarrow c]\!] = \{\, & (?;\ \mathsf{label}\ \mathtt{0}\ \mathsf{on}\ i) :: (!;\ \mathsf{label}\ 1\ \mathsf{on}\ c) :: (?;\ \mathsf{close}\ \mathsf{on}\ i) :: (!;\ \mathsf{close}\ \mathsf{on}\ c) :: \varepsilon, \\ & (?;\ \mathsf{label}\ 1\ \mathsf{on}\ i) :: (!;\ \mathsf{label}\ \mathtt{0}\ \mathsf{on}\ c) :: (?;\ \mathsf{close}\ \mathsf{on}\ i) :: (!;\ \mathsf{close}\ \mathsf{on}\ c) :: \varepsilon \,\}, \\ [\![c \leftrightarrow \mathsf{neg} \leftrightarrow o]\!] = \{\, & \text{analogous} \,\}.\end{aligned}$$

The denotation of doubleNeg uses the synchronized interleaving operator $\|$ to interleave traces from each denotation. Intuitively, it matches $(!;m)$ and $(?;m)$ elements and replaces them by synchronization elements $(\sigma;m)$; the remaining elements are freely interleaved. For example, it interleaves the following traces, respectively from $[\![i \leftrightarrow \mathsf{neg} \leftrightarrow c]\!]$ and $[\![c \leftrightarrow \mathsf{neg} \leftrightarrow o]\!]$,

$$\begin{aligned} & (?;\ \mathsf{label}\ \mathtt{0}\ \mathsf{on}\ i) :: (!;\ \mathsf{label}\ 1\ \mathsf{on}\ c) :: (?;\ \mathsf{close}\ \mathsf{on}\ i) :: (!;\ \mathsf{close}\ \mathsf{on}\ c) :: \varepsilon, \\ & (?;\ \mathsf{label}\ 1\ \mathsf{on}\ c) :: (!;\ \mathsf{label}\ \mathtt{0}\ \mathsf{on}\ o) :: (?;\ \mathsf{close}\ \mathsf{on}\ c) :: (!;\ \mathsf{close}\ \mathsf{on}\ o) :: \varepsilon,\end{aligned}$$

to produce, among others, the following traces in $[\![(i \leftrightarrow \mathsf{neg} \leftrightarrow c) \|_c (c \leftrightarrow \mathsf{neg} \leftrightarrow o)]\!]$:

$$\begin{aligned} & (?;\ \mathsf{label}\ \mathtt{0}\ \mathsf{on}\ i) :: (\sigma;\ \mathsf{label}\ 1\ \mathsf{on}\ c) :: (!;\ \mathsf{label}\ \mathtt{0}\ \mathsf{on}\ o) :: (?;\ \mathsf{close}\ \mathsf{on}\ i) :: (\sigma;\ \mathsf{close}\ \mathsf{on}\ c) :: \cdots, \\ & (?;\ \mathsf{label}\ \mathtt{0}\ \mathsf{on}\ i) :: (\sigma;\ \mathsf{label}\ 1\ \mathsf{on}\ c) :: (?;\ \mathsf{close}\ \mathsf{on}\ i) :: (!;\ \mathsf{label}\ \mathtt{0}\ \mathsf{on}\ o) :: (\sigma;\ \mathsf{close}\ \mathsf{on}\ c) :: \cdots.\end{aligned}$$

Contrasting the two given traces illustrates the concurrent nature of message-passing processes. In the first execution, the left neg process sends `0` before the right receives its close message on i; this order is reversed in the second trace. Finally, the denotation of doubleNeg′ hides all messages on c:

$$\begin{aligned}[\![\nu(c : \mathsf{negBit}\ i)\,.\,\mathsf{doubleNeg}]\!] = \{\, & (?;\ \mathsf{label}\ \mathtt{0}\ \mathsf{on}\ i) :: (!;\ \mathsf{label}\ \mathtt{0}\ \mathsf{on}\ o) :: (?;\ \mathsf{close}\ \mathsf{on}\ i) :: \cdots, \\ & (?;\ \mathsf{label}\ \mathtt{0}\ \mathsf{on}\ i) :: (?;\ \mathsf{close}\ \mathsf{on}\ i) :: (!;\ \mathsf{label}\ \mathtt{0}\ \mathsf{on}\ o) :: \cdots, \ldots \,\}.\end{aligned}$$

Example 3.2. To illustrate how traces with binding help model channel transmission, consider the parallel composition $\mathsf{comp} :: \{\,\}\ [\,]\ (c : \mathbf{1}) = P \|_a Q$, where

$$\begin{aligned} P &:: \{\,\}\ [\,]\ (a : \mathbf{1} \otimes \mathbf{1}) &&= \mathsf{send}\ a\ (c.\mathsf{close}\ c);\ \mathsf{close}\ a \\ Q &:: \{\,\}\ [a : \mathbf{1} \otimes \mathbf{1}]\ (c : \mathbf{1}) &&= b \leftarrow \mathsf{receive}\ a;\ \mathsf{wait}\ a;\ \mathsf{wait}\ b;\ \mathsf{close}\ c\end{aligned}$$

Process P denotes two traces, capturing the independent executions of processes close a and close c:

$$\begin{aligned}\{\, & (!;\ \mathsf{chan}\ \mathsf{on}\ a) :: c\,.\,(!;\ \mathsf{close}\ \mathsf{on}\ c) :: (!;\ \mathsf{close}\ \mathsf{on}\ a) :: \varepsilon, \\ & (!;\ \mathsf{chan}\ \mathsf{on}\ a) :: c\,.\,(!;\ \mathsf{close}\ \mathsf{on}\ a) :: (!;\ \mathsf{close}\ \mathsf{on}\ c) :: \varepsilon \,\}\end{aligned}$$

The process Q exhibits no non-determinism and denotes a single trace:

$$[\![Q]\!] = \{\, (?;\ \mathsf{chan}\ \mathsf{on}\ a) :: b\,.\,(?;\ \mathsf{close}\ \mathsf{on}\ a) :: (?;\ \mathsf{close}\ \mathsf{on}\ b) :: (!;\ \mathsf{close}\ \mathsf{on}\ c) :: \varepsilon \,\}$$

Their parallel composition denotes a synchronized interleaving of these two trace sets. It is here that binding pays off: we can α-vary the bound name c identifying the transmitted channel in the traces of P to a fresh name b to avoid clashing with the free name c in the trace of Q. The interleaved trace set captures synchronization on both a and the transmitted channel:

$$[\![\mathsf{comp}]\!] = \{\, (\sigma;\ \mathsf{chan}\ \mathsf{on}\ a) :: b\,.\,(\sigma;\ \mathsf{close}\ \mathsf{on}\ a) :: (\sigma;\ \mathsf{close}\ \mathsf{on}\ b) :: (!;\ \mathsf{close}\ \mathsf{on}\ c) :: \varepsilon \,\}$$

It contains a single trace, because only the first of P's executions successfully synchronizes with Q.

There are no non-terminating executions, i.e., every process performs a finite number of actions:

Proposition 3.3 (Termination). *For all processes P and traces $t \in [\![P]\!]$, t is finite.*

4 MESSAGE-OBSERVING SESSION TYPES AND PROCESS SPECIFICATIONS

Our type soundness result will show that if a process typechecks against a specification, then its range of behaviours (the traces it denotes) is among those allowed by its specification. We specify which behaviours a process specification allows by giving specifications a trace semantics in section 4.2. First, we define message-observing session types and process specifications.

Recall that each process is a client of some services and the server of a distinguished service. From the perspective of a server of type A, session types are generated by the grammar:

$$\begin{array}{rlll}
\text{Session Types} & A, B ::= & \mathbf{1} & \text{End communication and terminate}\\
& \mid & \oplus_{l\in L} A_l & \text{Internal choice}\\
& \mid & \&_{l\in L} A_l & \text{External choice}\\
& \mid & (a : A) \otimes (b : B) & \text{Channel transmission}\\
& \mid & (a : A) \multimap (b : B) & \text{Channel reception}\\
& \mid & \mathsf{CASE}\ a\ \{\,\mathsf{close} \Rightarrow A\,\} & \text{Termination observation}\\
& \mid & \mathsf{CASE}\ a\ \{\,l \Rightarrow A_l\,\}_{l\in L} & \text{Label observation}\\
& \mid & \mathsf{CASE}\ a\ \{\,\langle b\rangle \Rightarrow A\,\} & \text{Higher-order observation}
\end{array}$$

$$\text{Session Type Context } \Pi, \Delta, \mathrm{I} ::= a_1 : A_1, \ldots, a_n : A_n$$

We explain session types from the perspective of a server. A server of $\mathbf{1}$ sends a close message signalling the end of communication on a channel. The internal choice type $\oplus_{l\in L} A_l$ specifies sending a label $l \in L$, and then communicating according to A_l. Dually, the external choice type $\&_{l\in L} A_l$ specifies receiving a label $l \in L$, and then communicating according to A_l. Many session-typed languages provide a session type $A \otimes B$. A server for a channel c of type $A \otimes B$ sends a channel a of type A on c and then communicates on c according to B. Our syntax $(a : A) \otimes (b : B)$ extends the usual syntax to support mutual observation between the types A and B. It binds the name a in B to allow B to observe communications on the transmitted channel, and it binds b in A to allow A to observe subsequent communications on the carrier channel. A server of a channel c of type $(a : A) \otimes (b : B)$ sends a channel a of type $[c/b]A$, and then communicates on c according to B. Symmetrically, a server of $c : (a : A) \multimap (b : B)$ specifies receiving a channel a of type $[c/b]A$ and then communicating on c according to B. This type has an identical binding structure. We call these types **weak head normal** because they do not involve computation at the outermost level.

Observing types restrict communication based on communication on ambient channels. The type $\mathsf{CASE}\ a\ \{\,\mathsf{close} \Rightarrow A\,\}$ observes an ambient channel a and reduces to A when a close message is sent on a. The type $\mathsf{CASE}\ a\ \{\,l \Rightarrow A_l\,\}_{l\in L}$ reduces to the type A_l when a label l is sent on a. Finally, $\mathsf{CASE}\ a\ \{\,\langle b\rangle \Rightarrow A\,\}$ reduces to A if a channel named b is transmitted over a. Because a priori, we do not know the name of the transmitted channel, $\mathsf{CASE}\ a\ \{\,\langle b\rangle \Rightarrow A\,\}$ binds the name b in A.

The free channels in a type are inductively defined on the structure in the obvious manner.

Our motivating examples used a concrete syntax "name :: $\{\Sigma\}$ $[\Delta]$ $(c : C) = P$" to specify a process name, implemented by P, that used channels Δ to provide a service $c : C$, where these channels were free to observe ambient channels Σ. These ambient channels were either (unhidden) internal channels, or channels in other processes whose meaning would be given by process composition. By inspecting the syntax of P, we can partition Σ into a context I of internal channels and a context Π containing the remaining ambient channels. We can then translate the concrete specification syntax $\{\Pi, \mathrm{I}\}$ $[\Delta]$ $(c : C)$ into the abstract syntax $\Pi \mathrel{\#} (\Delta \mid \mathrm{I} \vdash c : C)$. Our abstract syntax distinguishes Π and I because they play different roles in typechecking: channels in I can be hidden by P, while those in Π cannot appear in P. Most's typing judgment $P \Vdash \Pi \mathrel{\#} (\Delta \mid \mathrm{I} \vdash c : C) /\!\!/ \mathfrak{T}$

checks P against its specification, generating constraints $\mathfrak{T}$. Our concrete syntax names processes only to simplify examples; process names do not appear in our formal system.

To specify channel transmission, we need to specify multiple independent processes that execute concurrently. Indeed, send a $(b.P)$; Q spawns process P providing a fresh b, sends b over a, and runs P concurrently with Q providing a. Checking send a $(b.P)$; Q against its specification requires decomposing its specification into specifications for P and Q, and checking P and Q. Accordingly, we extend our abstract syntax for specifications to simultaneously specify multiple independent processes. We write $\Pi \# ((\Delta_1 \mid \mathrm{I}_1 \vdash c_1 : C_1) \mid \cdots \mid (\Delta_n \mid \mathrm{I}_n \vdash c_n : C_n))$ to simultaneously specify n processes, where channels Π and Δ_i, $\mathrm{I}_i, c_i : C_i$ are ambient in Δ_j, $\mathrm{I}_j, c_j : C_j$ for $i \neq j$. We call $(\Delta \mid \mathrm{I} \vdash c : C)$ a process **interface**: it specifies the communication interface for a process. We range over interface sequences $((\Delta_1 \mid \mathrm{I}_1 \vdash c_1 : C_1) \mid \cdots \mid (\Delta_n \mid \mathrm{I}_n \vdash c_n : C_n))$ using the meta-variable $\mathcal{G}$.

A specification $\Pi \# \mathcal{G}$ is well-defined only if for each $a : A$ in Π or $\mathcal{G}$, the free channels in A are contained in the domain of $\Pi \# \mathcal{G}$, i.e., are assigned types by the specification. This precludes, e.g., $\cdot \# (a : A \mid \cdot \vdash c : \mathsf{CASE}\ b \{ \mathsf{close} \Rightarrow C \})$ because the channel b is not typed by the specification.

4.1 Type Reductions

In our motivating examples, we informally reduced types in specifications after observing messages on channel. To make this explicit, we first define reduction on types, and then extend it to entire specifications. Type reduction is given by a syntactic operation A/π that reduces a type A by an observed communication π. **Observed communication**, close on a, label l on a, and chan b on a, closely resembles messages from the trace semantics. The only difference is in the case of channel transmissions, where a free channel name b identifies the transmitted channel. This free name will be used by the process typing judgment to update types with the name of the received channel. We write $\mathrm{fc}(\pi)$ for the free channels in π, and $\mathrm{cc}(\pi) = a$ for the **carrier channel**. **Type reduction** is partial, capture-avoiding, and inductively defined by the following clauses (all other cases are undefined):

$$\begin{aligned}
\mathbf{1}/\pi &= \mathbf{1} \\
(\odot_{l \in L} A_l)/\pi &= \odot_{l \in L} (A_l/\pi) && \text{for } \odot \in \{ \oplus, \& \} \\
((a : A) \odot (b : B))/\pi &= (a : A/\pi) \odot (b : B/\pi) && \text{for } \odot \in \{ \otimes, \multimap \} \\
(\mathsf{CASE}\ c \{ \mathsf{close} \Rightarrow A \})/\pi &= \begin{cases} A & \text{if } \pi = \mathsf{close\ on}\ c \\ \mathsf{CASE}\ c \{ \mathsf{close} \Rightarrow A/\pi \} & \text{if } \mathrm{cc}(\pi) \neq c \end{cases} \\
(\mathsf{CASE}\ c \{ l \Rightarrow A_l \}_{l \in L})/\pi &= \begin{cases} A_l & \text{if } \pi = \mathsf{label}\ l\ \mathsf{on}\ c \text{ and } l \in L \\ \mathsf{CASE}\ c \{ l \Rightarrow A_l/\pi \}_{l \in L} & \text{if } \mathrm{cc}(\pi) \neq c \end{cases} \\
(\mathsf{CASE}\ c \{ \langle b \rangle \Rightarrow A \})/\pi &= \begin{cases} A & \text{if } \pi = \mathsf{chan}\ b\ \mathsf{on}\ c \\ \mathsf{CASE}\ c \{ \langle b \rangle \Rightarrow A/\pi \} & \text{if } \mathrm{cc}(\pi) \neq c \end{cases}
\end{aligned}$$

When a message is observed on a channel in a process specification, we simultaneously reduce all types in the specification. This eliminates the need to track past observations and it simplifies the presentation. The syntax $(a_1 : A_1, \ldots, a_n : A_n)/\pi = a_1 : (A_1/\pi), \ldots, a_n : (A_n/\pi)$ simultaneously reduces contexts; it is defined whenever all A_i/π are defined. It lifts to specifications context-wise.

4.2 The Semantics of Specifications

A specification $\Pi \# \mathcal{G}$ specifies a collection of permitted behaviours, i.e., a trace set $[\![\Pi \# \mathcal{G}]\!]$. To make this discussion concrete, consider a specification $\Pi \# (\Delta \mid \mathrm{I} \vdash c : C)$. The traces in $[\![\Pi \# (\Delta \mid \mathrm{I} \vdash c : C)]\!]$ are process traces that involve sending or receiving messages on $\Delta, c : C$, with synchronizations on I. However, these traces may also contain **constraint elements** $(¿; m)$. An element $(¿;\mathsf{close}\ \mathsf{on}\ a)$ means that the execution described by the remaining tail of the trace is permitted only if a close message appears on the ambient channel $a \in \mathrm{dom}(\Pi)$. Constraint elements $(¿;\mathsf{label}\ l\ \mathsf{on}\ a)$ and $(¿;\mathsf{chan}\ \mathsf{on}\ a)$ are analogous. We use constraint elements to semantically specify that a process is well-typed, provided that some other processes perform some action.

A trace set $[\![\Pi \mathbin{\#} \mathcal{G}]\!]$ is the least set satisfying the following collection of inequalities. It is well-defined because type reduction reduces the number of type operators in a specification, so the right-hand specification always contains fewer type operators than the left-hand specification. The right-hand side may be undefined; in this case, we treat it as an empty set.

We first specify label transmission. Where $\pi = \mathsf{label}\ k\ \mathsf{on}\ a$, $m = \mathsf{label}\ k\ \mathsf{on}\ a$, and $k \in L$:

$$[\![\Pi \mathbin{\#} \mathcal{G} \mid (\Delta \mathbin{\shortmid} \mathrm{I} \vdash a : \oplus_{l\in L} A_l)]\!] \supseteq (!;\ m) :: [\![\Pi/\pi \mathbin{\#} \mathcal{G}/\pi \mid (\Delta/\pi \mathbin{\shortmid} \mathrm{I}/\pi \vdash a : A_k)]\!] \tag{9}$$

$$[\![\Pi \mathbin{\#} \mathcal{G} \mid (\Delta, a : \oplus_{l\in L} A_l \mathbin{\shortmid} \mathrm{I} \vdash c : C)]\!] \supseteq (?;\ m) :: [\![\Pi/\pi \mathbin{\#} \mathcal{G}/\pi \mid (\Delta/\pi, a : A_k \mathbin{\shortmid} \mathrm{I}/\pi \vdash c : C/\pi)]\!] \tag{10}$$

$$[\![\Pi \mathbin{\#} \mathcal{G} \mid (\Delta \mathbin{\shortmid} \mathrm{I}, a : \oplus_{l\in L} A_l \vdash c : C)]\!] \supseteq (\sigma;\ m) :: [\![\Pi/\pi \mathbin{\#} \mathcal{G}/\pi \mid (\Delta/\pi \mathbin{\shortmid} \mathrm{I}/\pi, a : A_k \vdash c : C/\pi)]\!] \tag{11}$$

The first clause specifies that a provider of type $\oplus_{l\in L} A_l$ can send any label l in L. The provider must obey its original specification, reduced by the just-observed message label l on a. Receiving a label is dually specified. The last clause specifies that a process formed of a composition along $a : \oplus_{l\in L} A_l$ can perform a synchronization on that channel. We define clauses with inequalities to avoid imposing an order on communication on different channels. For example, given a specification $\cdot \mathbin{\#} (a : \oplus(\mathtt{0} : \mathbf{1}) \mathbin{\shortmid} \cdot \vdash b : \oplus(\mathtt{0} : \mathbf{1}))$, the above clauses ensure that a and b can be used in either order because its denotation includes traces starting with either channel.

A provider of type $\mathbf{1}$ can terminate only if it has no other channels. This ensures that channels do not accidentally get discarded and it is captured by the first clause below. Receiving and synchronizing on channels of type $\mathbf{1}$ ends communication on that channel. The last clause captures termination when there are no processes left. Where $\pi = \mathsf{close}\ \mathsf{on}\ a$:

$$[\![\Pi \mathbin{\#} \mathcal{G} \mid (\cdot \mathbin{\shortmid} \cdot \vdash a : \mathbf{1})]\!] \supseteq (!;\ \mathsf{close}\ \mathsf{on}\ a) :: [\![\Pi/\pi \mathbin{\#} \mathcal{G}/\pi]\!] \tag{12}$$

$$[\![\Pi \mathbin{\#} \mathcal{G} \mid (\Delta, a : \mathbf{1} \mathbin{\shortmid} \mathrm{I} \vdash c : C)]\!] \supseteq (?;\ \mathsf{close}\ \mathsf{on}\ a) :: [\![\Pi/\pi \mathbin{\#} \mathcal{G}/\pi \mid (\Delta/\pi \mathbin{\shortmid} \mathrm{I}/\pi \vdash c : C/\pi)]\!] \tag{13}$$

$$[\![\Pi \mathbin{\#} \mathcal{G} \mid (\Delta \mathbin{\shortmid} \mathrm{I}, a : \mathbf{1} \vdash c : C)]\!] \supseteq (\sigma;\ \mathsf{close}\ \mathsf{on}\ a) :: [\![\Pi/\pi \mathbin{\#} \mathcal{G}/\pi \mid (\Delta/\pi \mathbin{\shortmid} \mathrm{I}/\pi \vdash c : C/\pi)]\!] \tag{14}$$

$$[\![\Pi \mathbin{\#} \cdot]\!] \supseteq \{\varepsilon\} \tag{15}$$

Channel transmission changes the shape of specifications to capture spawning processes. Where $\pi = (\mathsf{chan}\ b\ \mathsf{on}\ a)$ with b fresh:

$$\begin{aligned}&[\![\Pi \mathbin{\#} \mathcal{G} \mid (\Delta_1, \Delta_2 \mathbin{\shortmid} \mathrm{I}_1, \mathrm{I}_2 \vdash a : (b : B) \otimes (a : A))]\!]\\ &\supseteq (!;\ \mathsf{chan}\ \mathsf{on}\ a) :: b\,.\,[\![\Pi/\pi \mathbin{\#} \mathcal{G}/\pi \mid (\Delta_1/\pi \mathbin{\shortmid} \mathrm{I}_1/\pi \vdash b : B) \mid (\Delta_2/\pi, a : A \mathbin{\shortmid} \mathrm{I}_2/\pi \vdash c : C/\pi)]\!]\end{aligned} \tag{16}$$

$$\begin{aligned}&[\![\Pi \mathbin{\#} \mathcal{G} \mid (\Delta, a : (b : B) \otimes (a : A) \mathbin{\shortmid} \mathrm{I} \vdash c : C)]\!]\\ &\supseteq (?;\ \mathsf{chan}\ \mathsf{on}\ a) :: b\,.\,[\![\Pi/\pi \mathbin{\#} \mathcal{G}/\pi \mid (\Delta/\pi, b : B, a : A \mathbin{\shortmid} \mathrm{I}/\pi \vdash c : C/\pi)]\!]\end{aligned} \tag{17}$$

The first clause specifies the permitted behaviours of a provider $\mathsf{send}\ a\ (b.P);\ Q$. Operationally, this process forks P and Q, respectively providing b and a, and sends b over a. The used and internal channels of $\mathsf{send}\ a\ (b.P);\ Q$ are then divided between P and Q. Accordingly, the first clause splits its interface $\Delta_1, \Delta_2 \mathbin{\shortmid} \mathrm{I}_1, \mathrm{I}_2 \vdash a : (b : B) \otimes (a : A)$ in two. The channels assigned to P and Q depend on the particular implementation, so the clause allows for arbitrary partitions Δ_1, Δ_2 and $\mathrm{I}_1, \mathrm{I}_2$. This clause also illustrates why the observed message $\mathsf{chan}\ b\ \mathsf{on}\ a$ (used in type reductions) specifies a free name b, in contrast to the message element $\mathsf{chan}\ \mathsf{on}\ a$ used in traces. When we reduce the remaining types in the specification to account for the channel transmission, we must specify the free name b that carries the communications of P. However, this name b is not externally meaningful, and we bind it in the trace to avoid name clashes in process compositions. The second clause captures receiving a channel over b; it updates the context to contain the received channel. The elided clause for synchronization is analogous to the second clause.

Finally, we specify the constraints generated by ambient channels in Π:

$$[\![\Pi, a : \mathbf{1} \# \mathcal{G}]\!] \supseteq (¿;\ \mathsf{close\ on}\ a) :: [\![\Pi/\pi \# \mathcal{G}/\pi]\!], \quad \text{where } \pi = \mathsf{close\ on}\ a; \tag{18}$$

$$[\![\Pi, a : \odot_{l \in L} A_l \# \mathcal{G}]\!] \supseteq (¿;\ \mathsf{label}\ l\ \mathsf{on}\ a) :: [\![\Pi/\pi, a : A_l \# \mathcal{G}/\pi]\!], \quad \text{where } \pi = \mathsf{label}\ l\ \mathsf{on}\ a, l \in L, \text{ and } \odot \in \{ \oplus, \& \}; \tag{19}$$

$$[\![\Pi, a : (b : B) \odot (a : A) \# \mathcal{G}]\!] \supseteq (¿;\ \mathsf{chan\ on}\ a) :: b\,.\,[\![\Pi/\pi, a : A, b : B \# \mathcal{G}/\pi]\!], \quad \text{where } \pi = \mathsf{chan}\ b\ \mathsf{on}\ a, \text{ and } \odot \in \{ \otimes, \multimap \}. \tag{20}$$

The first clause specifies that if we reduce a specification by observing a close message on an ambient channel a, then the traces permitted by the reduced specification are valid provided that some process in the environment closes a. The second clause is analogous. The third clause captures the fact that sending a channel results in a pair of channels in a context: the transmitted channel and its carrier. We assume without loss of generality that b is fresh in the third clause.

The omitted clauses for $\multimap$ and & are analogous. There are no clauses for observing types: they allow communication only after reducing to a weak-head normal type. In particular, type reductions are silent and they occur as a result of communication on channels with weak-head normal types.

Example 4.1. The concrete specification neg :: { } [i : Bits$_1$] (c : negBit i) of section 2.6 corresponds to the specification $\cdot \# (i : \mathsf{Bits}_1 \mid \cdot \vdash c : \mathsf{negBit}\ i)$. It only allows the executions

$$(?;\ \mathsf{label}\ \mathtt{0}\ \mathsf{on}\ i) :: (!;\ \mathsf{label}\ \mathtt{1}\ \mathsf{on}\ c) :: (?;\ \mathsf{close\ on}\ i) :: (!;\ \mathsf{close\ on}\ c) :: \varepsilon,$$
$$(?;\ \mathsf{label}\ \mathtt{0}\ \mathsf{on}\ i) :: (?;\ \mathsf{close\ on}\ i) :: (!;\ \mathsf{label}\ \mathtt{1}\ \mathsf{on}\ c) :: (!;\ \mathsf{close\ on}\ c) :: \varepsilon,$$

plus the two traces obtained by exchanging 0 and 1. The first two traces illustrate the concurrent nature of communication on different channels. Indeed, negBit i only allows communication on c after communication on i, but no other ordering is imposed between messages on i and c.

Example 4.2. The concrete specification neg :: {i : Bits$_1$} [c : negBit i] (o : IdBitSeq$_1$ i) of section 2.6 corresponds to the specification $i : \mathsf{Bits}_1 \# (c : \mathsf{negBit}\ i \mid \cdot \vdash o : \mathsf{IdBitSeq}_1\ i)$. It allows:

$$(¿;\ \mathsf{label}\ \mathtt{0}\ \mathsf{on}\ i) :: (?;\ \mathsf{label}\ \mathtt{1}\ \mathsf{on}\ c) :: (!;\ \mathsf{label}\ \mathtt{0}\ \mathsf{on}\ o) :: (¿;\ \mathsf{close\ on}\ i) :: (?;\ \mathsf{close\ on}\ c) :: (!;\ \mathsf{close\ on}\ o) :: \varepsilon,$$
$$(¿;\ \mathsf{label}\ \mathtt{0}\ \mathsf{on}\ i) :: (¿;\ \mathsf{close\ on}\ i) :: (?;\ \mathsf{label}\ \mathtt{1}\ \mathsf{on}\ c) :: (!;\ \mathsf{label}\ \mathtt{0}\ \mathsf{on}\ o) :: (?;\ \mathsf{close\ on}\ i) :: (!;\ \mathsf{close\ on}\ o) :: \varepsilon, \text{ etc.}$$

Interpreting each trace as an execution or sequence of actions permitted to neg, the first trace specifies that if a label 0 is sent on an ambient channel i, then neg can receive a bit 1 on c, send 0 on o, and so on and so forth. The second trace is analogous, but illustrates that we impose no extraneous orderings on messages on different channels.

Example 4.3. The specification $a : \mathbf{1}, b : \mathbf{1} \# (\cdot \mid \cdot \vdash c : \mathsf{CASE}\ a\ \{ \mathsf{close} \Rightarrow \mathsf{CASE}\ b\ \{ \mathsf{close} \Rightarrow \mathbf{1} \} \})$ permits two traces:

$$(¿;\ \mathsf{close\ on}\ a) :: (¿;\ \mathsf{close\ on}\ b) :: (!;\ \mathsf{close\ on}\ c) :: \varepsilon,$$
$$(¿;\ \mathsf{close\ on}\ b) :: (¿;\ \mathsf{close\ on}\ a) :: (!;\ \mathsf{close\ on}\ c) :: \varepsilon.$$

These two traces illustrate that the type of c does not impose an order on ambient communications: because we can reduce under CASE, communication on a and b can occur in any order. Indeed, the first trace corresponds to reducing the specification to $b : \mathbf{1} \# (\cdot \mid \cdot \vdash c : \mathsf{CASE}\ b\ \{ \mathsf{close} \Rightarrow \mathbf{1} \})$

(after observing close on a) and then to $\cdot \mathbin{\#} (\cdot \mid \cdot \vdash c : \mathbf{1})$ (after observing close on b); the second captures first reducing to $a : \mathbf{1} \mathbin{\#} (\cdot \mid \cdot \vdash c : \text{CASE } a \{ \text{close} \Rightarrow \mathbf{1} \})$, then to $\cdot \mathbin{\#} (\cdot \mid \cdot \vdash c : \mathbf{1})$.

Example 4.4. The specification $\cdot \mathbin{\#} (\cdot \mid \cdot \vdash a : (b : \mathbf{1}) \otimes \text{CASE } b \{ \text{close} \Rightarrow \mathbf{1} \})$ specifies sending a channel of type $\mathbf{1}$, and with the carrier closing after the sent channel. Its trace set is:

$$\begin{aligned}
&(!;\ \text{chan on } a) :: b \,.\, [\![\cdot \mathbin{\#} (\cdot \mid \cdot \vdash b : \mathbf{1}) \mid (\cdot \mid \cdot \vdash a : \text{CASE } b \{ \text{close} \Rightarrow \mathbf{1} \})]\!] \\
&= (!;\ \text{chan on } a) :: b \,.\, (!;\ \text{close on } b) :: [\![\cdot \mathbin{\#} (\cdot \mid \cdot \vdash a : \mathbf{1})]\!] \\
&= \{ (!;\ \text{chan on } a) :: b \,.\, (!;\ \text{close on } b) :: (!;\ \text{close on } a) :: \varepsilon \}.
\end{aligned}$$

This example shows how multiple interfaces in a specification are used to specify relationships between messages from different processes that are spawned in the course of channel transmission.

5 OPERATIONS ON TRACES

Our trace semantics for processes uses two operations on traces: deletion and synchronized interleaving. Name deletion deletes all messages in a trace whose names appear in a given set, while synchronized interleavings interleave two traces while ensuring that input and output actions match up. We also specify how to delete constraints elements from traces. All three operations will be used by our typechecking algorithm when generating traces with constraints.

5.1 Deleting Channel Names from Traces

The construct $\nu(\overrightarrow{a_i : A_i}) \,.\, P$ binds the channels a_i to hide them from external view. Semantically, hiding is captured by deleting all messages on channels a_i from the traces of P. To account for channel transmission, we may need to delete messages on other channel names. In particular, if a_i carries some channel b, then we must also delete all messages on b from the trace.

Given a set X of names and a trace t, the deletion of X in t is the trace $t \setminus X$ inductively defined by:

$$\varepsilon \setminus X = \varepsilon \quad \text{and} \quad ((s;\ m) :: c_1, \dots, c_n \,.\, t) \setminus X = \begin{cases} t \setminus (X \cup \{ c_1, \dots, c_n \}) & \text{if } \mathrm{cc}(m) \in X \\ (s;\ m) :: c_1, \dots, c_n \,.\, (t \setminus X) & \text{otherwise} \end{cases}$$

We assume without loss of generality that the names c_i are chosen distinct from those already in X.

Example 5.1. If $t = (?;\ \text{chan on } a) :: b \,.\, (?;\ \text{close on } a) :: (?;\ \text{close on } b) :: (!;\ \text{close on } c) :: \varepsilon$, then $t \setminus \{ a \} = (!;\ \text{close on } c) :: \varepsilon$.

5.2 Deleting Constraints from Traces

The operation $t/¿$ deletes all constraints from t. We will use it when checking that processes have consistent constraint sets when typing process compositions. It is inductively defined by

$$\varepsilon/¿ = \varepsilon \qquad \text{and} \qquad (s;\ m) :: \vec{x} \,.\, t/¿ = \begin{cases} t/¿ & \text{if } s = ¿ \\ (s;\ m) :: \vec{x} \,.\, t/¿ & \text{otherwise} \end{cases}$$

5.3 Trace Reduction

When interleaving two traces, we need to delete constraints in one trace satisfied by messages in the other trace. We do so using a partial operation t/π that deletes a constraint $(¿; m)$ in t if it is satisfied by the observed communication π. It is inductively defined on the structure of the trace:

$$\varepsilon/\pi = \varepsilon \qquad ((¿;\ \text{close on } a) :: t)/(\text{close on } a) = t$$
$$((¿;\ \text{label } l \text{ on } a) :: t)/(\text{label } l \text{ on } a) = t \qquad ((¿;\ \text{chan on } a) :: c \,.\, t)/(\text{chan } b \text{ on } a) = [b/c]t$$
$$((s;\ m) :: \vec{x} \,.\, t)/\pi = (s;\ m) :: \vec{x} \,.\, (t/\pi) \quad \text{if } \mathrm{cc}(m) \neq \mathrm{cc}(\pi),$$

and it is undefined in all other cases. The fourth clause instantiates a bound name c for a transmitted channel with the free name b actually observed in the course of channel transmission. The last clause handles an observation on a channel different from the head of the trace. It is defined only if t/π is defined. We always α-vary $\vec{x}$ to be distinct from the names in π.

Example 5.2. The reduction $((¿;\ \text{label}\ k\ \text{on}\ a) :: \varepsilon)/(\text{label}\ l\ \text{on}\ a)$ is undefined because the label l observed on a does not match the label k expected by the constraint. Similarly, the reduction $((!;\ \text{close on}\ a) :: \varepsilon)/(\text{label}\ l\ \text{on}\ a)$ is undefined because the trace we are reducing sends on a, while the observation "label l on a" is meant to be an observation on an ambient channel a.

5.4 Synchronized Trace Interleavings

Synchronized interleavings interleave traces according to a synchronous communication semantics. It marks matching input-output elements as synchronized, and ensures that constraints imposed by one trace are not violated by the other trace. This commutative operator is defined by lexicographic induction on the length of the traces. We group its clauses into four categories:

(1) the base cases: interleaving a trace with an empty trace;
(2) synchronizing cases: the heads of each trace perform actions on the same channel;
(3) commuting cases: the heads of each trace perform actions on different channels;
(4) ill-defined cases: unsynchronizeable sends and receives, etc.

The base cases specify that the empty trace is the unit for interleaving:

$$\varepsilon \parallel t = \{\, t \,\}, \qquad t \parallel \varepsilon = \{\, t \,\}.$$

The first synchronizing case holds whenever two traces send and receive the same message:

$$((!;\ m) :: \vec{x} \,.\, t_1) \parallel ((?;\ m) :: \vec{x} \,.\, t_2) = (\sigma;\ m) :: \vec{x} \,.\, (t_1 \parallel t_2)\,.$$

It specifies that these two traces synchronize (captured by the σ sign), and that the tail is given by interleaving their tails. To ensure that the resulting trace is well-formed, we require that both traces bind the same names. It is this implicit α-variation that matches sent and received channel names b when interleaving traces of the form $(!;\ \text{chan on}\ a) :: b \,.\, t_1$ and $(?;\ \text{chan on}\ a) :: b \,.\, t_2$.

The other principal case manages constraints. It states that a constraint in one trace can be dropped whenever it is satisfied by a corresponding action in the other trace, or whenever both traces have the same constraint. If a constraint is matched against a different message on that channel, then the constraint cannot be satisfied and there are no interleavings.

$$((s;\ m) :: \vec{x} \,.\, t_1) \parallel ((¿;\ m) :: \vec{x} \,.\, t_2) = (s;\ m) :: \vec{x} \,.\, (t_1 \parallel t_2)$$

The commuting cases handle heads that act on different channels. Let $(m \propto \vec{x})$ be the observed communication induced by a message m and channel names $\vec{x}$: $(\text{close on}\ a \propto \emptyset) = \text{close on}\ a$; $(\text{label}\ l\ \text{on}\ a \propto \emptyset) = \text{label}\ l\ \text{on}\ a$; $(\text{chan on}\ a \propto b) = \text{chan}\ b\ \text{on}\ a$; and $(m \propto \vec{x})$ is undefined otherwise. Given a partial map f, let $f(x) \simeq y$ define y whenever $f(x)$ is defined. Where $T_i = (s_i;\ m_i) :: \vec{x}_i \,.\, t_i$,

$$\left.\begin{aligned} &(s_1;\ m_1) :: \vec{x}_1 \,.\, t_1 \parallel (s_2;\ m_2) :: \vec{x}_2 \,.\, t_2 \\ &= \{\, (s_1;\ m_1) :: \vec{x}_1 \,.\, t \mid T_2/(m_1 \propto \vec{x}_1) \simeq T_2' \wedge t \in t_1 \parallel T_2' \,\} \cup \\ &\cup \{\, (s_2;\ m_2) :: \vec{x}_2 \,.\, t \mid T_1/(m_2 \propto \vec{x}_2) \simeq T_1' \wedge t \in T_1' \parallel t_2 \,\} \end{aligned}\right\} \quad \text{if } \mathrm{cc}(m_1) \neq \mathrm{cc}(m_2).$$

This case is well-defined because the lengths of T_1' and T_2' are respectively at most the lengths of T_1 and T_2. Our trace reduction operator hides considerable complexity, so we unpack the definition.

First, observe that the commuting case ensures that if a synchronization between $(s_1; m_1)$ and an element of T_2 is eventually possible, then it occurs. Indeed, if $\mathrm{cc}(m_1)$ appears free in T_2 with an observable sign, then $T_2/(m_1 \propto \vec{x}_1)$ is undefined (see, e.g., example 5.2). As a result, the first set (prefixing $(s_1; m_1)$) is empty, and we interleave T_1 with the tail of T_2 in the second set to eventually synchronize $(s_1; m_1)$ with the corresponding element of T_2.

Example 5.3. The interleaving $(!;\ \mathsf{close}\ \mathsf{on}\ a) :: \varepsilon \parallel (?;\ \mathsf{close}\ \mathsf{on}\ b) :: (?;\ \mathsf{close}\ \mathsf{on}\ a) :: \varepsilon$ is determined by the commuting case. The reduction in the first set of the clause is undefined, so the first set is empty. The second, $\{ (?;\ \mathsf{close}\ \mathsf{on}\ b) :: t \mid t \in ((!;\ \mathsf{close}\ \mathsf{on}\ a) :: \varepsilon) \parallel ((?;\ \mathsf{close}\ \mathsf{on}\ a) :: \varepsilon) \}$, is defined in terms of a principal case. This principal case captures the synchronization on a, and the interleavings are given by $\{ (?;\ \mathsf{close}\ \mathsf{on}\ b) :: (\sigma;\ \mathsf{close}\ \mathsf{on}\ a) :: \varepsilon \}$.

Second, the commuting case detects deadlock. Indeed, if two traces attempt to synchronize on different channels in opposite orders, then the carrier of the head of one trace is free in the other trace. This implies that the reduction in each set is undefined, resulting in no interleavings:

$$((?;\ \mathsf{close}\ \mathsf{on}\ a) :: (!;\ \mathsf{close}\ \mathsf{on}\ b) :: \varepsilon) \parallel ((?;\ \mathsf{close}\ \mathsf{on}\ b) :: (!;\ \mathsf{close}\ \mathsf{on}\ a) :: \varepsilon) = \emptyset.$$

Third, the commuting case uses reduction to handle constraint satisfaction. Indeed, if $(s_1; m_1)$ satisfies a constraint in T_2, then the reduction $T_2/(m_1 \propto \vec{x}_1)$ deletes that constraint before interleaving the result with the tail t_1.

Example 5.4. Interleaving $((!;\ \mathsf{close}\ \mathsf{on}\ a) :: \varepsilon) \parallel ((?;\ \mathsf{close}\ \mathsf{on}\ b) :: (¿;\ \mathsf{close}\ \mathsf{on}\ a) :: \varepsilon)$ produces the interleavings $\{ (!;\ \mathsf{close}\ \mathsf{on}\ a) :: (?;\ \mathsf{close}\ \mathsf{on}\ b) :: \varepsilon,\quad (?;\ \mathsf{close}\ \mathsf{on}\ b) :: (!;\ \mathsf{close}\ \mathsf{on}\ a) :: \varepsilon \}$, where the send on a in the first trace satisfies the constraint in the second trace.

Finally, the commuting case ensures that constraint satisfaction obeys synchronization boundaries. For example, consider interleaving the traces $(¿;\ \mathsf{close}\ \mathsf{on}\ a) :: (?;\ \mathsf{close}\ \mathsf{on}\ b) :: \varepsilon$ and $(!;\ \mathsf{close}\ \mathsf{on}\ b) :: (!;\ \mathsf{close}\ \mathsf{on}\ a) :: \varepsilon$. Intuitively, they synchronize on b. However, the constraint on a cannot be satisfied by the send on a, because the constraint is imposed before the synchronization, while the send occurs after the synchronization on b. This unsatisfiability implies that the traces should have no synchronizing interleavings. This is captured by our interleaving operator: the only applicable clause is the commuting case, and both sets defining it are empty.

All other cases are taken to be ill-defined, so we define their set of interleavings to be empty. These include, e.g., cases of the form $((!;\ \mathsf{close}\ \mathsf{on}\ a) :: t) \parallel ((!;\ \mathsf{label}\ l\ \mathsf{on}\ a) :: t')$ (sending on the same channel) and mismatched sends and receives: $((!;\ \mathsf{close}\ \mathsf{on}\ a) :: t) \parallel ((?;\ \mathsf{label}\ l\ \mathsf{on}\ a) :: t')$.

A trace is **safely constrained** if any carrier channel associated with a constraint sign is not also associated with input, output, or synchronization. By proposition 5.5, constraints can only limit the interleavings of observable actions. In other words, checking constraints when interleaving traces does not introduce new observable behaviours. The opposite inclusion will play a key role when typechecking process composition:

Proposition 5.5. *Given safely constrained traces t_1 and t_2, $(t_1 \parallel t_2)/¿ \subseteq (t_1/¿) \parallel (t_2/¿)$.*

5.5 Type-Theoretic Properties

Section 4.2 gave a trace semantics for process specifications. We relate the behaviour of trace operators to this trace assignment. Recall that a process specification $\Pi \mathbin{\#} \mathcal{G}$ is well-formed only if every channel appearing free in a type in $\Pi, \mathcal{G}$ is typed by some other type assignment in the specification.

Proposition 5.6 semantically characterizes hiding internal channels in a specification. The well-formedness hypothesis is required to ensure that no types become ill-scoped. For example, it precludes deleting $a : \mathbf{1}$ from $\Pi \mathbin{\#} (\Delta \mid \mathrm{I}, a : \mathbf{1} \vdash c : \mathrm{CASE}\ a\ \{ \mathsf{close} \Rightarrow C \})$.

Proposition 5.6. *If $t \in [\![\Pi \mathbin{\#} \mathcal{G} \mid (\Delta \mid \mathrm{I}, \Gamma \vdash a : A)]\!]$ and $\Pi \mathbin{\#} \mathcal{G} \mid (\Delta \mid \mathrm{I} \vdash a : A)$ is well-formed, then $(t \setminus \mathrm{dom}(\Gamma)) \in [\![\Pi \mathbin{\#} \mathcal{G} \mid (\Delta \mid \mathrm{I} \vdash a : A)]\!]$.*

The following proposition characterizes independent parallel executions. To concretize it, consider specifications $\Pi, \Delta_2, \mathrm{I}_2, a_2 : A_2 \# (\Delta_1 \mid \mathrm{I}_1 \vdash a_1 : A_1)$ and $\Pi, \Delta_1, \mathrm{I}_1, a_1 : A_1 \# (\Delta_2 \mid \mathrm{I}_2 \vdash a_2 : A_2)$. They specify two processes whose types can observe each other's channels. We can compose these two specifications in a single specification: $\Pi \# (\Delta_1 \mid \mathrm{I}_1 \vdash a_1 : A_1) \mid (\Delta_2 \mid \mathrm{I}_2 \vdash a_2 : A_2)$. The proposition states that interleaving the traces of each individual specification results in traces allowed by the composed specification. This proposition is useful when characterizing traces involving channel transmissions. More generally, where $\overline{\mathcal{G}}$ denotes the typed channels in $\mathcal{G}$:

Proposition 5.7. *If* $t_1 \in [\![\Pi, \overline{\mathcal{G}_2} \# \mathcal{G}_1]\!]$ *and* $t_2 \in [\![\Pi, \overline{\mathcal{G}_1} \# \mathcal{G}_2]\!]$, *then* $t_1 \parallel t_2 \subseteq [\![\Pi \# \mathcal{G}_1 \mid \mathcal{G}_2]\!]$.

An alternate form of specification composition results in synchronization. Indeed, consider processes specifications $\Pi, \Delta_2, \mathrm{I}_2, c : C \# (\Delta_1 \mid \mathrm{I}_1 \vdash a : A)$ and $\Pi, \Delta_1, \mathrm{I}_1 \# (\Delta_2, a : A \mid \mathrm{I}_2 \vdash c : C)$. The parallel composition operation composes them along the channel a so that they synchronize on this channel. Subject to side conditions, their composition satisfies $\Pi \# (\Delta_1, \Delta_2 \mid \mathrm{I}_1, a : A, \mathrm{I}_2 \vdash c : C)$ where a becomes an inner synchronization channel. This synchronizing parallel composition is also characterized by synchronized interleaving:

Proposition 5.8. *If* $t_1 \in [\![\Pi, \Delta_2, \mathrm{I}_2, c : C \# (\Delta_1 \mid \mathrm{I}_1 \vdash a : A)]\!]$ *and* $t_2 \in [\![\Pi, \Delta_1, \mathrm{I}_1 \# (\Delta_2, a : A \mid \mathrm{I}_2 \vdash c : C)]\!]$, *then* $t_1 \parallel t_2 \subseteq [\![\Pi \# (\Delta_1, \Delta_2 \mid \mathrm{I}_1, a : A, \mathrm{I}_2 \vdash c : C)]\!]$.

6 TYPECHECKING PROCESSES

We give a typechecking algorithm to statically check that a process satisfies its specification, i.e., that each execution of the process is permitted by its specification. It is driven by a collection of inference rules and operates in two alternating phases.

The first phase decomposes process compositions into individual processes. It then determines the next channel a that a given process will use to communicate, and kicks off the second phase. The second phase focusses on the type of a. It first tries to reduce the focussed type to weak head normal form, and then proceeds to typecheck the given process with respect to that weak head normal type. Because our system is compositional, types may depend on ambient channels whose meaning will eventually be given by composition, and type reduction may cause constraints to be imposed on ambient communications. To capture these, our algorithm generates a collection constraints that must be satisfied by the environment in order for the process to be well-typed.

6.1 Phase One: Uniform Process Typing

The first phase, which we call uniform typing, is driven by a judgment $P \Vdash \Pi \# (\Delta \mid \mathrm{I} \vdash a : A) \,/\!\!/\, \mathfrak{T}$ inductively defined by the rules of fig. 1. We use colours to indicate the modes of each parameter in the judgment, where red indicates inputs to the judgment and blue indicates outputs. Here, we attempt to check process P against the well-formed specification $\Pi \# (\Delta \mid \mathrm{I} \vdash a : A)$. The judgment outputs a set $\mathfrak{T}$ of traces that consist of the communications of P interleaved with constraints on channels in Π. By our soundness result (theorem 7.1), restricting each trace in $\mathfrak{T}$ to its observable actions (i.e., those with sign ?, !, or σ) will result in the set $[\![P]\!]$ of traces of P. We implicitly assume that process specifications are well-formed, i.e., that all names are in scope.

The rule (New) introduces internal channels that can be used to type process P. The constraint set for its conclusion is obtained by deleting the new channels from the constraint set for P. This is analogous to the trace semantics of hiding in section 3. Because we assume that all specifications are well-formed, we know that no types in $\Pi \# (\Delta \mid \mathrm{I} \vdash c : C)$ attempt to observe $a_1, \ldots, a_n$.

The rule (Par) ensures that two processes can be composed only if they impose compatible constraints. This is ensured by premise $\mathfrak{T}_1 \therefore \mathfrak{T}_2$ that holds if and only if:

$$\forall t_1 \in \mathfrak{T}_1 \,.\, \forall t_2 \in \mathfrak{T}_2 \,.\; (t_1/¿ \parallel t_2/¿) = (t_1 \parallel t_2)\,/¿. \tag{21}$$

$\boxed{P \Vdash \Pi \mathbin{\#} (\Delta \shortmid \mathrm{I} \vdash a : A) \sslash \mathfrak{T}}$ Process P satisfies specification $\Pi \mathbin{\#} (\Delta \shortmid \mathrm{I} \vdash a : A)$ subject to constraints $\mathfrak{T}$

$$\frac{P \Vdash \Pi \mathbin{\#} (\Delta \shortmid \mathrm{I}, a_1 : A_1, \dots, a_n : A_n \vdash c : C) \sslash \mathfrak{T}}{\nu(a_1 : A_1, \dots, a_n : A_n) . P \Vdash \Pi \mathbin{\#} (\Delta \shortmid \mathrm{I} \vdash c : C) \sslash \mathfrak{T} \setminus \{ a_1, \dots, a_n \}}\ (\textsc{New})$$

$$\frac{P \Vdash \Pi, \Delta_2, \mathrm{I}_2, c : C \mathbin{\#} (\Delta_1 \shortmid \mathrm{I}_1 \vdash a : A) \sslash \mathfrak{T}_1 \quad Q \Vdash \Pi, \Delta_1, \mathrm{I}_1 \mathbin{\#} (\Delta_2, a : A \shortmid \mathrm{I}_2 \vdash c : C) \sslash \mathfrak{T}_2 \quad \mathfrak{T}_1 \therefore \mathfrak{T}_2}{P \parallel_a Q \Vdash \Pi \mathbin{\#} (\Delta_1, \Delta_2 \shortmid \mathrm{I}_1, a : A, \mathrm{I}_2 \vdash c : C) \sslash \mathfrak{T}_1 \parallel \mathfrak{T}_2}\ (\textsc{Par})$$

$$\frac{P \Rightarrow \Pi \mathbin{\#} (\Delta \shortmid \mathrm{I} \vdash [a : A]) \sslash \mathfrak{T} \quad \mathsf{princ}(P) = a}{P \Vdash \Pi \mathbin{\#} (\Delta \shortmid \mathrm{I} \vdash a : A) \sslash \mathfrak{T}}\ ({\Vdash}\mathrm{R})$$

$$\frac{P \Rightarrow \Pi \mathbin{\#} (\Delta, [a : A] \shortmid \mathrm{I} \vdash c : C) \sslash \mathfrak{T} \quad \mathsf{princ}(P) = a}{P \Vdash \Pi \mathbin{\#} (\Delta, a : A \shortmid \mathrm{I} \vdash c : C) \sslash \mathfrak{T}}\ ({\Vdash}\mathrm{L})$$

Fig. 1. Uniform Process Typing Judgment

Intuitively, it specifies that if traces t_1 and t_2 corresponding to P and Q can be composed as process executions, then each synchronized interleaving of t_1 and t_2 qua process traces can be obtained by synchronizing t_1 and t_2 as constraint traces. In other words, we can interleave the constraints in t_1 and t_2 such that all of the constraints are consistent with the process actions in t_1 and t_2, and recover all valid process interleavings. The key point of this definition is that we can recover all valid process interleavings despite the presence of constraints, i.e., that $(t_1/¿ \parallel t_2/¿) \subseteq (t_1 \parallel t_2)\,/¿$; we already know by proposition 5.5 that adding constraints cannot produce new process interleavings, i.e., that $(t_1/¿ \parallel t_2/¿) \supseteq (t_1 \parallel t_2)\,/¿$.

Example 6.1. We illustrate how constraint satisfaction rules out incompatible compositions by attempting to compose the following two processes (respectively, P and Q) in parallel:

$$\mathsf{case}\ a\ \{ l \Rightarrow b.\mathtt{0};\ \mathsf{wait}\ a;\ \mathsf{close}\ b \}_{l \in \mathsf{Bit}} \Vdash \cdot \mathbin{\#} (a : \mathsf{Bits}_1 \shortmid \cdot \vdash b : \mathsf{Bits}_1) \sslash \mathfrak{T}_1$$

$$\mathsf{case}\ a\ \{ l \Rightarrow b.l;\ \mathsf{wait}\ a;\ \mathsf{close}\ b \}_{l \in \mathsf{Bit}} \Vdash a : \mathsf{Bits}_1 \mathbin{\#} (b : \mathsf{Bits}_1 \shortmid \cdot \vdash c : \mathsf{IdBitSeq}_1\ a) \sslash \mathfrak{T}_2.$$

Their composition should not satisfy $P \parallel_a Q \Vdash \cdot \mathbin{\#} (a : \mathsf{Bits}_1 \shortmid b : \mathsf{Bits}_1 \vdash c : \mathsf{IdBitSeq}_1\ a) \sslash \mathfrak{T}$ because an input of 1 on a does not result in the output 1 on c specified by $\mathsf{IdBitSeq}_1\ a$. The following two traces violate (21) (they have no interleavings) and keep (Par) from being applied:

$$(?;\ \mathsf{label}\ \mathtt{1}\ \mathsf{on}\ a) :: (!;\ \mathsf{label}\ \mathtt{0}\ \mathsf{on}\ b) :: (?;\ \mathsf{close}\ \mathsf{on}\ a) :: (!;\ \mathsf{close}\ \mathsf{on}\ b) :: \varepsilon \in \mathfrak{T}_1$$

$$(?;\ \mathsf{label}\ \mathtt{0}\ \mathsf{on}\ b) :: (¿;\ \mathsf{label}\ \mathtt{0}\ \mathsf{on}\ a) :: (?;\ \mathsf{label}\ \mathtt{0}\ \mathsf{on}\ c) :: \cdots \in \mathfrak{T}_2$$

The rules (⊩R) and (⊩L) uniformly identify the next channel a on which a process will communicate, and kick off the second phase that reduces the type of a. This channel a is called the **principal channel** of the process, and is given by $\mathsf{princ}(P) = a$ for

$$P \coloneqq \mathsf{close}\ a \mid \mathsf{wait}\ a;\ Q \mid a.k;\ Q \mid \mathsf{case}\ a\ \{ l \Rightarrow Q_l \}_{l \in L} \mid \mathsf{send}\ a\ (b.Q_1);\ Q_2 \mid b \leftarrow \mathsf{receive}\ a;\ Q.$$

We remark that there is exactly one rule in the first phase for each process forming construct.

6.2 Phase Two: Focussed Type Reduction

The second phase is driven by a pair of type-reduction judgments $P \Rightarrow \Pi \mathbin{\#} (\Delta \shortmid \mathrm{I} \vdash [a : A]) \sslash \mathfrak{T}$ and $P \Rightarrow \Pi \mathbin{\#} (\Delta, [a : A] \shortmid \mathrm{I} \vdash c : C) \sslash \mathfrak{T}$ that reduce a focussed type $[a : A]$ to weak head normal form, and checks that if the principal channel of P is a, then P communicates on a according to A. The right-focussed judgment is inductively defined by the rules of fig. 2 (weak head normal cases) and fig. 3 (reduction cases); the analogous left-focussed judgment exchanges sending and receiving.

$\boxed{P \Rightarrow \Pi \mathbin{\#} (\Delta \mid \mathrm{I} \vdash [a : A]) \mathbin{/\!/} \mathfrak{T}}$ Focussed reduction of $a : A$ on the right

$$\frac{}{\text{close } a \Rightarrow \Pi \mathbin{\#} (\cdot \mid \cdot \vdash [a : \mathbf{1}]) \mathbin{/\!/} \{ (!;\ \text{close on } a) :: \varepsilon \}}\ (\Rightarrow\mathbf{1}\mathrm{R})$$

$$\frac{P \Vdash \Pi/\pi \mathbin{\#} (\Delta/\pi \mid \mathrm{I}/\pi \vdash a : A_k) \mathbin{/\!/} \mathfrak{T} \qquad \pi = \text{label } k \text{ on } a}{a.k;\ P \Rightarrow \Pi \mathbin{\#} (\Delta \mid \mathrm{I} \vdash [a : \oplus_{l \in L \cup \{k\}} A_l]) \mathbin{/\!/} (!;\ \text{label } k \text{ on } a) :: \mathfrak{T}}\ (\Rightarrow\oplus\mathrm{R})$$

$$\frac{P_l \Vdash \Pi/\pi \mathbin{\#} (\Delta/\pi \mid \mathrm{I}/\pi \vdash a : A_l) \mathbin{/\!/} \mathfrak{T}_l \qquad \pi = \text{label } l \text{ on } a \qquad (\forall l \in L)}{\text{case } a \{ l \Rightarrow P_l \}_{l \in L \cup L'} \Rightarrow \Pi \mathbin{\#} (\Delta \mid \mathrm{I} \vdash [a : \&_{l \in L} A_l]) \mathbin{/\!/} \bigcup_{l \in L} (?;\ \text{label } k \text{ on } a) :: \mathfrak{T}_l}\ (\Rightarrow\&\mathrm{R})$$

$$\frac{\begin{array}{l} P \Vdash (\Pi, \Delta_2, \mathrm{I}_2)/\pi, a : A \mathbin{\#} (\Delta_1/\pi \mid \mathrm{I}_1/\pi \vdash b : B) \mathbin{/\!/} \mathfrak{T}_1 \\ Q \Vdash (\Pi, \Delta_1, \mathrm{I}_1)/\pi, b : B \mathbin{\#} (\Delta_2/\pi \mid \mathrm{I}_2/\pi \vdash a : A) \mathbin{/\!/} \mathfrak{T}_2 \qquad \pi = \text{chan } b \text{ on } a \quad \mathfrak{T}_1 \therefore \mathfrak{T}_2 \end{array}}{\text{send } a\ (b.P);\ Q \Rightarrow \Pi \mathbin{\#} (\Delta_1, \Delta_2 \mid \mathrm{I}_1, \mathrm{I}_2 \vdash [a : (b : B) \otimes (a : A)]) \mathbin{/\!/} (!;\ \text{chan on } a) :: b\,.\,(\mathfrak{T}_1 \parallel \mathfrak{T}_2)}\ (\Rightarrow\otimes\mathrm{R})$$

$$\frac{P \Vdash \Pi/\pi \mathbin{\#} (\Delta/\pi, b : B \mid \mathrm{I}/\pi \vdash a : A) \mathbin{/\!/} \mathfrak{T} \qquad \pi = \text{chan } b \text{ on } a}{b \leftarrow \text{receive } a;\ P \Rightarrow \Pi \mathbin{\#} (\Delta \mid \mathrm{I} \vdash [a : (b : B) \multimap (a : A)]) \mathbin{/\!/} (?;\ \text{chan on } a) :: b\,.\,\mathfrak{T}}\ (\Rightarrow\multimap\mathrm{R})$$

Fig. 2. Right-Focussed Process Typing — Focussed Type in Weak Head Normal Form

$$\frac{P \Rightarrow \Pi/\pi \mathbin{\#} (\Delta/\pi \mid \mathrm{I}/\pi \vdash [a : A]) \mathbin{/\!/} \mathfrak{T} \qquad \pi = \text{close on } c}{P \Rightarrow \Pi, c : \mathbf{1} \mathbin{\#} (\Delta \mid \mathrm{I} \vdash [a : \text{CASE } c \{ \text{close} \Rightarrow A \}]) \mathbin{/\!/} (¿;\ \text{close on } c) :: \mathfrak{T}}\ (\Rightarrow\mathbf{1}[\mathrm{R}])$$

$$\frac{P \Rightarrow \Pi/\pi_l, c : C_l \mathbin{\#} (\Delta/\pi_l \mid \mathrm{I}/\pi_l \vdash [a : A_l]) \mathbin{/\!/} \mathfrak{T}_l \quad \pi_l = \text{label } l \text{ on } c \quad (\forall l \in L) \quad L \subseteq L' \subseteq L'' \quad \odot \in \{ \oplus, \& \}}{P \Rightarrow \Pi, c : \odot_{l \in L'} C_l \mathbin{\#} (\Delta \mid \mathrm{I} \vdash [a : \text{CASE } c \{ l \Rightarrow A_l \}_{l \in L''}]) \mathbin{/\!/} \bigcup_{l \in L} (¿;\ \text{label } l \text{ on } c) :: \mathfrak{T}_l}\ (\Rightarrow\oplus\&[\mathrm{R}])$$

$$\frac{P \Rightarrow \Pi/\pi, b : B, c : C \mathbin{\#} (\Delta/\pi \mid \mathrm{I}/\pi \vdash [a : A]) \mathbin{/\!/} \mathfrak{T} \qquad \pi = \text{chan } b \text{ on } c \qquad \odot \in \{ \otimes, \multimap \}}{P \Rightarrow \Pi, c : (b : B) \odot (c : C) \mathbin{\#} (\Delta \mid \mathrm{I} \vdash [a : \text{CASE } c \{ \langle b \rangle \Rightarrow A \}]) \mathbin{/\!/} (¿;\ \text{chan on } c) :: b\,.\,\mathfrak{T}}\ (\Rightarrow\otimes\multimap[\mathrm{R}])$$

Fig. 3. Right-Focussed Process Typing — Reducing the Type in Focus

For example, the rule $(\Rightarrow\mathbf{1}\mathrm{R})$ of fig. 2 specifies that the process close a communicates according to the weak head normal type $a : \mathbf{1}$, provided it has empty client and internal contexts. The client context must be empty to ensure that we do not accidentally discard any clients. The internal context must be empty to ensure that we use all internal channels in the specification. We must do so, for the current process may have been spawned by a larger process (typed by channels in Π) that observes channels in I, and we must not allow these channels to be discarded. Because no constraints are imposed on the environment, the output constraint set is exactly $[\![\text{close } a]\!]$.

The rule $(\Rightarrow\oplus\mathrm{R})$ specifies that a process $a.k;\ P$ is a server for a channel $a : \oplus_{l \in L \cup \{k\}} A_l$ provided P is well-typed when $a : A_k$ and the remainder of the specification has been reduced by label k on a. This reduction ensures that all types that depend on a observe the transmitted label k. The resulting constraint set is given by prefixing constraint set $\mathfrak{T}$ for P by the output action (!, label k on a).

Channel transmission send a $(b.P)$; Q spawns a process P that provides a fresh channel b, sends b over a, and continues as Q. Processes P and Q cannot share any channels. This is ensured by $(\Rightarrow\otimes\mathrm{R})$: it treats all channels used or provided by P as ambient in Q's typing judgment and vice-versa. The rule also ensures that their specifications impose compatible constraints, analogously to (Par).

Sometimes, the focussed channel is not yet weak head normal. In these cases, we must reduce it to a weak head normal type (see fig. 3). The rule $(\Rightarrow\mathbf{1}[\mathrm{R}])$ reduces a focussed channel of type CASE c { close $\Rightarrow A$ }. This reduction is justified so long as a close message is observed on ambient channel c. This constraint is captured by prefixing the constraint ($¿$, close on c) on the constraint set $\mathfrak{T}$ generated by the premise. The rule ($\Rightarrow\oplus\&$[R]) can generate constraints for a subset of labels in a choice to support composition with processes that only send a subset of the permitted labels. The remaining reduction rules are analogous. We remark that the rules of fig. 3 only reduce types that observe ambient channels. A focussed type that observes a used or provided channel cannot be reduced because we cannot reduce types based on future communications. Indeed, the observation required to reduce the type can only be provided by the process being checked, but the process will communicate on the focussed principal channel before it will communicate on the channel being observed. Types that observe local channels are instead immediately reduced by the rules of fig. 2.

Example 6.2. Set $C = \text{CASE } a \{ \text{close} \Rightarrow \mathbf{1} \}$ and $P_{x,y} = (\text{wait } x;\ \text{close } y)$. We illustrate the mechanics of our typechecking algorithm by attempting to typecheck $\nu(b : \mathbf{1}) . (P_{a,b} \parallel_b P_{b,c})$ against the specification $\cdot \# (a : \mathbf{1} \mid \cdot \vdash c : C)$. Typechecking succeeds if we can build a derivation of $\nu(b : \mathbf{1}) . (P_{a,b} \parallel_b P_{b,c}) \Vdash \cdot \# (a : \mathbf{1} \mid \cdot \vdash c : C) /\!\!/ \mathfrak{T}$ for some $\mathfrak{T}$. First, we build a candidate derivation by considering only the inputs to the judgment (the red parts of the tree). The only possible bottom rule in our case is (New), and we use proof search to complete the derivation. The focussed nature of our system ensures that search amounts to inversion: in each case, at most one rule can be used to extend the derivation. Eventually each branch either gets stuck or reaches a leaf (an axiom). The following candidate derivation shows that we can successfully build the red portion of the tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\text{close } b \Rightarrow c : \mathbf{1} \# (\cdot \mid \cdot \vdash [b : \mathbf{1}]) /\!\!/ \mathcal{L}_1}\;(\Rightarrow\mathbf{1}\text{R})}{\text{close } b \Rightarrow c : \mathbf{1} \# (\cdot \mid \cdot \vdash b : \mathbf{1}) /\!\!/ \mathcal{L}_1}\;(\Vdash\text{R})}{P_{a,b} \Rightarrow c : C \# ([a : \mathbf{1}] \mid \cdot \vdash b : \mathbf{1}) /\!\!/ \mathcal{L}_2}\;(\Rightarrow\mathbf{1}\text{L})}{P_{a,b} \Vdash c : C \# (a : \mathbf{1} \mid \cdot \vdash b : \mathbf{1}) /\!\!/ \mathcal{L}_2}\;(\Vdash\text{L}) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\text{close } c \Vdash \cdot \# (\cdot \mid \cdot \vdash [c : \mathbf{1}]) /\!\!/ \mathcal{R}_1}\;(\Rightarrow\mathbf{1}\text{R})}{\text{close } c \Vdash a : \mathbf{1} \# (\cdot \mid \cdot \vdash [c : C]) /\!\!/ \mathcal{R}_2}\;(\Rightarrow\mathbf{1}[\text{R}])}{\text{close } c \Vdash a : \mathbf{1} \# (\cdot \mid \cdot \vdash c : C) /\!\!/ \mathcal{R}_2}\;(\Vdash\text{R})}{P_{b,c} \Rightarrow a : \mathbf{1} \# ([b : \mathbf{1}] \mid \cdot \vdash c : C) /\!\!/ \mathcal{R}_3}\;(\Rightarrow\mathbf{1}\text{L})}{P_{b,c} \Vdash a : \mathbf{1} \# (b : \mathbf{1} \mid \cdot \vdash c : C) /\!\!/ \mathcal{R}_3}\;(\Vdash\text{L})}{P_{a,b} \parallel_b P_{b,c} \Vdash \cdot \# (a : \mathbf{1} \mid b : \mathbf{1} \vdash c : C) /\!\!/ \mathcal{L}_2 \parallel \mathcal{R}_3}\;(\text{Par})}{\nu(b : \mathbf{1}) . (P_{a,b} \parallel_b P_{b,c}) \Vdash \cdot \# (a : \mathbf{1} \mid \cdot \vdash c : C) /\!\!/ (\mathcal{L}_2 \parallel \mathcal{R}_3) \setminus \{ b \}}\;(\text{New})$$

Assuming every branch reaches a leaf, the axioms specify the leaf's judgment's output (the trace set in blue). We thread these trace sets back down through the derivation tree, using the operations specified in each rule to build that rules output trace set. In this case:

$$\mathcal{L}_1 = \{ (!;\ \text{close on } b) :: \varepsilon \} \qquad \mathcal{L}_2 = (?;\ \text{close on } a) :: \mathcal{L}_1$$
$$\mathcal{R}_1 = \{ (!;\ \text{close on } c) :: \varepsilon \} \qquad \mathcal{R}_2 = (¿;\ \text{close on } a) :: \mathcal{R}_1 \qquad \mathcal{R}_3 = (?;\ \text{close on } b) :: \mathcal{R}_2$$
$$\mathcal{L}_2 \parallel \mathcal{R}_3 = \{ (?;\ \text{close on } a) :: (\sigma;\ \text{close on } b) :: (!;\ \text{close on } c) :: \varepsilon \}$$
$$(\mathcal{L}_2 \parallel \mathcal{R}_3) \setminus \{ b \} = \{ (?;\ \text{close on } a) :: (!;\ \text{close on } c) :: \varepsilon \}$$

Typechecking succeeds if the resulting derivation is valid (including the side condition (21) for (Par)).

Our rules determine a terminating typechecking algorithm when all choice types are indexed by finite label sets. Indeed, all constraint sets and their traces are finite, so trace operations terminate, and the alternating phases reduce the complexity of processes and their specifications. Most of the typechecking complexity lies in computing operations on traces. We conjecture that they could efficiently be implemented by representing traces as tries or radix trees.

7 SAFETY PROPERTIES

Our typechecking algorithm is sound: if we can typecheck a process against a process specification, then all executions of that process are permitted by its specification. Put differently, our typechecking algorithm ensures that well-typed processes communicate safely:

Theorem 7.1 (Soundness). *The typechecking algorithm is sound: if any of*

(1) $P \Vdash \Pi \# (\Delta \mid \mathrm{I} \vdash a : A) /\!\!/ \mathfrak{T}$,

(2) $P \Rightarrow \Pi \mathbin{\#} (\Delta \mid \mathrm{I} \vdash [a : A]) \mathbin{/\!\!/} \mathfrak{T}$*, or*
(3) $P \Rightarrow \Pi \mathbin{\#} (\Delta, [a : A] \mid \mathrm{I} \vdash c : C) \mathbin{/\!\!/} \mathfrak{T}$*,*

then $\mathfrak{T}/\text{¿} = [\![P]\!]$ *(the set* $\mathfrak{T}$ *extents the executions of* P *with constraints) and* $\mathfrak{T} \subseteq [\![\Pi \mathbin{\#} \Delta \mid \mathrm{I} \vdash a : A]\!]$ *(the constraint set* $\mathfrak{T}$ *satisfies the specification* $\Pi \mathbin{\#} \Delta \mid \mathrm{I} \vdash a : A$*).*

Corollary 7.2. *If* $P \Vdash \cdot \mathbin{\#} (\Delta \mid \mathrm{I} \vdash a : A) \mathbin{/\!\!/} \mathfrak{T}$*, then* $\mathfrak{T} = [\![P]\!]$ *and* $[\![P]\!] \subseteq [\![\cdot \mathbin{\#} (\Delta \mid \mathrm{I} \vdash a : A)]\!]$.

A key feature of Most is its modular approach to process specification. Processes whose specifications impose no constraints on their environment can always be composed:

Proposition 7.3 (Compositionality). *If* $P \Vdash \cdot \mathbin{\#} (\Delta_1 \mid \mathrm{I}_1 \vdash a : A) \mathbin{/\!\!/} \mathfrak{T}_1$ *and* $Q \Vdash \cdot \mathbin{\#} (\Delta_2, a : A \mid \mathrm{I}_2 \vdash c : C) \mathbin{/\!\!/} \mathfrak{T}_2$*, then* $P \parallel_a Q \Vdash \cdot \mathbin{\#} (\Delta_1, \Delta_2 \mid \mathrm{I}_1, a : A, \mathrm{I}_2 \vdash c : C) \mathbin{/\!\!/} \mathfrak{T}_1 \parallel \mathfrak{T}_2$.

8 EXTENDING MOST

Most provides a foundation for the development of session-typed languages with rich specifications. In this section, we conjecturally sketch extensions along several axes that allow Most to capture richer computational phenomena: selection, asynchrony, value dependency, and recursion.

8.1 Selection

Go [The Go Project 2024] includes a *select* statement that, given a collection of communication actions to perform, randomly performs one from the set of actions that are possible. For example, if processes P and Q attempt to communicate with a process R, then process R could use a select statement to communicate with the first of the two processes that is ready. We show how to extend Most with a linear variant of a select statement building on ideas from differential linear logic [Ehrhard 2018] and its applications to concurrency [Rocha and Caires 2021, 2023]. Differential linear logic offers a means of *adding* proofs of the same sequent. In the intuitionistic case, if we ascribe the sequents of intuitionistic linear logic with proof terms (processes), we get the rule

$$\frac{\Gamma \vdash P :: a : A \quad \Gamma \vdash Q :: a : A}{\Gamma \vdash P + Q :: a : A}$$

The possible behaviours of $P + Q$ are all those of P or Q, so the sum denotes $[\![P + Q]\!] = [\![P]\!] \cup [\![Q]\!]$. This clause satisfies the identities described by Rocha and Caires, namely, that process composition distributes over sums and that all processes are idempotent with regard to the sum, i.e., that $P{+}P \equiv P$ for all P. Our discussion suggests the following typing rule for sums of Most processes:

$$\frac{P_1 \Vdash \Pi \mathbin{\#} (\Delta \mid \mathrm{I} \vdash a : A) \mathbin{/\!\!/} \mathcal{T}_1 \quad P_2 \Vdash \Pi \mathbin{\#} (\Delta \mid \mathrm{I} \vdash a : A) \mathbin{/\!\!/} \mathcal{T}_2}{P_1 + P_2 \Vdash \Pi \mathbin{\#} (\Delta \mid \mathrm{I} \vdash a : A) \mathbin{/\!\!/} \mathcal{T}_1 \cup \mathcal{T}_2} \ (\textsc{Sum})$$

Returning to our motivating example, suppose that we want to receive channels a and c over b and d, respectively, but that we do not want to impose a particular order on which of b or d is used first. The following sum of processes receives on whichever of b or d is ready first, non-deterministically breaking any ties, and communicates over the other before continuing as R':

$$(a \leftarrow \mathsf{receive}\ b;\ c \leftarrow \mathsf{receive}\ d;\ R') + (c \leftarrow \mathsf{receive}\ d;\ a \leftarrow \mathsf{receive}\ b;\ R').$$

8.2 Asynchrony

Adapting Most to use an asynchronous communication semantics is straightforward, but it comes at the cost of technical complexity. The intended semantics treats channels as unbounded FIFO buffers, where processes can always send (add a message to a buffer), and they can receive a message if one is buffered. We implement FIFO buffers using *continuation channels*, where each channel carries exactly one message, and each message consists of the datum plus the name of the (continuation) channel that will carry the next message. To preclude interference between channels, continuation channel names are globally fresh. To illustrate, consider the trace that describes sending on a the label l, receiving label k, and closing a. In the synchronous setting, we simply send the three messages on a, giving a trace $(!;\ \mathsf{label}\ l\ \mathsf{on}\ a) :: (?;\ \mathsf{label}\ k\ \mathsf{on}\ a) :: (!;\ \mathsf{close}\ \mathsf{on}\ a) :: \varepsilon$. In the asynchronous setting, the label l is paired with a fresh name a_1 when sent on a; label k is paired with a fresh name a_2 and sent on a_1; and the close message is sent on a_2. To enforce global freshness, ensure that trace sets remain finite, and preserve compositionality, we model continuation channels in traces by bound channel names. The corresponding trace is $(!;\ \mathsf{label}\ l\ \mathsf{on}\ a) :: a_1 . (?;\ \mathsf{label}\ k\ \mathsf{on}\ a_1) :: a_2 . (!;\ \mathsf{close}\ \mathsf{on}\ a_2) :: \varepsilon$, where a_1 and a_2 are bound. Modifying the semantic clauses to capture asynchronous communication is then a matter of adapting the trace operations of section 5 to handle continuation channels, and of threading bindings for continuation channels throughout the semantic clauses. For example, eq. (3) becomes $[\![a.k;\ P]\!] = (!;\ \mathsf{label}\ k\ \mathsf{on}\ a) :: a . [\![P]\!]$, where a is bound in the traces in $[\![P]\!]$. To type processes in the asynchronous setting, we must extend the syntax of observed communications π to include the name of the continuation channel, and extend type reductions A/π to substitute the continuation channel for the carrier of π in A. Because type reduction is localized to the operation A/π and all reductions occur in lockstep in the typing rules, we never need to worry about a continuation channel name going out of scope, and no other changes are required.

8.3 Value-Dependent Session Types

Most can be extended to support value-dependent session types [Toninho, Caires, et al. 2011], where session types can depend on values drawn from an underlying dependent functional language. Value-dependent type formers $\forall x : \tau . A$ and $\exists x : \tau . A$ respectively specify servers that receive or send values v of type τ and then communicate according to $[v/x]A$. We first extend our syntax:

Processes	$P ::= \cdots$	$\mid$	$\mathsf{send}\ M\ \mathsf{on}\ a;\ P$	Evaluate term M and send its value on a
		$\mid$	$\mathsf{let}\ x = \mathsf{rcv}\ a\ \mathsf{in}\ P$	Bind received value to x in P
Session types	$A ::= \cdots$	$\mid$	$\forall x : \tau . A$	Dependent value receiving
		$\mid$	$\exists x : \tau . A$	Dependent value sending
		$\mid$	$\mathsf{CASE}\ a\ \{\mathsf{val}\ x \Rightarrow A\}$	Value observation

Value observation $\mathsf{CASE}\ a\ \{\mathsf{val}\ x \Rightarrow A\}$ observes a transmitted value v on a channel a and reduces to the type $[v/x]A$. Its reduction is defined analogously to the other immediate reductions:

$$(\mathsf{CASE}\ c\ \{\mathsf{val}\ x \Rightarrow A\})/\pi = \begin{cases} [M/x]A & \text{if } \pi = \mathsf{val}\ M\ \mathsf{on}\ c \\ \mathsf{CASE}\ c\ \{\mathsf{val}\ x \Rightarrow A/\pi\} & \text{if } \mathsf{cc}(\pi) \neq c \end{cases}$$

Write $\mathbf{val}[\tau]$ for the sets of closed values of type τ. We give meaning to value transmission by allowing terms to be embedded in traces:[3]

$$[\![\mathsf{send}\ M\ \mathsf{on}\ a;\ P]\!] = (!;\ \mathsf{val}\ M\ \mathsf{on}\ a) :: [\![P]\!], \qquad [\![\mathsf{let}\ x = \mathsf{rcv}\ a\ \mathsf{in}\ P]\!] = \bigcup_{v \in \mathbf{val}[\tau]} (?;\ \mathsf{val}\ v\ \mathsf{on}\ a) :: [\![P]\!].$$

We then extend interleaving to use the underlying definitional equality on terms so that whenever M and v are definitionally equal, $((!;\ \mathsf{val}\ M\ \mathsf{on}\ a) :: t_1) \parallel ((?;\ \mathsf{val}\ v\ \mathsf{on}\ a) :: t_2) = (\sigma;\ \mathsf{val}\ v\ \mathsf{on}\ a) :: (t_1 \parallel t_2)$

To type processes that use values, we assume a typing judgment $\Gamma \vdash_{\mathsf{f}} M : \tau$ for the functional language and extend the process typing judgment to include Γ. The right process typing rules follow a similar pattern as for label transmission, and the other rules follow analogously:

$$\frac{P \Vdash \Gamma, x : \tau;\ \Pi/\pi \mathbin{\#} (\Delta/\pi \mid \mathrm{I}/\pi \vdash a : A) \mathbin{/\!\!/} \mathcal{T} \qquad \pi = \mathsf{val}\ x\ \mathsf{on}\ a}{\mathsf{let}\ x = \mathsf{rcv}\ a\ \mathsf{in}\ P \Rightarrow \Gamma;\ \Pi \mathbin{\#} (\Delta \mid \mathrm{I} \vdash [a : \forall x : \tau . A]) \mathbin{/\!\!/} \bigcup_{v \in \mathbf{val}[\tau]} (?;\ \mathsf{val}\ v\ \mathsf{on}\ a) :: [v/x]\mathcal{T}}\ (\Rightarrow\forall\mathrm{R})$$

$$\frac{P \Vdash \Gamma;\ \Pi/\pi \mathbin{\#} (\Delta/\pi \mid \mathrm{I}/\pi \vdash a : [M/x]A) \mathbin{/\!\!/} \mathcal{T} \qquad \Gamma \vdash_{\mathsf{f}} M : \tau \qquad \pi = \mathsf{val}\ M\ \mathsf{on}\ a}{\mathsf{send}\ M\ \mathsf{on}\ a;\ P \Rightarrow \Gamma;\ \Pi \mathbin{\#} (\Delta \mid \mathrm{I} \vdash [a : \exists x : \tau . A]) \mathbin{/\!\!/} (!;\ \mathsf{val}\ M\ \mathsf{on}\ a) :: \mathcal{T}}\ (\Rightarrow\exists\mathrm{R})$$

[3]Extending our semantics to capture evaluation in the underlying dependent type theory is beyond the scope of this sketch.

$$\frac{P \Vdash \Gamma, x : \tau;\ \Pi/\pi \ \#\ (\Delta/\pi \mid \mathrm{I}/\pi \vdash a : A) \mathbin{/\!\!/} \mathcal{T} \qquad \pi = \mathsf{val}\ x\ \mathsf{on}\ a \qquad \sharp \in \{\forall, \exists\}}{P \Rightarrow \Gamma;\ \Pi, c : (\sharp x : \tau.C)\ \#\ (\Delta \mid \mathrm{I} \vdash [a : \mathsf{CASE}\ c\ \{\, \mathsf{val}\ x \Rightarrow A \,\}]) \mathbin{/\!\!/} \bigcup_{v \in \mathbf{val}[\tau]} (¿;\ \mathsf{val}\ v\ \mathsf{on}\ a) :: [v/x]\mathcal{T}} \ (\Rightarrow\forall\exists[\mathrm{R}])$$

8.4 Recursion

Most can be extended to support mutually recursive types and processes using coinductively defined typing judgments. We implement mutually recursive processes using process declarations $f(\vec{x}) = P_f$ gathered in an implicit global context Σ_P. These declarations specify that the process named f is given by its implementation P_f involving channels $\vec{x}$. The syntax call $f(\vec{a})$ calls process f with channel names $\vec{a}$, and it corresponds to running $[\vec{a}/\vec{x}]P_f$. Equirecursive types are given by type declarations $X[\vec{x}] = A_X$ gathered in an implicit global context Σ_A. Channel name variables $\vec{x}$ specify the channels that A_X can observe, and $X[\vec{a}]$ is definitionally equal to $[\vec{a}/\vec{x}]A_X$. A complete program $\mathbb{P}$ is a called process name paired with explicit declarations for each called name.

Processes	$P ::= \cdots \mid \mathsf{call}\ f(\vec{a})$	Call named process f with channels $\vec{a}$
Session types	$A ::= \cdots \mid X[\vec{x}]$	Equirecursive type names
Type declarations	$\Sigma_A ::= \cdot \mid \Sigma_A, X[\vec{x}] = A_X$	$X[\vec{x}]$ is definitionally equal to A_X
Process declarations	$\Sigma_P ::= \cdot \mid \Sigma_P, f(\vec{x}) = P_f$	Process named f is implemented by P_f
Programs	$\mathbb{P} ::= \mathsf{call}\ f(\vec{a})\ \mathsf{where}\ \Sigma_P$	Complete or top-level program

We assume that recursive declarations are contractive [Pierce 2002, p. 300]. This standard assumption simplifies our theory and allows us to interpret equirecursive types as regular trees. To ensure type reduction A/π remains well defined, we interpret its definition coinductively instead of inductively.

Example 8.1. For each fixed type A, the type $\mathsf{list}_A = \oplus\{\, \mathsf{nil} : \mathbf{1}, \mathsf{cons} : A \otimes \mathsf{list}_A \,\}$ specifies lists of channels of type A. The type $\mathsf{stream}_A = \&\{\, \mathsf{head} : A, \mathsf{tail} : \mathsf{stream}_A \,\}$ specifies a stream of channels of type A. Given a process declaration $\mathsf{u}(x) = (\mathsf{case}\ x\ \{\, \mathsf{head} \Rightarrow \mathsf{close}\ x \mid \mathsf{tail} \Rightarrow \mathsf{call}\ \mathsf{u}(x) \,\})$, the process "call $\mathsf{u}(a)$" is a closed process providing $a : \mathsf{stream}_{\mathbf{1}}$.

Example 8.2. The types Bits and IdBitSeq a and the process id from sections 2.1 and 2.2 are encoded by the following declarations, where we use the label \$ and type $\mathbf{1}$ to specify finite sequences:

$$\begin{aligned}
\mathsf{Bits} &= \oplus\{\, \$: \mathbf{1}, \mathtt{0} : \mathsf{Bits}, \mathtt{1} : \mathsf{Bits} \,\} \\
\mathsf{IdBitSeq}[x] &= \mathsf{CASE}\ x\ \{\, \$ \Rightarrow \oplus\{\, \$: \mathsf{CASE}\ x\ \{\, \mathsf{close} \Rightarrow \mathbf{1} \,\}\,\} \\
&\qquad\qquad \mid \mathtt{0} \Rightarrow \oplus\{\, \mathtt{0} : \mathsf{IdBitSeq}[x] \,\} \mid \mathtt{1} \Rightarrow \oplus\{\, \mathtt{1} : \mathsf{IdBitSeq}[x] \,\}\,\} \\
\mathsf{id}(x, y) &= \mathsf{case}\ x\ \{\, \$ \Rightarrow \mathsf{wait}\ x;\ \mathsf{close}\ y \mid \mathtt{0} \Rightarrow y.\mathtt{0};\ \mathsf{call}\ \mathsf{id}(x, y) \mid \mathtt{1} \Rightarrow y.\mathtt{1};\ \mathsf{call}\ \mathsf{id}(x, y) \,\}.
\end{aligned}$$

We extend our semantics to support calling named processes. Let $\mathcal{P}$ be the set of declared calling interfaces $f(\vec{x})$. A process environment is a map $\rho : \mathcal{P} \to \mathbf{TSets}$ giving the meaning of declared processes, where **TSets** is the lattice of sets of traces ordered by inclusion. We generalize $[\![P]\!]$ from an element of **TSets** to a continuous map $[\![P]\!] : (\mathcal{P} \to \mathbf{TSets}) \to \mathbf{TSets}$. Calling $f(\vec{x})$ with channels $\vec{a}$ means retrieving its meaning from the environment and instantiating it with the names $\vec{a}$:

$$[\![\mathsf{call}\ f(\vec{a})]\!]\rho = \{\, [\vec{a}/\vec{x}]t \mid t \in \rho(f(\vec{x})) \,\}.$$

We adapt the existing clauses by threading through the environments, e.g.: $[\![P \|_a Q]\!]\rho = [\![P]\!]\rho \| [\![Q]\!]\rho$. To capture parallel composition in the presence of potentially infinite traces, we must extend $\|$ to be a fair merge operator. This ensures that each element from each input traces appears at a finite depth in the interleavings. We omit the details due to space constraints.

Complete programs are processes executed in an environment where every called process name has a corresponding declaration. Their semantics are given by

$$\begin{aligned}
&[\![\mathsf{call}\ f(\vec{a})\ \mathsf{where}\ \Sigma_P]\!] = [\![\mathsf{call}\ f(\vec{a})]\!][\![\Sigma_P]\!], \text{ where} \\
&[\![f_1(\vec{x}_1) = P_1, \ldots, f_n(\vec{x}_n) = P_n]\!] = \mathsf{FIX}(\lambda\rho \in (\mathcal{P} \to \mathbf{TSets}).\ [\rho \mid f_1(\vec{x}_1) \mapsto [\![P_1]\!]\rho \mid \cdots \mid f_n(\vec{x}_n) \mapsto [\![P_n]\!]\rho]),
\end{aligned}$$

FIX is the continuous fixed-point operator on $\mathcal{P} \to \mathbf{TSets}$, and $[g \mid y_1 \mapsto v_1 \mid \cdots \mid y_n \mapsto v_n]\,(x)$ is v_i if $x = y_i$ and $g(x)$ otherwise. Intuitively, $[\![\Sigma_P]\!]$ maps each $f_i(\vec{x}_i)$ to the meaning of $[\![P_i]\!]$ in the fixed point giving meaning to the mutually recursive declarations.

Example 8.3. If $\Sigma_P = (f(x) = x.1;\ \mathsf{call}\ g(x)),\ \ (g(x) = x.0;\ \mathsf{call}\ f(x))$, then

$$[\![\Sigma_P]\!] = \mathsf{FIX}\,(\lambda\rho.\,[\rho \mid f \mapsto (!;\ \mathsf{label}\ 1\ \mathsf{on}\ x) :: \rho(g) \mid g \mapsto (!;\ \mathsf{label}\ 0\ \mathsf{on}\ x) :: \rho(f)])$$
$$[\![\mathsf{call}\ f(a)\ \mathsf{where}\ \Sigma_P]\!] = \{\,(!;\ \mathsf{label}\ 1\ \mathsf{on}\ a) :: (!;\ \mathsf{label}\ 0\ \mathsf{on}\ a) :: (!;\ \mathsf{label}\ 1\ \mathsf{on}\ a) :: \cdots\}.$$

Process specifications still denote sets $[\![\Pi \mathbin{\#} \mathcal{G}]\!]$ of traces satisfying the specification. However, to capture recursive behaviour, we adapt its definition to use a coinductively defined elementhood relation. Elementhood is given by a coinductive reading of the rules defining $t \in [\![\Pi \mathbin{\#} \mathcal{G}]\!]$ generated by the obvious analogs of section 4.2, e.g.:

$$\frac{t \in [\![\Pi/\pi \mathbin{\#} \mathcal{G}/\pi]\!] \quad \pi = \mathsf{close\ on}\ a}{(!;\ \mathsf{close\ on}\ a) :: t \in [\![\Pi \mathbin{\#} \mathcal{G} \mid (\cdot \mid \cdot \vdash a : \mathbf{1})]\!]} \qquad \frac{t \in [\![\Pi/\pi \mathbin{\#} \mathcal{G}/\pi \mid (\Delta/\pi \mid \mathrm{I}/\pi \vdash a : A_k)]\!] \quad \pi = \mathsf{label}\ k\ \mathsf{on}\ a}{(!;\ \mathsf{label}\ k\ \mathsf{on}\ a) :: t \in [\![\Pi \mathbin{\#} \mathcal{G} \mid (\Delta \mid \mathrm{I} \vdash a : \oplus_{l \in L} A_l)]\!]}$$

In order to type mutually recursive processes, we give our typing rules a coinductive reading and treat each component as an input to the judgment. Then, we extend our rules with:

$$\frac{[\vec{a}/\vec{x}]P_f \Vdash \Pi \mathbin{\#} (\Delta \mid \mathrm{I} \vdash c : C) \mathbin{/\!\!/} \mathcal{T} \quad (f(\vec{x}) = P_f) \in \Sigma_P}{\mathsf{call}\ f(\vec{a}) \Vdash \Pi \mathbin{\#} (\Delta \mid \mathrm{I} \vdash c : C) \mathbin{/\!\!/} \mathcal{T}}\ (\textsc{Call})$$

This system can be given a finitary presentation by moving from arbitrary coinductive derivations to circular derivations (finite derivations with loops) [Derakhshan and Pfenning 2022; Fortier and Santocanale 2013]. We refer the reader to [Somayyajula and Pfenning 2022] for a discussion of how to do so to achieve typechecking in finite time, and also a discussion of how to encode inductive and coinductive types using value dependency and (general) equirecursive types.

9 RELATED WORK

Most builds on linear-logical foundations of session types pioneered by Caires and Pfenning [2010]. Indeed, the simply typed fragment of our system is based on a proofs-as-processes interpretation of multiplicative-additive intuitionistic linear logic (MAILL), where propositions correspond to binary session types that specify communications on communication channels. A key difference with our work is that such interpretations typically combine parallel composition and hiding into a single operation. This is because process composition corresponds to the cut rule of linear logic. In contrast, we have separate parallel composition and hiding operators, and we track internal channels. This separation ensures that composition does not hide channels observed by specifications, thereby ensuring that specifications remain well-defined after process composition. We can recover cut-style process composition with a derived rule. Our uniform process typing system can then be seen as a conservative extension of typical interpretations of identity-free MAILL. Another key difference is our use of focussing. Processes can be specified with observing types, but we can only check specific communication actions (e.g., sending a label) against weak head normal types (e.g., an internal choice). This means that to typecheck a process, we must reduce the type of its principal channel to weak head normal form, potentially generating constraints on ambient communications along the way. Our focussing approach ensures that we only reduce the types of principal channels. This avoids the premature or spurious generation of constraints on ambient channels that could affect constraint compatibility in process composition, and it ensures that typechecking is deterministic.

There are various dependent type systems for binary session types. Toninho, Caires, et al. [2011] and Toninho and Yoshida [2018] interpret logical quantifiers as value-dependent session types in languages with functional programming features. As discussed in section 8.3, value-dependent session types depend on transmitted functional values. Variations include label-dependent session types [Thiemann and Vasconcelos 2019], which reify labels as a first-class type, and arithmetic refinements [Das and Pfenning 2020], which offer type-level arithmetic using transmitted natural numbers. Like other binary session types, value-dependent session types take a channel-local approach: a type only considers communication on a single channel. In contrast, types in Most specify protocols on individual channels while potentially depending on ambient communications.

Multiparty session types [Honda, Yoshida, et al. 2016] are an alternative approach to specifying systems of communicating processes. Instead of specifying communications on individual channels, multiparty session types specify interactions between multiple parties using a global type, and then for each party project a local type specifying its interactions with the other parties. Though multiparty session types can specify complex interactions, they are not compositional and typically require that entire systems be specified at once. To achieve compositionality, Stolze et al. [2023] introduced *partial multiparty sessions types*, which specify multiparty sessions where some parties may be missing. Determining when partial sessions can be composed and the resulting composition is technically complex. In contrast, Most is designed to ensure compositional process specifications, and the composition of process specifications is determined by a handful of inference rules.

The dynamics of processes in proofs-as-processes interpretations of intuitionistic linear logic is often given by multiset rewriting semantics [Cervesato and Scedrov 2009]. We instead describe the behaviour of processes using a trace semantics. We do so to simplify relating the meaning of processes (a set of traces) with the meaning of process specifications (a set of allowed traces). It also provides a means of ensuring that processes and specifications are mutually compatible when typechecking their composition. We conjecture that our trace semantics captures the same observable communication behaviours [Atkey 2017; Kavanagh 2022] as multiset-based semantics. It is challenging to define trace semantics or ordered semantics for process calculi with name generation or α-conversion. Some approaches, e.g., typed event structures for π-calculi [Varacca and Yoshida 2010], specify in advance all names that will be created; others use complex semantic composition operators involving renamings [Crafa et al. 2007]. In contrast, we use binding to range over all possible fresh names in a manner reminiscent of higher-order abstract syntax. This avoids unintended name clashes when interleaving traces during process composition. Our compact presentation also minimizes the number of traces considered by our typechecking algorithm. Traces with binding are similar to nominal sequences [Gabbay and Ghica 2012], which use a coabstraction operator to bind atoms appearing later in a sequence. In contrast, our traces bind not atoms, but names found in later messages. As a result, our traces more closely resemble abstract binding trees.

10 CONCLUSION

In this work we describe Most, a message-observing binary session type system. It uses type-level processes to specify protocols that vary based on ambient communications. This allows us to express more precise safety guarantees that cannot be expressed by prior work.

Our main goal in designing Most is to ensure compositionality. We achieve this using a novel trace-based semantics that uses traces with bindings to compactly handle higher-order communications. Our main technical results are type safety and compositionality for well-typed processes.

In the future, we plan to extend Most with shared channels. This would allow Most to capture shared data structures like shared queues, and shared services like databases.

ACKNOWLEDGMENTS

This work was funded in part by a Natural Sciences and Engineering Research Council of Canada Postdoctoral Fellowship (awarded to the first author) and Discovery Grant (grant number 206263).

REFERENCES

Robert Atkey. 2017. “Observed Communication Semantics for Classical Processes”. In: *Programming Languages and Systems* (Lecture Notes in Computer Science) 10201. 26th European Symposium on Programming. ESOP 2017 (Uppsala, Sweden, Apr. 22–29, 2017). Ed. by Hongseok Yang. Springer-Verlag GmbH Germany, Berlin, Germany, 56–82. ISBN: 978-3-662-54434-1. https://doi.org/10.1007/978-3-662-54434-1_3.

Stephanie Balzer and Frank Pfenning. Sept. 2017. “Manifest Sharing With Session Types”. *Proceedings of the ACM on Programming Languages*, 1, ICFP, (Sept. 2017), 37. https://doi.org/10.1145/3110281.

Luís Caires and Frank Pfenning. 2010. “Session Types as Intuitionistic Linear Propositions”. In: *CONCUR 2010 — Concurrency Theory* (Lecture Notes in Computer Science) 6269. 21st International Conference, CONCUR 2010 (Paris, France, Aug. 31–Sept. 3, 2010). Ed. by Paul Gastin and François Laroussinie. Springer-Verlag Berlin Heidelberg, 222–236. ISBN: 978-3-642-15374-7. https://doi.org/10.1007/978-3-642-15375-4_16.

Luís Caires, Frank Pfenning, and Bernardo Toninho. Mar. 2016. “Linear Logic Propositions As Session Types”. *Mathematical Structures in Computer Science*, 26, 3, (Mar. 2016), 367–423. *Behavioural Types Part 2*. https://doi.org/10.1017/s0960129514000218.

Iliano Cervesato and Andre Scedrov. Oct. 2009. “Relating State-Based and Process-Based Concurrency Through Linear Logic (Full-Version)”. *Information and Computation*, 207, 10, (Oct. 2009), 1044–1077. *Special Issue: 13th Workshop on Logic, Language, Information and Computation (WoLLIC 2006)*. https://doi.org/10.1016/j.ic.2008.11.006.

Silvia Crafa, Daniele Varacca, and Nobuko Yoshida. 2007. “Compositional Event Structure Semantics for the Internal π-Calculus”. In: *CONCUR 2007 — Concurrency Theory* (Lecture Notes in Computer Science) 4703. 18th International Conference on Concurrency Theory. CONCUR 2007 (Lisbon, Portugal, Sept. 3–8, 2007). Ed. by Luís Caires and Vasco T. Vasconcelos. Springer-Verlag Berlin Heidelberg, 317–332. ISBN: 978-3-540-74407-8. https://doi.org/10.1007/978-3-540-74407-8_22.

Ankush Das and Frank Pfenning. Aug. 26, 2020. “Session Types With Arithmetic Refinements”. In: *31st International Conference on Concurrency Theory (CONCUR 2020)* (Leibniz International Proceedings in Informatics (LIPIcs)). 31st International Conference on Concurrency Theory. CONCUR 2020 (Vienna, Austria (Virtual Conference), Sept. 1–4, 2020). Ed. by Igor Konnov and Laura Kovács. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Saarbrücken/Wadern, Germany, (Aug. 26, 2020). ISBN: 978-3-95977-160-3. https://doi.org/10.4230/LIPIcs.CONCUR.2020.13.

Pierre-Malo Deniélou and Nobuko Yoshida. 2011. “Dynamic Multirole Session Types”. In: *POPL'11*. 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'11 (Austin, Texas, Jan. 26–28, 2011). Association for Computing Machinery, Inc., New York, New York, 435–446. ISBN: 978-1-4503-0490-0. https://doi.org/10.1145/1926385.1926435.

Farzaneh Derakhshan and Frank Pfenning. May 10, 2022. “Circular Proofs As Session-Typed Processes: A Local Validity Condition”. *Logical Methods in Computer Science*, 18, 2, (May 10, 2022), 8. https://doi.org/10.46298/lmcs-18(2:8)2022.

Thomas Ehrhard. Aug. 2018. “An Introduction To Differential Linear Logic: Proof-Nets, Models and Antiderivatives”. *Mathematical Structures in Computer Science*, 28, 7, (Aug. 2018), 995–1060. *Differential Linear Logic, Nets and Other Quantitative and Parallel Approaches to Proof-Theory*. https://doi.org/10.1017/s0960129516000372.

Jérôme Fortier and Luigi Santocanale. Sept. 2, 2013. “Cuts for Circular Proofs: Semantics and Cut-Elimination”. In: *Computer Science Logic 2013* (Leibniz International Proceedings in Informatics (LIPIcs)) 23. Computer Science Logic 2013. CSL'13 (Torino, Italy, Sept. 2–5, 2013). Ed. by Simona Ronchi Della Rocca. European Association for Computer Science Logic. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Saarbrücken/Wadern, Germany, (Sept. 2, 2013), 248–262. ISBN: 978-3-939897-60-6. https://doi.org/10.4230/LIPIcs.CSL.2013.248.

Murdoch Gabbay and Dan Ghica. Sept. 24, 2012. “Game Semantics in the Nominal Model”. *Electronic Notes in Theoretical Computer Science*, 286, (Sept. 24, 2012), 173–189. *Proceedings of the 28th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVIII)*. https://doi.org/10.1016/j.entcs.2012.08.012.

Kohei Honda. 1993. “Types for Dyadic Interaction”. In: *CONCUR'93* (Lecture Notes in Computer Science) 715. 4th International Conference on Concurrency Theory (Hildesheim, Germany, Aug. 23–26, 1993). Ed. by Eike Best. Springer-Verlag Berlin Heidelberg, Berlin, 509–523. ISBN: 978-3-540-47968-0. https://doi.org/10.1007/3-540-57208-2_35.

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. “Language Primitives and Type Discipline for Structured Communication-Based Programming”. In: *Programming Languages and Systems* (Lecture Notes in Computer Science) 1381. 7th European Symposium on Programming. ESOP'98 (Lisbon, Portugal, Mar. 28–Apr. 4, 1998). Ed. by Chris Hankin. Joint European Conferences on Theory and Practice of Software, ETAPS'98. Springer-Verlag Berlin Heidelberg, 122–138. ISBN: 978-3-540-69722-0. https://doi.org/10.1007/BFb0053567.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. Mar. 3, 2016. “Multiparty Asynchronous Session Types”. *Journal of the ACM*, 63, 1, (Mar. 3, 2016), 9. https://doi.org/10.1145/2827695.

Ryan Kavanagh. May 2022. “Fairness and Communication-Based Semantics for Session-Typed Languages”. *Information and Computation*, 285, B, (May 2022), 104892. https://doi.org/10.1016/j.ic.2022.104892.

Ryan Kavanagh and Brigitte Pientka. Mar. 8, 2024. *Message-Observing Sessions*. (Mar. 8, 2024). arXiv: 2403.04633 [cs.PL].

Benjamin Pierce. 2002. *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts. xxi+623 pp. ISBN: 0-262-16209-1.

Pedro Rocha and Luís Caires. Aug. 2021. "Propositions-As-Types and Shared State". *Proceedings of the ACM on Programming Languages*, 5, ICFP, (Aug. 2021), 79. https://doi.org/10.1145/3473584.

Pedro Rocha and Luís Caires. 2023. "Safe Session-Based Concurrency With Shared Linear State". In: *Programming Languages and Systems* (Lecture Notes in Computer Science) 13990. 32nd European Symposium on Programming. ESOP 2023 (Paris, France, Apr. 22–27, 2023). Ed. by Thomas Wies. Springer, Cham, Switzerland, 421–450. ISBN: 978-3-031-30044-8. https://doi.org/10.1007/978-3-031-30044-8_16.

Siva Somayyajula and Frank Pfenning. 2022. "Type-Based Termination for Futures". In: *7th International Conference on Formal Structures for Computation and Deduction* (Leibniz International Proceedings in Informatics (LIPIcs)). 7th International Conference on Formal Structures for Computation and Deduction. FSCD 2022 (Haifa, Israel, Aug. 2–5, 2022). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Saarbrücken/Wadern, Germany. ISBN: 978-3-95977-233-4. https://doi.org/10.4230/LIPIcs.FSCD.2022.12.

Claude Stolze, Marino Miculan, and Pietro Di Gianantonio. Apr. 2023. "Composable Partial Multiparty Session Types for Open Systems". *Software and Systems Modeling*, 22, 2, (Apr. 2023), 473–494. https://doi.org/10.1007/s10270-022-01040-x.

Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. "An Interaction-Based Language and Its Typing System". In: *PARLE'94. Parallel Architectures and Languages Europe* (Lecture Notes in Computer Science) 10201. 6th International PARLE Conference (Athens, Greece, July 4–8, 1994). Ed. by Costas Halatsis, Dimitrios Maritsas, George Philokyprou, and Sergios Theodoridis. Springer-Verlag Berlin Heidelberg, Berlin, 398–413. ISBN: 978-3-540-48477-6. https://doi.org/10.1007/3-540-58184-7_118.

The Go Project. Feb. 6, 2024. *The Go Programming Language Specification*. Language version go1.22. (Feb. 6, 2024). Retrieved Mar. 5, 2024 from https://perma.cc/436Q-ZL6J.

Peter Thiemann and Vasco T. Vasconcelos. Dec. 2019. "Label-Dependent Session Types". *Proceedings of the ACM on Programming Languages*, 4, POPL, (Dec. 2019), 67. https://doi.org/10.1145/3371135.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2011. "Dependent Session Types Via Intuitionistic Linear Type Theory". In: *PPDP'11*. 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming. PPDP'11 (Odense, Denmark, July 20–22, 2011). Association for Computing Machinery, Inc., New York, New York, 161–172. ISBN: 978-1-4503-0776-5. https://doi.org/10.1145/2003476.2003499.

Bernardo Toninho and Nobuko Yoshida. 2018. "Depending on Session-Typed Processes". In: *Foundations of Software Science and Computation Structures* (Lecture Notes in Computer Science) 10803. 21st International Conference, FOSSACS 2018 (Thessaloniki, Greece, Apr. 14–20, 2018). Ed. by Christel Baier and Ugo Dal Lago. European Joint Conferences on Theory and Practice of Software. SpringerOpen, Cham, Switzerland, 128–145. ISBN: 978-3-319-89366-2. https://doi.org/10.1007/978-3-319-89366-2_7.

Daniele Varacca and Nobuko Yoshida. Apr. 6, 2010. "Typed Event Structures and the Linear π-calculus". *Theoretical Computer Science*, 411, 19, (Apr. 6, 2010), 1949–1973. *Mathematical Foundations of Programming Semantics (MFPS 2006)*. https://doi.org/10.1016/j.tcs.2010.01.024.

Philip Wadler. Jan. 31, 2014. "Propositions As Sessions". *Journal of Functional Programming*, 24, 2-3, (Jan. 31, 2014), 384–418. https://doi.org/10.1017/s095679681400001x.

Received 2023-10-21; accepted 2024-02-24