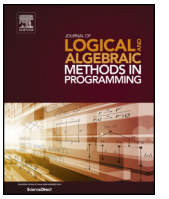
Journal of Logical and Algebraic Methods in Programming 124 (2022) 100731

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp



Session-typed concurrent contracts

Hannah Gommerstadt [b,*], Limin Jia [a], Frank Pfenning [a]

[a] Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA, 15213, USA
[b] Vassar College, 124 Raymond Avenue, Poughkeepsie, NY, 12604, USA

ARTICLE INFO

Article history:
Received 1 November 2020
Received in revised form 15 October 2021
Accepted 18 October 2021
Available online 22 October 2021

Keywords:
Contracts
Session types
Monitors

ABSTRACT

In sequential languages, dynamic contracts are usually expressed as boolean functions without externally observable effects, written within the language. We propose an analogous notion of concurrent contracts for languages with session-typed message-passing concurrency. Concurrent contracts are partial identity processes that monitor the bidirectional communication along channels and raise an alarm if a contract is violated. Concurrent contracts are session-typed in the usual way and must also satisfy a transparency requirement, which guarantees that terminating compliant programs with and without the contracts are observationally equivalent. We illustrate concurrent contracts with several examples. We also show how to generate contracts from a refinement session-type system and show that the resulting monitors are redundant for programs that can statically be seen to be well-typed.

1. Introduction

Contracts, specifying the conditions under which software components can safely interact, have been used for ensuring key properties of programs for decades. Recently, contracts for distributed processes have been studied in the context of session types [1,2]. These contracts can enforce the communication protocols, specified as session types, between processes. In this setting, we can assign each channel a monitor for detecting whether messages observed along the channel adhere to the prescribed session type. The monitor can then detect any deviant behavior the processes exhibit and trigger alarms. However, contracts based solely on session types are inherently limited in their expressive power. Many contracts that we would like to enforce cannot even be stated using session types alone. As a simple example, consider a "factorization service" which may be sent a (possibly large) integer x and is supposed to respond with a list of prime factors. Session types can only express that the request is an integer and the response is a list of integers, which is insufficient.

In this paper, we show that by generalizing the class of monitors beyond those derived from session types, we can enforce, for example, that multiplying the numbers in the response yields the original integer x. This paper focuses on monitoring more expressive contracts, specifically those that cannot be expressed with session types, or even refinement types.

To handle these contracts, we have designed a model where our monitors execute as transparent processes alongside the computation. They are able to maintain internal state which allows us to check complex properties, which we explore in our examples. These monitoring processes act as partial identities, which do not affect the computation except possibly raising an alarm, and merely observe the messages flowing through the system. They then perform whatever computation is

* Corresponding author.
E-mail address: hgommerstadt@vassar.edu (H. Gommerstadt).

https://doi.org/10.1016/j.jlamp.2021.100731
2352-2208/

```
factor_monitor : {factor_t ← factor_t}
c ← factor_monitor ← d =
  n ← recv c ; shift ← recv c ; send d n ; send d shift ;
  p ← recv d ; q ← recv d ; assert l (n = p ∗ q) ; send c p ; send c q ;
  wait d ; close c
```

Fig. 1. Factoring monitor.

needed, for example, they can compute the product of the factors, to determine whether the messages are consistent with the contract. If the message is not consistent, they stop the computation and blame the process responsible for the mistake. To show that our contracts subsume refinement-based contracts, we encode refinement types in our model by translating refinements into monitors. This encoding allows us to prove a blame (safety) theorem stating that monitors that enforce a less precise refinement type than the type of the process being monitored will not raise alarms. Unfortunately, the blame theory for the general model is challenging because not all contracts can be expressed as types.

The main contributions of this paper are:

- A novel approach to contract checking using partial-identity monitors, which are processes that do not affect the computation except to possibly raise an alarm
- A method for verifying that monitors are partial identities, and a proof that the method is correct
- Examples illustrating the contracts that our monitors can enforce
- A translation from refinement types to our monitoring processes and a blame theorem for this fragment

The rest of this paper is organized as follows. We first show a range of example contracts in Section 2. Next, we review the background on session types in Section 3. In Section 4, we show how to check that a monitor process is a partial identity and prove the method correct. We then show how we can encode refinements in our system in Section 5. We discuss related work in Section 6. This paper expands on Gommerstadt et al. [3] by including additional monitoring examples in Section 2 and more detailed metatheory in Section 5.

2. Contract examples

As a first simple example, let us consider a process that factors integers. The factor process receives a positive integer n over a channel and factors it into two integers p and q that are sent back over the same channel. We can model the communication pattern of factor with a session type. The factor_t type indicates that after an integer n is received, first, an integer p is sent, then, an integer q is sent, followed by the process terminating. The type-level operator $\uparrow$ marks where the direction of communication changes and the process shifts from receiving the value n to sending the values p and q. The code below implements the factor process which is providing a factoring service over the channel d. We note that session types only model process communication, we do not concern ourselves with the internals of the factoring process.

```
factor_t : {∀n. ↑ ∃p.∃q.1}
factor : {factor_t}
d ← factor =
  n ← recv d ; shift ← recv d;
  //factoring algorithm
  send d p ; send d q ; close d

c ← factor_good =
  d ← factor ; c ← factor_monitor ← d
```

For the factor process, we would like to enforce that $n = p * q$. Our approach involves implementing a monitoring process factor_monitor (shown in Fig. 1) that will check the required contract. In the code above, we implement the factor_good process, which first spawns a factor process on channel d. It then spawns a factor_monitor process which uses channel d and provides the monitored factoring service over the channel c.

The factor_monitor process first receives an integer n over channel c. The monitor then receives a shift message over the channel c. It then sends n and the shift message over channel d. The monitor now receives the integers p and q over channel d. It now asserts that the product of p and q is indeed n. If the assertion fails, the monitor terminates with the label l. Otherwise, the monitor sends p and q over channel c. Finally, the monitor terminates. The factor_monitor has type {factor_t ← factor_t} because it takes as argument a factoring service and produces a monitored factoring service.

In the rest of this section, we present a variety of monitoring processes that can enforce various contracts. Our monitors must be transparent, that is, they cannot change the computation. We accomplish this by making them act as partial identities (this is described in more detail in Section 4). Therefore, any monitor that enforces a contract on a channel must peel off each layer of the type a step at a time (by sending or receiving over the channel as dictated by the type), perform the required checks on values or labels, and then reconstruct the original type (again, by sending or receiving as appropriate). Since a minimal number of shifts can be inferred during elaboration of the syntax [4], we suppress it in most examples.

```
pos_mon : {list ← list}
a ← pos_mon ← b =
  case b of
  | nil ⇒ a.nil ; wait b ; close a
  | cons ⇒ x ← recv b ; assert l (x > 0) ; a.cons ; send a x ; a ← pos_mon ← b
```

Fig. 2. Integer refinement.

```
empty_t = ⊕{nil : 1}
empty_mon : {empty_t ← list}
a ← empty_mon ← b =
  case b of
  | nil ⇒ wait b ; a.nil ; close a
  | cons ⇒ abort l
```

```
non_empty_t = ⊕{cons : ∃n:int.list}
nonempty_mon : {non_empty_t ← list}
a ← nonempty_mon ← b =
  case b of
  | nil ⇒ abort l
  | cons ⇒ x ← recv b ; a.cons ;
    send a x ; a ← b
```

Fig. 3. Label refinement.

```
add : {list ← list, list}
r ← add ← m n =
  case m of
  | nil ⇒ n' ← empty_mon ← n ;
    case n' of | nil ⇒ r.nil ; wait n' ; wait m ; close r
  | cons ⇒ n' ← nonempty_mon ← n ;
    case n' of | cons ⇒ y ← recv n' ; r.cons ; send r (x + y) ; r ← add ← m n
```

Fig. 4. List addition example.

Our examples will mainly involve lists. We model a list of integers as follows:

$$\oplus\{\mathsf{nil} : \mathbf{1}; \mathsf{cons} : \exists.n : \mathsf{int}.\mathsf{list}\}$$

A list is an internal choice of two labels, a nil and a cons. One of these labels is first sent over a channel. In the case of nil, the list then terminates. In the case of cons, the list sends an integer over the channel and then recurs.

Refinement The simplest kind of monitoring process we can write is one that models a refinement of an integer type (shown in Fig. 2); for example, a process that checks whether every element in the list is positive. If the list is empty, the monitor receives the nil label along channel b, sends it along to channel a, and terminates. If the list is nonempty, the monitor first receives the cons label along channel b. Then, the monitor receives the head of the list from b, checks whether it is positive (if yes, the process continues, if not it aborts), and then sends the value along to reconstruct the monitored list a. Finally, the monitor implements a recursive call to check the rest of the list.

Our monitors can also exploit information contained in the labels present in the external and internal choices. We show two examples of monitors that model label refinement in Fig. 3. The empty_mon process checks whether the list offered at b is empty and aborts if b sends the label cons. Similarly, the nonempty_mon monitor checks whether the list offered at b is not empty and aborts if b sends the label nil. We show how these two monitors can then be used by a process that adds two lists element by element and aborts if they are of different lengths in Fig. 4. These two monitors enforce refinements: $\{\mathsf{nil}\} \subseteq \{\mathsf{nil}, \mathsf{cons}\}$ and $\{\mathsf{cons}\} \subseteq \{\mathsf{nil}, \mathsf{cons}\}$. We discuss refinement types further in Section 5.

Monitors with internal state We now move beyond refinement contracts, and model contracts that have to maintain some internal state. We first present a monitor that checks whether a set of right and left parentheses match (shown in Fig. 5). The match_mon monitor uses its internal state to push every left parenthesis it sees on its stack and to pop it off when it sees a right parenthesis. For brevity, we model our list of parentheses by marking every left parenthesis with a 1 and right parenthesis with a -1. So the sequence ()()) would look like $1, -1, 1, -1, -1$. As we can see, this is not a proper sequence of parentheses because adding all of the integer representations does not yield 0. The type match_mon : int → {list ← list} indicates that the monitor takes an integer argument and consumes a list in order to produce the resulting list. In a similar vein, we can implement a process that checks that a tree is serialized correctly, which is related to recent work on context-free session types by Thiemann and Vasconcelos [5].

We can also write a monitor that checks whether a given list is sorted in ascending order (shown in Fig. 6). The ascending_mon monitor uses its internal state to enforce a lower bound on subsequent elements of the list. In order to represent this bound, we add an int option type to our language. This value can either be None if no bound has yet been set, or Some b if b is the current bound.

If the list is empty, there is no bound to check, so no contract failure can happen. If the list is nonempty, we check to see if a bound has already been set. If not, we set the bound to be the first received element. If there is already a bound in place, then we check if the received element is greater or equal to the bound. If it is not, then the list must be unsorted, so we abort with a contract failure. We note that the input list n remains unchanged because every element that we examine we pass along unchanged to m.

```
match_mon : int → {list ← list}
a ← match_mon count ← b =
  case b of
  | nil ⇒ assert l1 (count = 0) ; a.nil ; wait b ; close a
  | cons ⇒ x ← recv b ; a.cons ;
    if (x = 1) then send a x ; a ← match_mon (count + 1) ← b;
    else if (x = −1) then assert l2 (count > 0) ; send a x ;
    a ← match_mon (count − 1) ← b ;
    else abort l3 //invalid input
```

Fig. 5. Parenthesis matching monitor.

```
ascending_mon : int option → {list ← list}
m ← ascending_mon bound ← n =
  case n of
  | nil ⇒ m.nil ; wait n ; close m
  | cons ⇒ x ← recv n ;
      case bound of
    | None ⇒ m.cons ; send m x ; m ← ascending_mon (Some x) ← n
    | Some a ⇒ assert l (x ≥ a) ; m.cons ; send m x ;
      m ← ascending_mon (Some x) ← n
```

Fig. 6. Ascending monitor.

```
result_mon : int → {list ← list}
k ← result_mon total ← l =
  case l of
  | nil ⇒ assert l (total = 0) ; k.nil ; wait a ; close k
  | cons ⇒ x ← recv l ; k.cons ; send k x ; k ← result_mon (total − h(x)) ← l

sorter : &{next : ∀n : int.sorter; done : list}
sorter_mon : int → {sorter ← sorter}
a ← sorter_mon total ← b =
  case a of
  | next ⇒ x ← recv a ; b.next ; send b x ; a ← sorter_mon (total + h(x)) ← b
  | done ⇒ b ← result_mon total ← a
```

Fig. 7. Result checking monitor.

To take the above example one step further, assume we have an actual sorting procedure that takes a stream of integers as input and produces a sorted list. We can use the ascending_mon process to verify that the elements of the output list are in sorted order. However, we still need to verify that the elements in the output list are in fact a permutation of the elements that were sent to the sorting procedure. Given a reasonable hash function, h, we can write monitors to accomplish this goal (shown in Fig. 7). We first introduce the sorter type which is an external choice with two options – next which receives an integer and continues behaving as a sorter, or done which completes the sorting and returns a list. We can write a monitor sorter_mon that hashes each element as it is sent to the sorting procedure and keeps track of a running total of the sum of the hashes. Once the sorting procedure has generated the resulting sorted list, we can use the result_mon to compute the hash of each element and subtract it from the total. After all of the elements are received, the monitor checks that the total is 0 – if it is, with high probability, the input stream and output list are permutations of each other. This example is an instance of *result checking* and is inspired by Wasserman and Blum [6].

Mapper Finally, we can also define monitors that check function contracts, such as a contract for a mapping function. Consider the mapper_proc process shown in Fig. 8 which receives an integer, doubles it, sends it back and recurs. Looking at the code, we can see that any positive integer that the mapper has processed will be strictly larger than the original integer. This contract can be imposed on the mapper itself, which is done in the mapper_mon process. This process first examines each integer the mapper receives and asserts it is positive. If this precondition is met, then the value is sent to the mapper and once the mapped value is received, an assertion confirms that it is larger than the original input. The map process applies a mapper m to a list l and produces a list. The wrapper process starts a monitor on the mapper m and passes the monitored mapper m' to the map process.

3. Session types

Session types prescribe the communication behavior of message-passing concurrent processes. We approach them here via their foundation in intuitionistic linear logic [7–9]. The key idea is that an intuitionistic linear sequent

$$A_1, \ldots, A_n \vdash C$$

is interpreted as the interface to a *process expression* P. We label each of the antecedents with a channel name a_i and the succedent with a channel name c. The a_i are the channels *used* and c is the channel *provided* by P.

```
mapper_t = &{done : 1 ; next : ∀n : int.∃n : int.mapper_t}
mapper_proc : {mapper_t}
m ← mapper_proc =
  case m of
  | done ⇒ close m
  | next ⇒ x ← recv m ; send m (2 ∗ x) ; m ← mapper_proc
mapper_mon : {mapper_t ← mapper_t}
n ← mapper_mon ← m =
  case n of
  | done ⇒ m.done ; wait m ; close n
  | next ⇒ x ← recv n ; assert l1 (x > 0)   //precondition
    m.next ; send m x ; y ← recv m ; assert l2 (y > x)   //postcondition
    n.next ; send n y ; n ← mapper_mon ← m

map : {list ← mapper_t ; list}
k ← map ← m l =
  case l of
  | nil ⇒ m.done ; k.nil ; wait l ; wait m ; close k
  | cons ⇒ x ← recv l ; m.next ; send m x ; y ← recv m ; k.cons ; send k y ;
    k ← map m l

wrapper : {list ← mapper_t ; list}
k ← wrapper ← m l =
  m′ ← mapper_mon ← m;   //run monitor
  k ← map m′ l
```

Fig. 8. Higher-order monitor.

$$a_1 : A_1, \ldots, a_n : A_n \vdash P :: (c : C)$$

We abbreviate the antecedents by Δ. All the channels a_i and c must be distinct, and bound variables may be silently renamed to preserve this invariant in the rules. Furthermore, the antecedents are considered modulo exchange. Cut corresponds to parallel composition of two processes that communicate along a private channel x, where P is the *provider* along x and Q the *client*.

$$\frac{\Delta \vdash P :: (x : A) \quad x : A, \Delta' \vdash Q :: (c : C)}{\Delta, \Delta' \vdash x{:}A \leftarrow P \ ; \ Q :: (c : C)} \ \mathsf{cut}$$

Operationally, the process $x : A \leftarrow P \ ; \ Q$ spawns P as a new process and continues as Q, where P and Q communicate along a fresh channel a, which is substituted for x. We sometimes omit the type A of x in the syntax when it is not relevant or can be inferred.

In order to define the operational semantics rigorously, we use *multiset rewriting* [10]. The configuration of executing processes is described as a collection $\mathcal{C}$ of propositions $\mathsf{proc}(c, P)$ (process P is executing, providing along c) and $\mathsf{msg}(c, M)$ (message M is sent along c), where all the channels c are distinct.

To begin with, a cut just spawns a new process, and is in fact the only way new processes are spawned. We describe a transition $\mathcal{C} \longrightarrow \mathcal{C}'$ by defining how a subset of $\mathcal{C}$ can be rewritten to a subset of $\mathcal{C}'$, possibly with a freshness condition that applies to all of $\mathcal{C}$ in order to guarantee the uniqueness of each channel provided.

$$\mathsf{proc}(c, x{:}A \leftarrow P \ ; \ Q) \longrightarrow \mathsf{proc}(a, [a/x]P), \mathsf{proc}(c, [a/x]Q) \quad (a \text{ fresh})$$

Each of the connectives of linear logic then describes a particular kind of communication behavior which we capture in similar rules. Before we move on to that, we consider the identity rule, in logical form and operationally.

$$\frac{}{A \vdash A} \ \mathsf{id} \qquad \frac{}{b : A \vdash a \leftarrow b :: (a : A)} \ \mathsf{id} \qquad \mathsf{proc}(a, a \leftarrow b), \mathcal{C} \longrightarrow [b/a]\mathcal{C}$$

Operationally, it corresponds to identifying the channels a and b, which we implement by substituting b for a in the remainder $\mathcal{C}$ of the configuration (which we make explicit in this rule). The process offering a terminates. We refer to $a \leftarrow b$ as *forwarding* since any messages along a are instead "forwarded" to b.

We consider each class of session type constructors, describing their process expression, typing, and asynchronous operational semantics. The linear logical semantics can be recovered by ignoring the process expressions and channels.

Internal and external choice Even though we distinguish a *provider* and its *client*, this distinction is orthogonal to the direction of communication: both may either send or receive along a common private channel. Session typing guarantees that both sides will always agree on the direction and kind of message that is sent or received, so our situation corresponds to so-called *binary session types* [11].

First, the *internal choice* $c : A \oplus B$ requires the provider to send a token inl or inr along c and continue as prescribed by type A or B, respectively. For practical programming, it is more convenient to support n-ary labeled choice $\oplus\{\ell : A_\ell\}_{\ell \in L}$ where L is a set of labels. A process providing $c : \oplus\{\ell : A_\ell\}_{\ell \in L}$ sends a label $k \in L$ along c and continues with type A_k. The client will operate dually, branching on a label received along c.

$$\frac{k \in L \quad \Delta \vdash P :: (c : A_k)}{\Delta \vdash c.k\ ;\ P :: (c : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus R \qquad \frac{\Delta, c : A_\ell \vdash Q_\ell :: (d : D) \quad \text{for every } \ell \in L}{\Delta, c : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (d : D)}\ \oplus L$$

The operational semantics is somewhat tricky, because we communicate asynchronously. We need to spawn a message carrying the label ℓ, but we also need to make sure that the *next* message sent along the same channel does not overtake the first. Sending a message therefore creates a fresh continuation channel c' for further communication, which we substitute in the continuation of the process. Moreover, the recipient also switches to this continuation channel after the message is received.

$$\mathsf{proc}(c, c.k\ ;\ P) \longrightarrow \mathsf{proc}(c', [c'/c]P), \mathsf{msg}(c, c.k\ ;\ c \leftarrow c') \quad (c' \textit{ fresh})$$
$$\mathsf{msg}(c, c.k\ ;\ c \leftarrow c'), \mathsf{proc}(d, \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L}) \longrightarrow \mathsf{proc}(d, [c'/c]Q_k)$$

It is interesting that the message along c, followed by its continuation c' can be expressed as a well-typed process expression using forwarding $c.k\ ;\ c \leftarrow c'$. This pattern will work for all other pairs of send/receive operations.

External choice reverses the roles of client and provider, both in the typing and the operational rules. Below are the semantics and the typing is in Fig. 10.

$$\mathsf{proc}(d, c.k\ ;\ Q) \longrightarrow \mathsf{msg}(c', c.k\ ;\ c' \leftarrow c), \mathsf{proc}(d, [c'/c]Q) \quad (c' \text{ fresh})$$
$$\mathsf{proc}(c, \mathsf{case}\ c\ (\ell \Rightarrow P_\ell)_{\ell \in L}), \mathsf{msg}(c', c.k\ ;\ c' \leftarrow c) \longrightarrow \mathsf{proc}(c', [c'/c]P_k)$$

Sending and receiving channels Session types are *higher-order* in the sense that we can send and receive channels along channels. Sending a channel is perhaps less intuitive from the logical point of view, so we show that and just summarize the rules for receiving.

If we provide $c : A \otimes B$, we send a channel $a : A$ along c and continue as B. From the typing perspective, it is a restricted form of the usual two-premise $\otimes R$ rule by requiring the first premise to be an identity. This restriction separates spawning of new processes from the sending of channels.

$$\frac{\Delta \vdash P :: (c : B)}{\Delta, a : A \vdash \mathsf{send}\ c\ a\ ;\ P :: (c : A \otimes B)}\ \otimes R^* \qquad \frac{\Delta, x : A, c : B \vdash Q :: (d : D)}{\Delta, c : A \otimes B \vdash x \leftarrow \mathsf{recv}\ c\ ;\ Q :: (d : D)}\ \otimes L$$

The operational rules follow the same patterns as the previous case.

$$\mathsf{proc}(c, \mathsf{send}\ c\ a\ ;\ P) \longrightarrow \mathsf{proc}(c', [c'/c]P), \mathsf{msg}(c, \mathsf{send}\ c\ a\ ;\ c \leftarrow c') \quad (c' \textit{ fresh})$$
$$\mathsf{msg}(c, \mathsf{send}\ c\ a\ ;\ c \leftarrow c'), \mathsf{proc}(d, x \leftarrow \mathsf{recv}\ c\ ;\ Q) \longrightarrow \mathsf{proc}(d, [c'/c][a/x]Q)$$

Receiving a channel (written as a linear implication $A \multimap B$) works symmetrically. Below are the semantics and the typing is shown in Fig. 10.

$$\mathsf{proc}(d, \mathsf{send}\ c\ a\ ;\ Q) \longrightarrow \mathsf{msg}(c', \mathsf{send}\ c\ a\ ;\ c' \leftarrow c), \mathsf{proc}(d, [c'/c]Q) \quad (c' \text{ fresh})$$
$$\mathsf{proc}(c, x \leftarrow \mathsf{recv}\ c\ ;\ P), \mathsf{msg}(c', \mathsf{send}\ c\ a\ ;\ c' \leftarrow c) \longrightarrow \mathsf{proc}(c', [c'/c][a/x]P)$$

Termination We have already seen that a process can terminate by forwarding. Communication along a channel ends explicitly when it has type $\mathbf{1}$ (the unit of $\otimes$) and is closed. By linearity there must be no antecedents in the right rule.

$$\frac{}{\cdot \vdash \mathsf{close}\ c :: (c : \mathbf{1})}\ \mathbf{1}R \qquad \frac{\Delta \vdash Q :: (d : D)}{\Delta, c : \mathbf{1} \vdash \mathsf{wait}\ c\ ;\ Q :: (d : D)}\ \mathbf{1}L$$

Since there cannot be any continuation, the message takes a simple form.

$$\mathsf{proc}(c, \mathsf{close}\ c) \longrightarrow \mathsf{msg}(c, \mathsf{close}\ c)$$
$$\mathsf{msg}(c, \mathsf{close}\ c), \mathsf{proc}(d, \mathsf{wait}\ c\ ;\ Q) \longrightarrow \mathsf{proc}(d, Q)$$

Quantification First-order quantification over elements of domains such as integers, strings, or booleans allows ordinary basic data values to be sent and received. At the moment, since we have no type families indexed by values, the quantified variables cannot actually appear in their scope. This will change in Section 5 so we anticipate this in these rules.

In order to track variables ranging over values, a new context Ψ is added to all judgments and the preceding rules are modified accordingly. All value variables n declared in Ψ must be distinct. Such variables are not linear, but can be

arbitrarily reused, and are therefore propagated to all premises in all rules. We write $\Psi \vdash v : \tau$ to check that value v has type τ in context Ψ.

$$\frac{\Psi \vdash v : \tau \quad \Psi\,;\,\Delta \vdash P :: (c : [v/n]A)}{\Psi\,;\,\Delta \vdash \mathsf{send}\,c\,v\,;\,P :: (c : \exists n{:}\tau\,.\,A)}\;\exists R \qquad \frac{\Psi, n{:}\tau\,;\,\Delta, c : A \vdash Q :: (d : D)}{\Psi\,;\,\Delta, c : \exists n{:}\tau\,.\,A \vdash n \leftarrow \mathsf{recv}\,c\,;\,Q :: (d : D)}\;\exists L$$

$$\mathsf{proc}(c, \mathsf{send}\,c\,v\,;\,P) \longrightarrow \mathsf{proc}(c', [c'/c]P), \mathsf{msg}(c, \mathsf{send}\,c\,v\,;\,c \leftarrow c') \quad (c' \text{ fresh})$$
$$\mathsf{msg}(c, \mathsf{send}\,c\,v\,;\,c \leftarrow c'), \mathsf{proc}(d, n \leftarrow \mathsf{recv}\,c\,;\,Q) \longrightarrow \mathsf{proc}(d, [c'/c][v/n]Q)$$

The situation for universal quantification is symmetric. The semantics are given below and the typing is shown in Fig. 10.

$$\mathsf{proc}(d, \mathsf{send}\,c\,v\,;\,Q) \longrightarrow \mathsf{msg}(c', \mathsf{send}\,c\,v\,;\,c' \leftarrow c), \mathsf{proc}(d, [c'/c]Q) \quad (c' \text{ fresh})$$
$$\mathsf{proc}(c, n \leftarrow \mathsf{recv}\,c\,;\,P), \mathsf{msg}(c', \mathsf{send}\,c\,v\,;\,c' \leftarrow c) \longrightarrow \mathsf{proc}(c', [c'/c][v/n]P)$$

Shifts Finally, we come to shifts. To make shifts explicit, we *polarize* the syntax and use so-called *shifts* to change the direction of communication. For more detail, see Pfenning [4].

$$\begin{array}{lll} \text{Negative types} & A^-, B^- & ::= \;\&\{\ell : A^-_\ell\}_{\ell\in L} \mid A^+ \multimap B^- \mid \forall n{:}\tau\,.\,A^- \mid \uparrow A^+ \\ \text{Positive types} & A^+, B^+ & ::= \;\oplus\{\ell : A^+_\ell\}_{\ell\in L} \mid A^+ \otimes B^+ \mid 1 \mid \exists n{:}\tau\,.\,A^+ \mid \downarrow A^- \\ \text{Types} & A, B, C, D & ::= \;A^- \mid A^+ \end{array}$$

From the perspective of the provider, all negative types receive and all positive types send. It is then clear that $\uparrow A$ must receive a shift message and then start sending, while $\downarrow A$ must send a shift message and then start receiving. For this restricted form of shift, the logical rules are otherwise uninformative. The semantics are given below and the typing is shown in Fig. 10.

$$\mathsf{proc}(c, \mathsf{send}\,c\,\mathsf{shift}\,;\,P) \longrightarrow \mathsf{proc}(c', [c'/c]P), \mathsf{msg}(c, \mathsf{send}\,c\,\mathsf{shift}\,;\,c \leftarrow c') \quad (c' \text{ fresh})$$
$$\mathsf{msg}(c, \mathsf{send}\,c\,\mathsf{shift}\,;\,c \leftarrow c'), \mathsf{proc}(d, \mathsf{shift} \leftarrow \mathsf{recv}\,d\,;\,Q) \longrightarrow \mathsf{proc}(d, [c'/c]Q)$$

$$\mathsf{proc}(d, \mathsf{send}\,c\,\mathsf{shift}\,;\,Q) \longrightarrow \mathsf{msg}(c', \mathsf{send}\,c\,\mathsf{shift}\,;\,c' \leftarrow c), \mathsf{proc}(d, [c'/c]Q) \quad (c' \text{ fresh})$$
$$\mathsf{proc}(c, \mathsf{shift} \leftarrow \mathsf{recv}\,c\,;\,P), \mathsf{msg}(c', \mathsf{send}\,c\,\mathsf{shift}\,;\,c' \leftarrow c) \longrightarrow \mathsf{proc}(c', [c'/c]P)$$

Recursive types Practical programming with session types requires them to be recursive, and processes using them also must allow recursion. To support rich subtyping and refinement types, it is convenient for recursive types to be *equirecursive*. This means a defined type such as list^+ is viewed as *equal* to its definition $\oplus\{\ldots\}$ rather than *isomorphic*. For this view to be consistent, we require type definitions to be *contractive* [12], that is, they need to provide at least one send or receive interaction before recursing.

The most popular formalization of equirecursive types is to introduce an explicit μ-constructor. For example,

$$\mathsf{list} = \mu\alpha\,.\,\oplus\{\,\mathsf{cons} : \exists n{:}\mathsf{int}.\,\alpha, \mathsf{nil} : \mathbf{1}\,\}$$

with rules unrolling the type $\mu\alpha\,.\,A$ to $[(\mu\alpha\,.\,A)/\alpha]A$. An alternative (see, for example, Balzers and Pfenning [13]) is to use an explicit definition just as we stated, for example, list and consider the left-hand side *equal* to the right-hand side in our discourse.

Recursive processes In addition to recursively defined types, we also need recursively defined processes. We follow the general approach of Toninho et al. [14] for the integration of a (functional) data layer into session-typed communication. A process can be named p, ascribed a type, and be defined as follows.

$$p : \forall n_1{:}\tau_1.\ \ldots\ \forall n_k{:}\tau_k.\{A \leftarrow A_1, \ldots, A_m\}$$
$$x \leftarrow p\,n_1\ \ldots\ n_k \leftarrow y_1, \ldots, y_m = P$$

where we check $(n_1{:}\tau_1, \ldots, n_k{:}\tau_k)\,;\,(y_1{:}A_1, \ldots, y_m{:}A_m) \vdash P :: (x : A)$

We use such process definitions when spawning a new process with the syntax

$$c \leftarrow p\,e_1\ \ldots, e_k \leftarrow d_1, \ldots, d_m\,;\,Q$$

which we check with the below rule where $\Theta = [e_1/n_1, \ldots, e_k/n_k]$ and $\Delta' = [\Theta](d_1{:}A_1, \ldots, d_m{:}A_m)$

$$\frac{(\Psi \vdash e_i : \tau_i)_{i\in\{1,\ldots,k\}} \quad \Psi\,;\,\Delta, c : [\Theta]A \vdash Q :: (d : D) \quad c \text{ fresh}}{\Psi\,;\,\Delta, \Delta' \vdash c \leftarrow p\,e_1 \ldots e_k \leftarrow d_1, \ldots, d_m\,;\,Q :: (d : D)}\;\mathsf{pdef}$$

After evaluating the value arguments, the call consumes the channels d_j (which will not be available to the continuation Q, due to linearity). The continuation Q will then be the (sole) client of c and the new process providing c will execute $[c/x][d_1/y_1]\ldots[d_m/y_m]P$.

One more quick shorthand used in the examples: a tail-call $c \leftarrow p\,\overline{e} \leftarrow \overline{d}$ in the definition of a process that provides along c is expanded into $c' \leftarrow p\,\overline{e} \leftarrow \overline{d}\,;\,c \leftarrow c'$ for a fresh c'.

$P, Q, R ::=$	
close c	send end and terminate
| wait c ; Q	recv end, continue with Q
| send c a ; Q	send channel a along c, and continue as Q
| $x \leftarrow$ recv c ; Q_x	receive a along c, continue as Q_a
| $c.\ell_j$; Q	send ℓ_j along c, continue as Q
| case c of $\{\ell_i \Rightarrow Q_i\}_i$	recv ℓ_j along c, cont. as Q_j
| send c v ; Q	send value v along c, continue as Q
| $n \leftarrow$ recv c ; Q_n	recv value v along c, continue as Q_v
| send c shift ; Q	send shift along c, continue as Q
| shift $\leftarrow$ recv c ; Q	receive shift along c, continue as Q
| $x \leftarrow P_x$; Q_x	create new a, spawn P_a, continue as Q_a
| $c \leftarrow d$	connect c with d and terminate
| assert l p ; Q	assert predicate p with label l and continue as Q
| abort l	abort with label l

Fig. 9. Process expressions.

Stopping computation Finally, in order to successfully monitor computation, we need the capability to assert conditions and stop the computation at particular points. We add assert blocks to check conditions on observable values and an abort block to stop computation. We tag the assert and abort blocks with a label l which allows us to determine which assertion failed when the computation aborts. The semantics are given below and the typing is in Fig. 10.

$$\mathsf{proc}(c, \mathsf{assert}\ l\ \mathsf{True};\ Q) \longrightarrow \mathsf{proc}(c, Q) \qquad \mathsf{proc}(c, \mathsf{assert}\ l\ \mathsf{False};\ Q) \longrightarrow \mathsf{abort}\ l$$

We overload the abort l notation to refer to both the semantic construct (as shown above) and the process expression (used frequently in our examples in Section 3). Progress and preservation were proven for the above system, with the exception of the abort and assert rules, in prior work [4]. The additional proof cases do not change the proof significantly. We summarize the process expressions in Fig. 9 and the process typing in Fig. 10.

Configuration typing Finally, we present the configuration typing. We assume that the comma operator is associative with $\cdot$ as the unit.

$$\mathcal{C} = \cdot \mid \mathsf{proc}(c, P) \mid \mathsf{msg}(c, P) \mid \mathcal{C}_1, \mathcal{C}_2 \mid \mathsf{abort}\ l$$

$$\frac{}{\cdot \Vdash \cdot} \qquad \frac{\Delta_1 \Vdash \mathcal{C}_1 : \Delta_2 \quad \Delta_1 \Vdash \mathcal{C}_2 : \Delta_2}{\Delta_1 \Vdash \mathcal{C}_1, \mathcal{C}_2 : \Delta_2} \qquad \frac{\Delta \vdash P :: (c : A)}{\Delta \Vdash \mathsf{proc}(c, P) : A} \qquad \frac{\Delta \vdash P :: (c : A)}{\Delta \Vdash \mathsf{msg}(c, P) : A}$$

4. Monitors as partial identity processes

In the literature on contracts, they are often depicted as guards on values sent to and returned from functions. In our case, contracts really *are* processes that monitor message-passing communications between processes. For us, a central property of contracts is that a program may be executed with or without contract checking and, unless an alarm is raised, the observable outcome should be the same. This means that contract monitors should be *partial identity processes* passing messages back and forth along channels while testing properties of the messages. We note that monitors may introduce divergence because non-termination is not observed.

This may seem very limiting at first, but session-typed processes can maintain local state. For example, consider the functional notion of a *dependent contract*, where the contract on the result of a function depends on its input. Here, a function would be implemented by a process that receives arguments and sends back the return value *along the same channel*, such as the mapper_mon process shown in Section 2. Therefore, a monitor can remember any (non-linear) "argument values" and use them to validate the "result value". Similarly, when a list is sent element by element, properties that can be easily checked include constraints on its length, or whether it is in ascending order. Moreover, local state can include additional (private) concurrent processes.

This raises a second question: how can we guarantee that a monitor really is a partial identity? The criterion should be general enough to allow us to naturally express the contracts from a wide range of examples. A key feature of our system is that contracts are expressed as session-typed processes.

The purpose of this section is to present and prove the correctness of a criterion on session-typed processes that guarantees that they are observationally equivalent to partial identity processes. This criterion is a type-based judgment which will be developed iteratively throughout this section. All the contracts in this paper can be verified to be partial identities using our definition.

4.1. Buffering values

We start by recalling our first monitoring example, the factor_monitor, shown in Fig. 1. To check that factor_monitor is a partial identity we need to track that p and q are received from the provider, in this order. In general, for any received

$$\frac{}{\Psi\,;\, b : A \vdash a \leftarrow b :: (a : A)}\ \mathsf{id} \qquad \frac{\Psi\,;\,\Delta \vdash P :: (x : A) \quad x : A, \Delta' \vdash Q :: (c : C)}{\Psi\,;\,\Delta, \Delta' \vdash x{:}A \leftarrow P\,;\, Q :: (c : C)}\ \mathsf{cut}$$

$$\frac{}{\cdot \vdash \mathsf{close}\ c :: (c : \mathbf{1})}\ \mathbf{1}R \qquad \frac{\Psi;\,\Delta \vdash Q :: (d : D)}{\Psi;\,\Delta, c : \mathbf{1} \vdash \mathsf{wait}\ c\,;\, Q :: (d : D)}\ \mathbf{1}L$$

$$\frac{\Psi \vdash b : \mathsf{bool} \quad \Psi\,;\,\Delta \vdash Q :: (x : A)}{\Psi\,;\,\Delta \vdash \mathsf{assert}\ \rho\ b;\, Q :: (x : A)}\ \mathsf{assert} \qquad \frac{}{\Psi\,;\,\Delta \vdash \mathsf{abort}\ \rho :: (x : A)}\ \mathsf{abort}$$

$$\frac{\Psi\,;\,\Delta \vdash P :: (c : A^+)}{\Psi\,;\,\Delta \vdash \mathsf{shift} \leftarrow \mathsf{recv}\ c\,;\, P :: (c : \uparrow A^+)}\ \uparrow R \qquad \frac{\Psi\,;\,\Delta, c : A^+ \vdash Q :: (d : D)}{\Psi\,;\,\Delta, c : \uparrow A^+ \vdash \mathsf{send}\ c\ \mathsf{shift}\,;\, Q :: (d : D)}\ \uparrow L$$

$$\frac{\Psi\,;\,\Delta \vdash P :: (c : A^-)}{\Psi\,;\,\Delta \vdash \mathsf{send}\ c\ \mathsf{shift}\,;\, P :: (c : \downarrow A^-)}\ \downarrow R \qquad \frac{\Psi\,;\,\Delta, c : A^- \vdash Q :: (d : D)}{\Psi\,;\,\Delta, c : \downarrow A^- \vdash \mathsf{shift} \leftarrow \mathsf{recv}\ c\,;\, Q :: (d : D)}\ \downarrow L$$

$$\frac{\Psi \vdash v : \tau \quad \Psi\,;\,\Delta \vdash P :: (c : [v/n]A)}{\Psi\,;\,\Delta \vdash \mathsf{send}\ c\ v\,;\, P :: (c : \exists n{:}\tau\,.\,A)}\ \exists R \qquad \frac{\Psi, n{:}\tau\,;\,\Delta, c : A \vdash Q :: (d : D)}{\Psi\,;\,\Delta, c : \exists n{:}\tau\,.\,A \vdash n \leftarrow \mathsf{recv}\ c\,;\, Q :: (d : D)}\ \exists L$$

$$\frac{\Psi, n{:}\tau\,;\,\Delta \vdash P :: (c : A)}{\Psi\,;\,\Delta \vdash n \leftarrow \mathsf{recv}\ c\,;\, P :: (c : \forall n{:}\tau\,.\,A)}\ \forall R \qquad \frac{\Psi \vdash v : \tau \quad \Psi\,;\,\Delta, c : [v/n]A \vdash Q :: (d : D)}{\Psi\,;\,\Delta, c : \forall n{:}\tau\,.\,A \vdash \mathsf{send}\ c\ v\,;\, Q :: (d : D)}\ \forall L$$

$$\frac{\Psi\,;\,\Delta \vdash P :: (c : B)}{\Psi\,;\,\Delta, a : A \vdash \mathsf{send}\ c\ \ a\,;\, P :: (c : A \otimes B)}\ \otimes R$$

$$\frac{\Psi;\,\Delta, x : A, c : B \vdash Q :: (d : D)}{\Psi;\,\Delta, c : A \otimes B \vdash x \leftarrow \mathsf{recv}\ c\,;\, Q :: (d : D)}\ \otimes L$$

$$\frac{\Psi;\,\Delta, x : A \vdash P :: (c : B)}{\Psi;\,\Delta \vdash x \leftarrow \mathsf{recv}\ c\,;\, P :: (c : A \multimap B)}\ \multimap R$$

$$\frac{\Psi\,;\,\Delta, c : B \vdash Q :: (d : D)}{\Psi\,;\,\Delta, a : A, c : A \multimap B \vdash \mathsf{send}\ c\ a\,;\, Q :: (d : D)}\ \multimap L$$

$$\frac{\Psi;\,\Delta \vdash P_\ell :: (c : A_\ell) \quad \text{for every } \ell \in L}{\Psi;\,\Delta \vdash \mathsf{case}\ c\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (c : \&\{\ell : A_\ell\}_{\ell \in L})}\ \&R$$

$$\frac{k \in L \quad \Psi;\,\Delta, c : A_k \vdash Q :: (d : D)}{\Psi;\,\Delta, c : \&\{\ell : A_\ell\}_{\ell \in L} \vdash c.k\,;\, Q :: (d : D)}\ \&L$$

$$\frac{k \in L \quad \Psi;\,\Delta \vdash P :: (c : A_k)}{\Psi;\,\Delta \vdash c.k\,;\, P :: (c : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus R$$

$$\frac{\Psi;\,\Delta, c : A_\ell \vdash Q_\ell :: (d : D) \quad \text{for every } \ell \in L}{\Psi;\,\Delta, c : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (d : D)}\ \oplus L$$

Fig. 10. Typing process expressions.

message, we need to enter it into a message queue $[q]$ and we need to check that the messages are passed on in the correct order. As a first cut (to be generalized several times), we write for negative types:

$$[q](b : B^-)\ ;\ \Psi \vdash P :: (a : A^-)$$

which expresses that the two endpoints of the monitor are $a : A^-$ and $b : B^-$ (both negative), and we have already received the messages in q along a.

A monitor, at the top level, is defined with

$$mon : \tau_1 \to \ldots \to \tau_n \to \{A \leftarrow A\}$$
$$a \leftarrow mon\ x_1 \ldots x_n \leftarrow b = P$$

where the context Ψ declares value variables x. The body P here is type-checked as one of (depending on the polarity of A)

$$[\,](b : A^-)\ ;\ \Psi \vdash P :: (a : A^-) \quad \text{or} \quad (b : A^+)\ ;\ \Psi \vdash P :: [\,](a : A^+)$$

where $\Psi = (x_1{:}\tau_1) \cdots (x_n{:}\tau_n)$. A use such as

$$c \leftarrow mon\ e_1 \ldots e_n \leftarrow c$$

is transformed into

$$c' \leftarrow mon\, e_1 \ldots e_n \leftarrow c\ ;\ c \leftarrow c'$$

for a fresh c' and type-checked accordingly.

In general, queues have the form $q = m_1 \cdots m_n$ with

$$\begin{array}{lllll} m & ::= & l_k & \text{labels} & \\ & | & c & \text{channels} & | \ n \quad \text{value variables} \\ & | & \mathsf{end} & \text{close} & | \ \mathsf{shift} \quad \text{shifts} \end{array}$$

where m_1 is the front of the queue and m_n the back.

When P receives a message, we add it to the end of the queue q. We also need to add it to Ψ to remember its type. In our example $\tau = \mathsf{int}$.

$$\frac{[q \cdot n](b : B)\ ;\ \Psi, n{:}\tau \vdash P :: (a : A^-)}{[q](b : B)\ ;\ \Psi \vdash n \leftarrow \mathsf{recv}\, a\ ;\ P :: (a : \forall n{:}\tau\,.\, A^-)}\ \forall R$$

Conversely, when we *send* along b the message must be equal to the one at the front of the queue (and therefore it must be a variable). The m is a value variable and remains in the context so it can be reused for later assertion checks. However, it could never be sent again since it has been removed from the queue.

$$\frac{[q](b : [m/n]B)\ ;\ \Psi, m{:}\tau \vdash Q :: (a : A^-)}{[m \cdot q](b : \forall n{:}\tau\,.\, B)\ ;\ \Psi, m{:}\tau \vdash \mathsf{send}\, b\, m\ ;\ Q :: (a : A^-)}\ \forall L$$

All the other send and receive rules for negative types ($\forall$, $\multimap$, $\&$) follow exactly the same pattern. For positive types, a queue must be associated with the channel along which the monitor provides (the succedent of the sequent judgment).

$$(b : B^+)\ ;\ \Psi \vdash Q :: [q](a : A^+)$$

Moreover, when end has been received along b the corresponding process has terminated and the channel is closed, so we generalize the judgment to

$$\omega\ ;\ \Psi \vdash Q :: [q](a : A^+) \qquad \text{with } \omega = \cdot \mid (b : B).$$

The shift messages change the direction of communication. They therefore need to switch between the two judgments and also ensure that the queue has been emptied before we switch direction. Here are the two rules for $\uparrow$, which appears in our factoring example:

$$\frac{[q \cdot \mathsf{shift}](b : B^-)\ ;\ \Psi \vdash P :: (a : A^+)}{[q](b : B^-)\ ;\ \Psi \vdash \mathsf{shift} \leftarrow \mathsf{recv}\, a\ ;\ P :: (a : \uparrow A^+)}\ \uparrow R$$

We notice that after receiving a shift, the channel a already changes polarity (we now have to send along it), so we generalize the judgment, allowing the succedent to be either positive or negative. And conversely for the other judgment.

$$\begin{array}{l} [q](b : B^-)\ ;\ \Psi \vdash P :: (a : A) \\ \omega\ ;\ \Psi \vdash Q :: [q](a : A^+) \quad \text{where } \omega = \cdot \mid (b : B) \end{array}$$

When we *send* the final shift, we initialize a new empty queue. Because the queue is empty the two sides of the monitor must have the same type.

$$\frac{(b : B^+)\ ;\ \Psi \vdash Q :: [\,](a : B^+)}{[\mathsf{shift}](b : \uparrow B^+)\ ;\ \Psi \vdash \mathsf{send}\, b\, \mathsf{shift}\ ;\ Q :: (a : B^+)}\ \uparrow L$$

The rules for forwarding are also straightforward. Both sides need to have the same type, and the queue must be empty. As a consequence, the immediate forward is always a valid monitor at a given type.

$$\frac{}{(b : A^+)\ ;\ \Psi \vdash a \leftarrow b :: [\,](a : A^+)}\ \mathsf{id}^+ \qquad \frac{}{[\,](b : A^-)\ ;\ \Psi \vdash a \leftarrow b :: (a : A^-)}\ \mathsf{id}^-$$

4.2. The current judgment and a first generalization

The current rules allow us to communicate *only along the channels a and b that are being monitored*. If we send channels along channels, however, these channels must be recorded in the typing judgment, but we are not allowed to communicate along them directly. On the other hand, if we spawn internal (local) channels, say, as auxiliary data structures, we should be able to interact with them since such interactions are not externally observable. Our judgment thus requires two additional contexts: Δ for channels internal to the monitor, and Γ for externally visible channels that may be sent along the monitored channels. Our full judgments therefore are

$$[q](b : B^-)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash P :: (a : A)$$
$$\omega\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash Q :: [q](a : A^+) \quad \text{where } \omega = \cdot \mid (b : B)$$

The rules developed so far in this section are summarized in Fig. 11.

4.3. Spawning new processes

The most complex part of checking that a process is a valid monitor involves spawning new processes. In order to be able to spawn and use local (private) processes, we have introduced the (so far unused) context Δ that tracks such channels. We use it here only in the following two rules:

$$\frac{\Psi\ ;\ \Delta \vdash P :: (c : C) \quad \omega\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta', c{:}C \vdash Q :: [q](a : A^+)}{\omega\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta, \Delta' \vdash (c : C) \leftarrow P\ ;\ Q :: [q](a : A^+)}\ \mathsf{c}_1^+$$

$$\frac{\Psi\ ;\ \Delta \vdash P :: (c : C) \quad [q](b : B^-)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta', c{:}C \vdash Q :: (a : A)}{[q](b : B^-)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta, \Delta' \vdash (c : C) \leftarrow P\ ;\ Q :: (a : A)}\ \mathsf{c}_1^-$$

The second premise (that is, the continuation of the monitor) remains the monitor, while the first premise corresponds to a freshly spawned local process accessible through channel c. All the ordinary left rules for sending or receiving along channels in Δ are also available for the two monitor validity judgments. By the strong ownership discipline of intuitionistic session types, none of this information can flow out of the monitor.

It is also possible for a single monitor to decompose into two monitors that operate concurrently, in sequence. In that case, the queue q may be split anywhere, as long as the intermediate type has the right polarity. Note that Γ must be chosen to contain all channels in q_2, while Γ' must contain all channels in q_1.

$$\frac{\omega\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash P :: [q_2](c : C^+) \quad (c : C^+)\ ;\ \Psi\ ;\ \Gamma'\ ;\ \Delta' \vdash Q :: [q_1](a : A^+)}{\omega\ ;\ \Psi\ ;\ \Gamma, \Gamma'\ ;\ \Delta, \Delta' \vdash c : C^+ \leftarrow P\ ;\ Q :: [q_1 \cdot q_2](a : A^+)}\ \mathsf{c}_2^+$$

Why is this correct? The first messages sent along a will be the messages in q_1. If we receive messages along c in the meantime, they will be first the messages in q_2 (since P is a monitor), followed by any messages that P may have received along b if $\omega = (b : B)$. The second rule is entirely symmetric, with the flow of messages in the opposite direction.

$$\frac{[q_1](b : B^-)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash P :: (c : C^-) \quad [q_2](c : C^-)\ ;\ \Psi'\ ;\ \Gamma'\ ;\ \Delta' \vdash Q :: (a : A)}{[q_1 \cdot q_2](b : B^-)\ ;\ \Psi\ ;\ \Gamma, \Gamma'\ ;\ \Delta, \Delta' \vdash c : C^- \leftarrow P\ ;\ Q :: (a : A)}\ \mathsf{c}_2^-$$

The next two rules allow a monitor to be attached to a channel x that is passed between a and b. The monitored version of x is called x', where x' is chosen fresh. This apparently violates our property that we pass on all messages exactly as received, because here we pass on a monitored version of the original. However, if monitors are partial identities, then the original x and the new x' are indistinguishable (unless a necessary alarm is raised), which will be a tricky part of the correctness proof.

$$\frac{(x : C^+)\ ;\ \Psi\ ;\ \cdot\ ;\ \Delta \vdash P :: [\,](x' : C^+) \quad \omega\ ;\ \Psi\ ;\ \Gamma, x'{:}C^+\ ;\ \Delta' \vdash Q :: [q_1 \cdot x' \cdot q_2](a : A^+)}{\omega\ ;\ \Psi\ ;\ \Gamma, x{:}C^+\ ;\ \Delta, \Delta' \vdash x' \leftarrow P\ ;\ Q :: [q_1 \cdot x \cdot q_2](a : A^+)}\ \mathsf{c}_3^+$$

$$\frac{[\,](x : C^-)\ ;\ \Psi\ ;\ \cdot\ ;\ \Delta \vdash P :: (x' : C^-)\ [q_1 \cdot x' \cdot q_2](b : B^-)\ ;\ \Psi\ ;\ \Gamma, x'{:}C^-\ ;\ \Delta' \vdash Q :: (a : A)}{[q_1 \cdot x \cdot q_2](b : B^-)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta, \Delta' \vdash x' \leftarrow P\ ;\ Q :: (a : A)}\ \mathsf{c}_3^-$$

There are two more versions of these rules, depending on whether the types of x and the monitored types are positive or negative. These rules play a critical role in monitoring higher-order processes, because monitoring $c : A^+ \multimap B^-$ may require us to monitor the continuation $c : B^-$ (already covered) but also communication along the channel $x : A^+$ received along c.

$$\frac{(\forall \ell \in L) \quad (b : B_\ell)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash Q_\ell :: [q \cdot \ell](a : A^+)}{(b : \oplus\{\ell : B_\ell\}_{\ell \in L})\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash \mathsf{case}\ b\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: [q](a : A^+)}\ \oplus L$$

$$\frac{\omega\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash P :: [q](a : B_k) \quad (k \in L)}{\omega\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash a.k\ ;\ P :: [k \cdot q](a : \oplus\{\ell : B_\ell\}_{\ell \in L})}\ \oplus R$$

$$\frac{(\forall \ell \in L) \quad [q \cdot \ell](b : B)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash P_\ell :: (a : A_\ell)}{[q](b : B)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash \mathsf{case}\ a\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (a : \&\{\ell : A_\ell\}_{\ell \in L})}\ \&R$$

$$\frac{[q](b : B_k)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash P :: (a : A) \quad (k \in L)}{[k \cdot q](b : \oplus\{\ell : B_\ell\}_{\ell \in L})\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash b.k\ ;\ P :: (a : A)}\ \&L$$

$$\frac{(b : B)\ ;\ \Psi\ ;\ \Gamma, x{:}C\ ;\ \Delta \vdash Q :: [q \cdot x](a : A)}{(b : C \otimes B)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash x \leftarrow \mathsf{recv}\ b\ ;\ Q :: [q](a : A)}\ \otimes L$$

$$\frac{\omega\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash P :: [q](a : A)}{\omega\ ;\ \Psi\ ;\ \Gamma, x{:}C\ ;\ \Delta \vdash \mathsf{send}\ a\ x\ ;\ P :: [x \cdot q](a : C \otimes A)}\ \otimes R^*$$

$$\frac{[q \cdot x](b : B)\ ;\ \Psi\ ;\ \Gamma, x{:}C\ ;\ \Delta \vdash P :: (a : A)}{[q](b : B)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash x \leftarrow \mathsf{recv}\ a\ ;\ P :: (a : C \multimap A)}\ \multimap R$$

$$\frac{[q](b : B)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash Q :: (a : A)}{[x \cdot q](b : C \multimap B)\ ;\ \Psi\ ;\ \Gamma, x{:}C\ ;\ \Delta \vdash \mathsf{send}\ b\ x\ ;\ Q :: (a : A)}\ \multimap L^*$$

$$\frac{\cdot\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash Q :: [q \cdot \mathsf{end}](a : A)}{(b : \mathbf{1})\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash \mathsf{wait}\ b\ ;\ Q :: [q](a : A)}\ \mathbf{1}L$$

$$\frac{}{\cdot\ ;\ \Psi\ ;\ \cdot\ ;\ \cdot \vdash \mathsf{close}\ a :: [\mathsf{end}](a : \mathbf{1})}\ \mathbf{1}R$$

$$\frac{(b : B)\ ;\ \Psi, n{:}\tau\ ;\ \Gamma\ ;\ \Delta \vdash Q :: [q \cdot n](a : A)}{(b : \exists n{:}\tau\,.\,B)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash n \leftarrow \mathsf{recv}\ b\ ;\ Q :: [q](a : A)}\ \exists L$$

$$\frac{\omega\ ;\ \Psi, m{:}\tau\ ;\ \Gamma\ ;\ \Delta \vdash P :: [q](a : [m/n]A)}{\omega\ ;\ \Psi, m{:}\tau\ ;\ \Gamma\ ;\ \Delta \vdash \mathsf{send}\ a\ m\ ;\ P :: [m \cdot q](a : \exists n{:}\tau\,.\,A)}\ \exists R$$

$$\frac{[q \cdot n](b : B)\ ;\ \Psi, n{:}\tau\ ;\ \Gamma\ ;\ \Delta \vdash P :: (a : A^-)}{[q](b : B)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash n \leftarrow \mathsf{recv}\ a\ ;\ P :: (a : \forall n{:}\tau\,.\,A^-)}\ \forall R$$

$$\frac{[q](b : [m/n]B)\ ;\ \Psi, m{:}\tau\ ;\ \Gamma\ ;\ \Delta \vdash P :: (a : A)}{[m \cdot q](b : \forall n{:}\tau\,.\,B)\ ;\ \Psi, m{:}\tau\ ;\ \Gamma\ ;\ \Delta \vdash \mathsf{send}\ b\ m\ ;\ Q :: (a : A)}\ \forall L$$

$$\frac{(b : B^-)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash Q :: [q \cdot \mathsf{shift}](a : A^+)}{(b : \downarrow B^-)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash \mathsf{shift} \leftarrow \mathsf{recv}\ b\ ;\ Q :: [q](a : A^+)}\ \downarrow L$$

$$\frac{[\,](b : A^-)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash P :: (a : A^-)}{(b : A^-)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash \mathsf{send}\ a\ \mathsf{shift}\ ;\ P :: [\mathsf{shift}](a : \downarrow A^-)}\ \downarrow R$$

$$\frac{[q \cdot \mathsf{shift}](b : B^-)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash P :: (a : A^+)}{[q](b : B^-)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash \mathsf{shift} \leftarrow \mathsf{recv}\ a\ ;\ P :: (a : \uparrow A^+)}\ \uparrow R$$

$$\frac{(b : B^+)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash Q :: [\,](a : A^+)}{[\mathsf{shift}](b : \uparrow B^+)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Delta \vdash \mathsf{send}\ b\ \mathsf{shift}\ ;\ Q :: (a : A^+)}\ \uparrow L$$

Fig. 11. Current rules.

In actual programs, we mostly use cut $x \leftarrow P\ ;\ Q$ in the form $x \leftarrow p\ \overline{e} \leftarrow \overline{d}\ ;\ Q$ where p is a defined process. The rules are completely analogous, except that for those rules that require splitting a context in the conclusion, the arguments $\overline{d}$ will provide the split for us. When a new sub-monitor is invoked in this way, we remember and eventually check that the process p must also be a partial identity process, unless we are already checking it. This has the effect that recursively defined monitors with proper recursive calls are in fact allowed. Every instance of a recursive monitor will start with an empty queue. This is important, because monitors for recursive types usually have a recursive structure. An illustration of this can be seen in pos_mon in Fig. 2. In order to verify this monitor, we would apply the $\oplus$R, $\exists$R, $\oplus$L, $\exists L$, and c_3^+ rules.

Message to client of c		Message to provider of c	
$\langle\!\langle \mathsf{msg}^+(c, c.k\,;\, c \leftarrow c')\rangle\!\rangle$	$= k$	$\langle\!\langle \mathsf{msg}^-(c', c.k\,;\, c' \leftarrow c)\rangle\!\rangle$	$= k$
$\langle\!\langle \mathsf{msg}^+(c, \mathsf{send}\ c\ d\,;\, c \leftarrow c')\rangle\!\rangle$	$= d$	$\langle\!\langle \mathsf{msg}^-(c', \mathsf{send}\ c\ d\,;\, c' \leftarrow c)\rangle\!\rangle$	$= d$
$\langle\!\langle \mathsf{msg}^+(c, \mathsf{close}\ c)\rangle\!\rangle$	$= \mathsf{end}$		
$\langle\!\langle \mathsf{msg}^+(c, \mathsf{send}\ c\ v\,;\, c \leftarrow c')\rangle\!\rangle$	$= v$	$\langle\!\langle \mathsf{msg}^-(c', \mathsf{send}\ c\ v\,;\, c' \leftarrow c)\rangle\!\rangle$	$= v$
$\langle\!\langle \mathsf{msg}^+(c, \mathsf{send}\ c\ \mathsf{shift}\,;\, c \leftarrow c')\rangle\!\rangle$	$= \mathsf{shift}$	$\langle\!\langle \mathsf{msg}^-(c', \mathsf{send}\ c\ \mathsf{shift}\,;\, c' \leftarrow c)\rangle\!\rangle$	$= \mathsf{shift}$

Fig. 12. Message queues.

4.4. Transparency

We need to show that monitors are *transparent*, that is, they are indeed observationally equivalent to partial identity processes. Because of the richness of types and process expressions and the generality of the monitors allowed, the proof has some complexities.

Configurations and messages First, we define the configuration typing, which consists of just three rules. Because we also send and receive ordinary values, we also need to type (closed) substitutions $\sigma = (v_1/n_1, \ldots, v_k/n_k)$ using the judgment $\sigma :: \Psi$.

$$\frac{}{(\cdot) :: (\cdot)} \qquad \frac{\cdot \vdash v : \tau}{(v/n) :: (n : \tau)} \qquad \frac{\sigma_1 :: \Psi_1 \quad \sigma_2 :: \Psi_2}{(\sigma_1, \sigma_2) :: (\Psi_1, \Psi_2)}$$

For configurations, we use the judgment

$$\Delta \vdash \mathcal{C} :: \Delta'$$

which expresses that process configuration $\mathcal{C}$ *uses* the channels in Δ and *provides* the channels in Δ'. Channels that are neither used nor offered by $\mathcal{C}$ are "passed through". Messages are just a restricted form of processes, so they are typed exactly the same way. We write *pred* for either proc or msg.

$$\frac{}{\Delta \vdash (\cdot) :: \Delta} \qquad \frac{\Delta_0 \vdash \mathcal{C}_1 :: \Delta_1 \quad \Delta_1 \vdash \mathcal{C}_2 :: \Delta_2}{\Delta_0 \vdash \mathcal{C}_1, \mathcal{C}_2 :: \Delta_2}$$

$$\frac{\Psi\,;\, \Delta \vdash P :: (c : A) \quad \sigma : \Psi}{\Delta', \Delta[\sigma] \vdash \mathit{pred}(c, P[\sigma]) :: (\Delta', c : A[\sigma])} \qquad \mathit{pred} ::= \mathsf{proc} \mid \mathsf{msg}^+ \mid \mathsf{msg}^-$$

To characterize observational equivalence of processes, we need to first characterize the possible messages and the direction in which they flow: towards the client (channel type is positive) or towards the provider (channel type is negative). We summarize the possible messages in Fig. 12. In each case, c is the channel along which the message is transmitted, and c' is the continuation channel.

Defining equivalence The notion of observational equivalence we need does not observe "nontermination", that is, it only compares messages that are actually received. Since messages can flow in two directions, we need to observe messages that arrive at either end. We therefore do *not* require, as is typical for bisimulation, that if one configuration takes a step, another configuration can also take a step. Instead we say if both configurations send an externally visible message, then the messages must be equivalent.

Supposing $\Gamma \vdash \mathcal{C} :: \Delta$ and $\Gamma \vdash \mathcal{D} :: \Delta$, we write $\Gamma \vdash \mathcal{C} \sim \mathcal{D} :: \Delta$ for our notion of observational equivalence. It is the largest relation satisfying that $\Gamma \vdash \mathcal{C} \sim \mathcal{D} :: \Delta$ implies

1. If $\Gamma' \vdash \mathsf{msg}^+(c, P) :: \Gamma$ then $\Gamma' \vdash (\mathsf{msg}^+(c, P), \mathcal{C}) \sim (\mathsf{msg}^+(c, P), \mathcal{D}) :: \Delta$.
2. If $\Delta \vdash \mathsf{msg}^-(c, P) :: \Delta'$ then $\Gamma \vdash (\mathcal{C}, \mathsf{msg}^-(c, P)) \sim (\mathcal{D}, \mathsf{msg}^-(c, P)) :: \Delta'$.
3. If $\mathcal{C} = (\mathcal{C}', \mathsf{msg}^+(c, P))$ with $\Gamma \vdash \mathcal{C}' :: \Delta'_1$ and $\Delta'_1 \vdash \mathsf{msg}^+(c, P) :: \Delta$
   and $\mathcal{D} = (\mathcal{D}', \mathsf{msg}^+(c, Q))$ with $\Gamma \vdash \mathcal{D}' :: \Delta'_2$ and $\Delta'_2 \vdash \mathsf{msg}^+(c, Q) :: \Delta$
   then $\Delta'_1 = \Delta'_2 = \Delta'$ and $P = Q$ and $\Gamma \vdash \mathcal{C}' \sim \mathcal{D}' :: \Delta'$.
4. If $\mathcal{C} = (\mathsf{msg}^-(c, P), \mathcal{C}')$ with $\Gamma \vdash \mathsf{msg}^-(c, P) :: \Gamma'_1$ and $\Gamma'_1 \vdash \mathcal{C}' :: \Delta$
   and $\mathcal{D} = (\mathsf{msg}^-(c, Q), \mathcal{D}')$ with $\Gamma \vdash \mathsf{msg}^-(c, Q) :: \Gamma'_2$ and $\Gamma'_2 \vdash \mathcal{D}' :: \Delta$
   then $\Gamma'_1 = \Gamma'_2 = \Gamma'$ and $P = Q$ and $\Gamma' \vdash \mathcal{C}' \sim \mathcal{D}' :: \Delta$.
5. If $\mathcal{C} \longrightarrow \mathcal{C}'$ then $\Gamma \vdash \mathcal{C}' \sim \mathcal{D} :: \Delta$
6. If $\mathcal{D} \longrightarrow \mathcal{D}'$ then $\Gamma \vdash \mathcal{C} \sim \mathcal{D}' :: \Delta$

Clauses (1) and (2) correspond to absorbing a message into a configuration, which may later be received by a process according to clauses (5) and (6). Clauses (3) and (4) correspond to observing messages, either by a client (clause (3)) or provider (clause (4)).

In clause (3) we take advantage of the property that a new continuation channel in the message P (one that does not appear already in Γ) is always chosen fresh when created, so we can consistently (and silently) rename it in $\mathcal{C}'$, Δ_1', and P (and $\mathcal{D}'$, Δ_2', and Q, respectively). This slight of hand allows us to match up the context and messages exactly. An analogous remark applies to clause (4). A more formal description would match up the contexts and messages modulo two renaming substitution which allow us to leave Γ and Δ fixed.

Clauses (5) and (6) make sense because a transition never changes the interface to a configuration, except when executing a forwarding $\mathsf{proc}(a, a \leftarrow b)$ which substitutes b for a in the remaining configuration. We can absorb this renaming into the renaming substitution. Cut creates a new channel, which remains internal since it is linear and will have one provider and one client within the new configuration. Unfortunately, our notation is already somewhat unwieldy and carrying additional renaming substitutions further obscures matters. We therefore omit them in this presentation.

The relation We now need to define a relation $\sim_M$ such that (a) it satisfies the closure conditions of $\sim$ and is therefore an observational equivalence, and (b) allows us to conclude that monitors satisfying our judgment are partial identities. Unfortunately, the theorem is rather complicated, so we will walk the reader through a sequence of generalizations that account for various complexities.

In order to handle the $\oplus$ and $\&$ connectives, we do not need value variables, nor do we need to pass channels. Then the top-level properties we would like to show are

(1^+) If $(y : A^+)\,;\cdot\,;\cdot \vdash P :: (x : A^+)[\,]$
then $y : A^+ \vdash \mathsf{proc}(x, x \leftarrow y) \sim_M P :: (x : A^+)$
(1^-) If $[\,](y : A^-)\,;\cdot\,;\cdot \vdash P :: (x : A^-)$
then $y : A^- \vdash \mathsf{proc}(x, x \leftarrow y) \sim_M P :: (x : A^-)$

Of course, asserting that $\mathsf{proc}(x, x \leftarrow y) \sim_M P$ will be insufficient, because this relation is not closed under the conditions of observational equivalence. For example, if we add a message along y to both sides, P will change its state once it receives the message, and the queue will record that this message still has to be sent. To generalize this, we need to define the queue that corresponds to a sequence of messages. First, a single message: We extend this to message sequences with $\langle\!\langle\,\rangle\!\rangle = (\cdot)$ and $\langle\!\langle \mathcal{E}_1, \mathcal{E}_2 \rangle\!\rangle = \langle\!\langle \mathcal{E}_1 \rangle\!\rangle \cdot \langle\!\langle \mathcal{E}_2 \rangle\!\rangle$, provided $\Delta_0 \vdash \mathcal{E}_1 :: \Delta_1$ and $\Delta_1 \vdash \mathcal{E}_2 :: \Delta_2$.

Then we build into the relation that sequences of messages correspond to the queue.

(2^+) If $(y{:}B^+)\,;\cdot\,;\cdot\,;\cdot \vdash P :: (x{:}A^+)[\langle\!\langle \mathcal{E} \rangle\!\rangle]$ then $y : B^+ \vdash \mathcal{E} \sim_M \mathsf{proc}(x, P) :: (x : A^+)$.
(2^-) If $[\langle\!\langle \mathcal{E} \rangle\!\rangle](y{:}B^-)\cdot\,;\cdot\,;\cdot \vdash P :: (x{:}A^-)$ then $y{:}B^- \vdash \mathcal{E} \sim_M \mathsf{proc}(x, P) :: (x{:}A^-)$.

When we add shifts the two propositions become mutually dependent, but otherwise they remain the same since the definition of $\langle\!\langle \mathcal{E} \rangle\!\rangle$ is already general enough. But we need to generalize the type on the opposite side of the queue to be either positive or negative, because it switches polarity after a shift has been received. Similarly, the channel might terminate when receiving $\mathbf{1}$, so we also need to allow ω, which is either empty or of the form $y : B$.

(3^+) If $\omega\,;\cdot\,;\cdot\,;\cdot \vdash P :: (x{:}A^+)[\langle\!\langle \mathcal{E} \rangle\!\rangle]$ then $\omega \vdash \mathcal{E} \sim_M \mathsf{proc}(x, P) :: (x{:}A^+)$.
(3^-) If $[\langle\!\langle \mathcal{E} \rangle\!\rangle](y{:}B^-)\,;\cdot\,;\cdot\,;\cdot \vdash P :: (x{:}A^+)$ then $y{:}B^- \vdash \mathcal{E} \sim_M \mathsf{proc}(x, P) :: (x{:}A)$.

Next, we can permit local state in the monitor (rules c_1^+ and c_1^-). The fact that neither of the two critical endpoints y and x, nor any (non-local) channels, can appear in the typing of the local process is key. That local process will evolve to a local configuration, but its interface will not change and it cannot access externally visible channels. So we generalize to allow a configuration $\mathcal{D}$ that does not use any channels, and any channels it offers are used by P.

(4^+) If $\omega\,;\cdot\,;\cdot\,;\Delta \vdash P :: [\langle\!\langle \mathcal{E} \rangle\!\rangle](x : A^+)$ and $\cdot \vdash \mathcal{D} :: \Delta$ then $\omega \vdash \mathcal{E} \sim_M \mathcal{D}, \mathsf{proc}(x, P) :: [q](x : A^+)$.
(4^-) If $[\langle\!\langle \mathcal{E} \rangle\!\rangle](y : B^-)\,;\cdot\,;\cdot\,;\Delta \vdash P :: (x : A)$ and $\cdot \vdash \mathcal{D} :: \Delta$ then $\Gamma, y : B^- \vdash \mathcal{E} \sim_M \mathcal{D}, \mathsf{proc}(x, P) :: (x : A)$.

Next, we can allow value variables necessitated by the universal and existential quantifiers. Since they are potentially dependent, we need to apply the closing substitution σ to a number of components in our relation.

(5^+) If $\omega\,;\Psi\,;\cdot\,;\Delta \vdash P :: [q](x : A^+)$ and $\sigma : \Psi$ and $q[\sigma] = \langle\!\langle \mathcal{E} \rangle\!\rangle$ and $\cdot \vdash \mathcal{D} :: \Delta[\sigma]$ then $\omega[\sigma] \vdash \mathcal{E} \sim_M \mathcal{D}, \mathsf{proc}(x, P[\sigma]) :: (x : A^+[\sigma])$.
(5^-) If $[q](y : B^-)\,;\Psi\,;\cdot\,;\Delta \vdash P :: (x : A)$ and $\sigma : \Psi$ and $q[\sigma] = \mathcal{E}$ and $\cdot \vdash \mathcal{D} :: \Delta[\sigma]$ then $y : B^-[\sigma] \vdash \mathcal{E} \sim_M \mathcal{D}, \mathsf{proc}(x, P[\sigma]) :: (x : A[\sigma])$.

Breaking up the queue by spawning a sequence of monitors (rule c_2^+ and c_2^-) just comes down to the compositionality of the partial identity property. This is a new and separate way that two configurations might be in the $\sim_M$ relation, rather than a replacement of a previous definition.

(6) If $\omega \vdash \mathcal{E}_1 \sim_M \mathcal{D}_1 :: (z : C)$ and $(z : C) \vdash \mathcal{E}_2 \sim_M \mathcal{D}_2 :: (x : A)$ then $\omega \vdash (\mathcal{E}_1, \mathcal{E}_2) \sim_M (\mathcal{D}_1, \mathcal{D}_2) :: (x : A)$.

At this point, the only types that have not yet accounted for are $\otimes$ and $\multimap$. If these channels were only "passed through" (without the four c_3 rules), this would be rather straightforward. However, for higher-order channel-passing programs, a monitor must be able to spawn a monitor on a channel that it receives before sending on the monitored version. First, we generalize properties (5) to allow the context Γ of channels that may occur in the queue q and the process P, but that P may not interact with.

(7^+) If $\omega \ ; \ \Psi \ ; \ \Gamma \ ; \ \Delta \vdash P :: [q](x : A^+)$ and $\sigma : \Psi$ and $q[\sigma] = \langle\!\langle \mathcal{E} \rangle\!\rangle$ and $\cdot \vdash \mathcal{D} :: \Delta[\sigma]$ then $\Gamma[\sigma], \omega[\sigma] \vdash \mathcal{E} \sim_M \mathcal{D}, \mathsf{proc}(x, P[\sigma]) :: (x : A^+[\sigma])$.
(7^-) If $[q](y : B^-) \ ; \ \Psi \ ; \ \Gamma \ ; \ \Delta \vdash P :: (x : A)$ and $\sigma : \Psi$ and $q[\sigma] = \mathcal{E}$ and $\cdot \vdash \mathcal{D} :: \Delta[\sigma]$ then $\Gamma[\sigma], y : B^-[\sigma] \vdash \mathcal{E} \sim_M \mathcal{D}, \mathsf{proc}(x, P[\sigma]) :: (x : A[\sigma])$.

In addition we need to generalize property (6) into (8) and (9) to allow multiple monitors to run concurrently in a configuration.

(8) If $\Gamma \vdash \mathcal{E} \sim_M \mathcal{D} :: \Delta$ then $(\Gamma', \Gamma) \vdash \mathcal{E} \sim_M \mathcal{D} :: (\Gamma', \Delta)$.
(9) If $\Gamma_1 \vdash \mathcal{E}_1 \sim_M \mathcal{D}_1 :: \Gamma_2$ and $\Gamma_2 \vdash \mathcal{E}_2 \sim_M \mathcal{D}_2 :: \Gamma_3$ then $\Gamma_1 \vdash (\mathcal{E}_1, \mathcal{E}_2) \sim_M (\mathcal{D}_1, \mathcal{D}_2) :: \Gamma_3$.

At this point we can state the main theorem regarding monitors.

Theorem 4.1. *If* $\Gamma \vdash \mathcal{E} \sim_M \mathcal{D} :: \Delta$ *according to properties* (7^+), (7^-), (8), *and* (9) *then* $\Gamma \vdash \mathcal{E} \sim \mathcal{D} :: \Delta$.

Proof. By closure under conditions 1-6 in the definition of $\sim$. □

By applying it as in equations (1^+) and (1^-), generalized to include value variables as in (5^+) and (5^-) we obtain that if $[\,](b : A^-) \ ; \ \Psi \vdash P :: (a : A^-)$ or $(b : A^+) \ ; \ \Psi \vdash P :: [\,](a : A^+)$ then P is a partial identity process.

5. Refinements as contracts

In this section we show how to check refinement types dynamically using our contracts. We encode refinements as type casts, which allows processes to remain well-typed with respect to the non-refinement type system (Section 3). These casts are translated at run time to monitors that validate whether the cast expresses an appropriate refinement. If so, the monitors behave as identity processes; otherwise, they raise an alarm and abort. For refinement contracts, we can prove a safety theorem, analogous to the classic "Well-typed Programs Can't be Blamed" [15], stating that if a monitor enforces a contract that casts from type A to type B, where A is a subtype of B, then this monitor will never raise an alarm.

5.1. Syntax and typing rules

We first augment messages and processes to include casts as follows. We write $\langle A \Leftarrow B \rangle^\rho$ to denote a cast from session type B to type A, where ρ is a unique label for the cast. The cast for values is written as $(\langle \tau \Leftarrow \tau' \rangle^\rho)$. A cast for values is necessary in order to coerce values as they are sent and received over channels. In this section, we treat the types τ' and τ as refinement types of the form $\{n{:}t \mid b\}$, where b is a boolean expression that expresses simple properties of the value n. We express any unrefined type τ by writing $\{n : t \mid \mathsf{true}\}$ which is compatible with earlier sections of this paper. For example, the following cast attempts to coerce integer values to positive integer values:
$\langle \{n : \mathsf{int} \mid n > 0\} \Leftarrow \{n : \mathsf{int}\} \rangle^\rho$.

$$P \ ::= \ \cdots \mid x \leftarrow \langle \tau \Leftarrow \tau' \rangle^\rho \ v \ ; \ Q \mid a{:}A \leftarrow \langle A \Leftarrow B \rangle^\rho \ b$$

Both of the additional rules to type casts are shown below. In the val_cast rule, the context Ψ stores values and their (refined) types. We only allow casts between two types that are compatible with each other (written $A \sim B$), which is co-inductively defined based on the structure of the types. The full definition is shown in Fig. 13.

$$\frac{A \sim B}{\Psi \ ; \ b : B \vdash a \leftarrow \langle A \Leftarrow B \rangle^\rho \ b :: (a : A)} \ \mathsf{id\_cast}$$

$$\frac{\Psi \vdash v : \tau' \quad \Psi, x : \tau \ ; \ \Delta \vdash Q :: (c : C) \quad \tau \sim \tau'}{\Psi \ ; \ \Delta \vdash x \leftarrow \langle \tau \Leftarrow \tau' \rangle^\rho \ v \ ; \ Q :: (c : C)} \ \mathsf{val\_cast}$$

$$\frac{}{\{n{:}\tau \mid b_1\} \sim \{n{:}\tau \mid b_2\}}\ \text{base} \qquad \frac{}{1 \sim 1}\ 1$$

$$\frac{A \sim A' \quad B \sim B'}{A \otimes B \sim A' \otimes B'}\ \otimes \qquad \frac{A' \sim A \quad B \sim B'}{A \multimap B \sim A' \multimap B'}\ \multimap$$

$$\frac{A_k \sim A'_k \text{ for } k \in I \cap J}{\oplus\{\ell_k : A_k\}_{k\in I} \sim \oplus\{\ell_k : A'_k\}_{k\in J}}\ \oplus \qquad \frac{A_k \sim A'_k \text{ for } k \in I \cap J}{\&\{\ell_k : A_k\}_{k\in I} \sim \&\{\ell_k : A'_k\}_{k\in J}}\ \&$$

$$\frac{A \sim B}{\downarrow A \sim \downarrow B}\ \downarrow \qquad \frac{A \sim B}{\uparrow A \sim \uparrow B}\ \uparrow$$

$$\frac{A \sim B \quad \tau_1 \sim \tau_2}{\exists n : \tau_1.A \sim \exists n : \tau_2.B}\ \exists \qquad \frac{A \sim B \quad \tau_1 \sim \tau_2}{\forall n : \tau_1.A \sim \forall n : \tau_2.B}\ \forall$$

Fig. 13. Compatibility.

one : $[\![\langle \mathbf{1} \Leftarrow \mathbf{1}\rangle^\rho]\!]_{a,b} = \text{wait } b; \text{close } a$

lolli : $[\![\langle A_1 \multimap A_2 \Leftarrow B_1 \multimap B_2\rangle^\rho]\!]_{a,b} =$
$x \leftarrow \text{recv } a\ ;\ y \leftarrow [\![\langle B_1 \Leftarrow A_1\rangle^\rho]\!]_{y,x} \leftarrow x\ ;\ \text{send } b\ y\ ;\ [\![\langle A_2 \Leftarrow B_2\rangle^\rho]\!]_{a,b}$

tensor : $[\![\langle A_1 \otimes A_2 \Leftarrow B_1 \otimes B_2\rangle^\rho]\!]_{a,b} =$
$x \leftarrow \text{recv } b\ ;\ y \leftarrow [\![\langle A_1 \Leftarrow B_1\rangle^\rho]\!]_{y,x} \leftarrow x\ ;\ \text{send } a\ y\ ;\ [\![\langle A_2 \Leftarrow B_2\rangle^\rho]\!]_{a,b}$

forall : $[\![\langle \forall\{n : \tau \mid e\}.\, A \Leftarrow \forall\{n : \tau' \mid e'\}.\, B\rangle^\rho]\!]_{a,b} =$
$n \leftarrow \text{recv } a\ ;\ \text{assert } \rho\, e'(x)\ ;\ \text{send } b\ x;\ [\![\langle A \Leftarrow B\rangle^\rho]\!]_{a,b}$

exists : $[\![\langle \exists\{n : \tau \mid e\}.\, A \Leftarrow \exists\{n : \tau' \mid e'\}.\, B\rangle^\rho]\!]_{a,b} =$
$n \leftarrow \text{recv } b\ ;\ \text{assert } \rho\, e(x)\ ;\ \text{send } a\ x;\ [\![\langle A \Leftarrow B\rangle^\rho]\!]_{a,b}$

up : $[\![\langle \uparrow A \Leftarrow \uparrow B\rangle^\rho]\!]_{a,b} =$
$\text{shift} \leftarrow \text{recv } b\ ;\ \text{send } a\ \text{shift}\ ;\ [\![\langle A \Leftarrow B\rangle^\rho]\!]_{a,b}$

down : $[\![\langle \downarrow A \Leftarrow \downarrow B\rangle^\rho]\!]_{a,b} =$
$\text{shift} \leftarrow \text{recv } a\ ;\ \text{send } b\ \text{shift}\ ;\ [\![\langle A \Leftarrow B\rangle^\rho]\!]_{a,b}$

plus : $[\![\langle \oplus\{\ell : A_\ell\}_{\ell\in I} \Leftarrow \oplus\{\ell : B_\ell\}_{\ell\in J}\rangle^\rho]\!]_{a,b} =$
$\text{case } b\ (\ell \Rightarrow Q_\ell)_{\ell\in I}$
$\text{where } \forall \ell, \ell \in I \cap J,\ a.\ell\ ;\ [\![\langle A_\ell \Leftarrow B_\ell\rangle^\rho]\!]_{a,b} = Q_\ell$
$\text{and } \forall \ell, \ell \in J \wedge \ell \notin I,\ Q_\ell = \text{abort } \rho$

with : $[\![\langle \&\{\ell : A_\ell\}_{\ell\in I} \Leftarrow \&\{\ell : B_\ell\}_{\ell\in J}\rangle^\rho]\!]_{a,b} =$
$\text{case } a\ (\ell \Rightarrow Q_\ell)_{\ell\in I}$
$\text{where } \forall \ell, \ell \in I \cap J,\ b.\ell\ ;\ [\![\langle A_\ell \Leftarrow B_\ell\rangle^\rho]\!]_{a,b} = Q_\ell$
$\text{and } \forall \ell, \ell \in I \wedge \ell \notin J,\ Q_\ell = \text{abort } \rho$

Fig. 14. Cast translation.

5.2. Translation to monitors

A cast $a \leftarrow \langle A \Leftarrow B\rangle^\rho\ b$ can be implemented as a monitor. This monitor ensures that the process that offers a service on channel b behaves according to the prescribed type A. Because of the typing rules, we are assured that channel b must adhere to the type B.

Fig. 14 is a summary of all the translation rules, except recursive types. The translation is of the form: $[\![\langle A \Leftarrow B\rangle^\rho]\!]_{a,b} = P$, where A, B are types; the channels a and b are the offering channel and monitoring channel (respectively) for the resulting monitoring process P; and ρ is the label of the monitor (i.e., the contract). While this approach differs from blame labels for higher-order functions, where the monitor carries two labels, one for the argument, and one for the body of the function, blaming a cast label is common in the context of cast calculi for gradual typing.

In our setting, the communication between processes is bi-directional. Though blame is always triggered by a process sending messages to the monitor, we do not automatically blame the process that sent the offending messages. In the case of forwarding, the processes at either end of the channel could be behaving according to the types (contracts) assigned to them, but the cast connecting them could have incompatible types. To handle this situation, we always blame the label of the failed cast.

This version of blame is less precise than other treatments of blame (including prior work on blame assignment in functional settings [16], and our prior work on blame for session types [1]). For example, consider a process P_1 that sends a channel x, which has a cast from integers to positive integers with label ρ, to process P_2. In this situation, process P_1 seems to be at fault for delegating a channel with a bad cast. However, an alarm will only be triggered when a negative integer is sent over channel x, and the label ρ will be blamed.

The translation is defined inductively over the structure of the types. The exists rule generates a process that first receives a value from the channel b, then checks the boolean condition e to validate the contract. For example, a translation of the cast: $[\![\langle \{\exists.n : \text{int}.A\} \Leftarrow \{\exists.n : \text{int} \mid n > 0\}.B\rangle^\rho]\!]_{a,b}$ would produce: $x \leftarrow \text{recv } b; \text{assert } (x > 0); \text{send } a\ x; [\![\langle \{A\} \Leftarrow \{B\}\rangle^\rho]\!]_{a,b}$. The

$$\frac{}{1 \leq 1}\ 1 \qquad \frac{A \leq A' \quad B \leq B'}{A \otimes B \leq A' \otimes B'}\ \otimes \qquad \frac{A' \leq A \quad B \leq B'}{A \multimap B \leq A' \multimap B'}\ \multimap$$

$$\frac{A_k \leq A'_k \text{ for } k \in J \quad J \subseteq I}{\oplus\{\ell_k : A_k\}_{k\in J} \leq \oplus\{\ell_k : A'_k\}_{k\in I}}\ \oplus \qquad \frac{A_k \leq A'_k \text{ for } k \in J \quad I \subseteq J}{\&\{\ell_k : A_k\}_{k\in J} \leq \&\{\ell_k : A'_k\}_{k\in I}}\ \&$$

$$\frac{A \leq B}{\downarrow A \ \leq \downarrow B}\ \downarrow \qquad \frac{A \leq B}{\uparrow A \ \leq \uparrow B}\ \uparrow \qquad \frac{\mathsf{def}(A) \leq \mathsf{def}(B)}{A \leq B}\ \mathsf{def}$$

$$\frac{A \leq B \quad \tau_1 \leq \tau_2}{\exists n : \tau_1.A \leq \exists n : \tau_2.B}\ \exists \qquad \frac{A \leq B \quad \tau_2 \leq \tau_1}{\forall n : \tau_1.A \leq \forall n : \tau_2.B}\ \forall$$

$$\frac{\tau_1 \leq \tau_2 \quad \forall v{:}\tau_1, [v/x]b_1 \mapsto^* \mathsf{true} \text{ implies } [v/x]b_2 \mapsto^* \mathsf{true}}{\{x{:}\tau_1 \mid b_1\} \leq \{x{:}\tau_2 \mid b_2\}}\ \mathsf{refine}$$

Fig. 15. Subtyping.

forall rule is similar, except the argument position is contravariant, so the boolean expression e' is checked on the offering channel a.

The tensor rule generates a process that first receives a channel (x) from the channel being monitored (b), then spawns a new monitor to monitor x, making sure that it behaves as type A_1. Then, it passes the new monitor's offering channel y to a. Finally, the monitor continues to monitor b to make sure that it behaves as type A_2. The lolli rule is similar to the tensor rule, except that the monitor first receives a channel from its offering channel. Similar to the higher-order function case, the argument position is contravariant, so the newly spawned monitor checks that the received channel behaves as type B_1. The with rule generates a process that checks that all of the external choices promised by the type $\&\{\ell : A_\ell\}_{\ell\in I}$ are offered by the process being monitored. If a label in the set I is not implemented, then the monitor aborts with the label ρ. The plus rule requires that, for internal choices, the monitor checks that the monitored process only offers choices within the labels in the set $\oplus\{\ell : A_\ell\}_{\ell\in I}$.

We translate casts with recursive types as follows. For each pair of compatible recursive types A and B, we generate a unique monitor name f and record its type $f : \{A \leftarrow B\}$ in a context Σ. The translation algorithm needs to take additional arguments, including Σ to generate and invoke the appropriate recursive process when needed. For instance, when generating the monitor process for $f : \{\mathsf{list} \leftarrow \mathsf{list}\}$, we follow the rule for translating internal choices. For $[\![\langle \mathsf{list} \Leftarrow \mathsf{list}\rangle^\rho]\!]_{y,x}$ we apply the cons case in the translation to get $y \leftarrow f \leftarrow x$.

5.3. Safety and transparency of casts

We prove two formal properties of cast-based monitors: safety, and transparency. We also prove preservation in the presence of well-typed casts. Because of the expressiveness of our contracts, a general safety (or blame) theorem is difficult to achieve. However, for cast-based contracts, we can prove that a cast which enforces a subtyping relation, and the corresponding monitor, will not raise an alarm.

We first define our subtyping relation in Fig. 15. In addition to the subtyping between refinement types, we also include label subtyping for our session types. A process that offers more external choices can always be used as a process that offers fewer external choices. Similarly, a process that offers fewer internal choices can always be used as a process that offers more internal choices (e.g., non-empty list can be used as a list). The subtyping rules for internal and external choices are drawn from work by Gay and Hole [12]. For recursive types, we directly examine their definitions. Because of these recursive types, our subtyping rules are co-inductively defined.

Our safety theorem guarantees that well-typed casts do not raise alarms. The key is to show that the monitor process generated from the translation algorithm in Fig. 14 is well-typed under a typing relation which guarantees that no abort l state can be reached. The type system presented so far in the paper (summarized in Fig. 10) allows a monitor that may evaluate to abort l to be typed. We will refer to this type system as T. We define a stronger type system S which consists of the rules in Fig. 10 with the exception of the abort rule and we replace the assert rule with the assert_strong rule. This new rule verifies that the condition b is true using the fact that the refinements are stored in the context Ψ.

$$\frac{\Psi \vDash b\ \mathsf{true} \quad \Psi\ ;\ \Delta \vdash Q :: (x : A)}{\Psi\ ;\ \Delta \vdash \mathsf{assert}\ \rho\ b;\ Q :: (x : A)}\ \mathsf{assert\_strong}$$

Theorem 5.1 *(Monitors are well-typed).*

1. $b : B \vdash_T [\![\langle A \Leftarrow B\rangle^\rho]\!]^\Psi_{a,b} :: (a : A)$.
2. *If* $B \leq A$, *then* $b : B \vdash_S [\![\langle A \Leftarrow B\rangle^\rho]\!]^\Psi_{a,b} :: (a : A)$.

Proof. The proof is by induction over the monitor translation rules. To prove (1) we use type system T. To prove (2) we use type system S and the sub-typing relation to show that (a) for the internal and external choice cases, no branches that include abort are generated; and (b) for the forall and exists cases, the assert never fails (i.e., the assert_strong rule applies). We present some representative cases.
Case 1:
$[\![\langle \mathbf{1} \Leftarrow \mathbf{1} \rangle^{\rho}]\!]_{a,b} = \mathsf{wait}\, b; \mathsf{close}\, a$
This process is well-typed with respect to both S and T by applying the 1L and 1R rules.

Case $\multimap$:
$[\![\langle A_1 \multimap A_2 \Leftarrow B_1 \multimap B_2 \rangle^{\rho}]\!]_{a,b} = x \leftarrow \mathsf{recv}\, a\, ;\, y \leftarrow [\![\langle B_1 \Leftarrow A_1 \rangle^{\rho}]\!]_{y,x} \leftarrow x\, ;\, \mathsf{send}\, b\, y;\, ;\, [\![\langle A_2 \Leftarrow B_2 \rangle^{\rho}]\!]_{a,b}$
(1) By I.H, we know that $E' = \Psi\, ;\, x : A_1 \vdash_T [\![\langle B_1 \Leftarrow A_1 \rangle^{\rho}]\!]^{\Psi}_{y,x} :: (y : B_1)$ and $E'' = \Psi\, ;\, b : B_2 \vdash_T [\![\langle A_2 \Leftarrow B_2 \rangle^{\rho}]\!]^{\Psi}_{a,b} :: (a : A_2)$. Apply the $\multimap$L rule to E'' to typecheck the send. Then apply the $\multimap$R rule to typecheck the receive. Therefore, this process is well-typed with respect to T.
(2) We have that $B_1 \multimap B_2 \leq A_1 \multimap A_2$. By subtyping, this means that $A_1 \leq B_1$ and $B_2 \leq A_2$. By I.H., we know that $E' = \Psi\, ;\, x : A_1 \vdash_S [\![\langle B_1 \Leftarrow A_1 \rangle^{\rho}]\!]^{\Psi}_{y,x} :: (y : B_1)$ and $E'' = \Psi\, ;\, b : B_2 \vdash_S [\![\langle A_2 \Leftarrow B_2 \rangle^{\rho}]\!]^{\Psi}_{a,b} :: (a : A_2)$. Apply the $\multimap$L rule to E'' to typecheck the send. Then apply the $\multimap$R rule to typecheck the receive. Therefore, this process is well-typed with respect to S.

Case $\forall$:
$[\![\langle \forall\{n : \tau \mid e\}.\, A \Leftarrow \forall\{n : \tau' \mid e'\}.\, B \rangle^{\rho}]\!]_{a,b} = n \leftarrow \mathsf{recv}\, a;\, ;\, \mathsf{assert}\, \rho\, e'(x)\, ;\, \mathsf{send}\, b\, x;\, [\![\langle A \Leftarrow B \rangle^{\rho}]\!]_{a,b}$
(1) By I.H, we know that $E' = \Psi\, ;\, b : B \vdash_T [\![\langle A \Leftarrow B \rangle^{\rho}]\!]^{\Psi}_{a,b} :: (a : A)$. Apply the $\forall$L rule to E' to typecheck the send. Then, apply the assert rule to typecheck the assertion. Finally, apply the $\forall$R rule to typecheck the receive. Therefore, this process is well-typed with respect to T.
(2) Because $\forall\{n : \tau' \mid e'\}.B \leq \forall\{n : \tau' \mid e\}.A$ we know that $B \leq A$. By I.H., we know that $E' = \Psi\, ;\, b : B \vdash_S [\![\langle A \Leftarrow B \rangle^{\rho}]\!]^{\Psi}_{a,b} :: (a : A)$. Apply the $\forall$L rule to E' to typecheck the send. We now need to show that we can typecheck the assertion using the assert_strong rule. That is, we need to show that $e'(x)$ is true. From the subtyping refine rule, we see that for $x : \tau, [x/y]e \mapsto^* \mathsf{true}$ implies $x : \tau, [x/y]e' \mapsto^* \mathsf{true}$. Therefore, $e'(x)$ is true and will not abort. We then apply the assert_strong rule to typecheck the assertion. Finally, apply the $\forall$R rule to typecheck the receive. Therefore, this process is well-typed with respect to S.

Case $\oplus$:
$[\![\langle \oplus\{\ell : A_\ell\}_{\ell \in I} \Leftarrow \oplus\{\ell : B_\ell\}_{\ell \in J} \rangle^{\rho}]\!]_{a,b} = \mathsf{case}\, b\, (\ell \Rightarrow Q_\ell)_{\ell \in I}$ where $\forall \ell, \ell \in I \cap J,\ a.\ell\, ;\, [\![\langle A_\ell \Leftarrow B_\ell \rangle^{\rho}]\!]_{a,b} = Q_\ell$ and $\forall \ell, \ell \in J \wedge \ell \notin I,\ Q_\ell = \mathsf{abort}\ \rho$
(1) By I.H., we know that $E' = \Psi\, ;\, b : B_\ell \vdash_T [\![\langle A_\ell \Leftarrow B_\ell \rangle^{\rho}]\!]^{\Psi}_{a,b} :: (a : A_\ell)$. Apply the $\oplus$R rule to E' to typecheck sending the label $a.\ell$. Then apply the $\oplus$L rule to typecheck the case. Therefore, this process is well-typed with respect to T.
(2) Because $\oplus\{\ell : B_\ell\}_{\ell \in J} \leq \oplus\{\ell : A_\ell\}_{\ell \in I}$ we know that $B_k \leq A_k$. By I.H., we know that $E' = \Psi\, ;\, b : B_\ell \vdash_S [\![\langle A_\ell \Leftarrow B_\ell \rangle^{\rho}]\!]^{\Psi}_{a,b} :: (a : A_\ell)$. Apply the $\oplus$R rule to E' to typecheck sending the label $a.\ell$. Then apply the $\oplus$L rule to typecheck the case. From the subtyping, we also know that $J \subseteq I$. This means that there does not exist an $\ell \in J \wedge l \notin I$, and no abort branch will be generated. Therefore, this process is well-typed with respect to S. □

As a corollary, we can show that when executing in a well-typed context, a monitor process translated from a well-typed cast will never raise an alarm.

Corollary 5.2 *(Well-typed casts cannot raise alarms).* $\vdash \mathcal{C} :: (b : B)$ *and* $B \leq A$ *implies* $\mathcal{C}, \mathsf{proc}(a, [\![\langle A \Leftarrow B \rangle^{\rho}]\!]_{a,b}) \not\longrightarrow^* \mathsf{abort}\ \rho$.

Proof. By Theorem 5.1 we have that $\Psi\, ;\, b : B \vdash_S [\![\langle A \Leftarrow B \rangle^{\rho}]\!]^{\Psi}_{a,b} :: (a : A)$. Type system S does not allow aborts, so $\mathcal{C}, \mathsf{proc}(a, [\![\langle A \Leftarrow B \rangle^{\rho}]\!]_{a,b}) \not\longrightarrow^* \mathsf{abort}\ \rho$. □

Next, we prove that monitors translated from casts are partial identity processes.

Theorem 5.3 *(Casts are transparent).*
$b : B \vdash \mathsf{proc}(b, a \leftarrow b) \sim \mathsf{proc}(a, [\![\langle A \Leftarrow B \rangle^{\rho}]\!]_{a,b}) :: (a : A)$.

Proof. We just need to show that the translated process passes the partial identity checks. We can show this by induction over the translation rules and by applying the rules developed in Section 4.2 and Section 4.3. We note that rules in Section 4 only consider identical types; however, our casts only cast between two compatible types. Therefore, we can lift A and B to their super types (i.e., insert abort cases for mismatched labels), and then apply the checking rules. This does not change the semantics of the monitors. We present some representative cases.
Case 1:

$[\![\langle \mathbf{1} \Leftarrow \mathbf{1}\rangle^\rho]\!]_{a,b} = \mathsf{wait}\ b; \mathsf{close}\ a$
Apply the 1R rule and then the 1L rule to get $(b : 1); \Psi; \cdot; \cdot \vdash \mathsf{wait}\ b; \mathsf{close}\ b :: []\,(a : A)$.

Case $\otimes$:
$[\![\langle A_1 \otimes A_2 \Leftarrow B_1 \otimes B_2\rangle^\rho]\!]_{a,b} = x \leftarrow \mathsf{recv}\ b\ ;\ y \leftarrow [\![\langle A_1 \Leftarrow B_1\rangle^\rho]\!]_{y,x} \leftarrow x\ ;\ \mathsf{send}\ a\ y\ ;\ [\![\langle A_2 \Leftarrow B_2\rangle^\rho]\!]_{a,b}$
By I.H. we have that $b : B_2 \vdash \mathsf{proc}(b, a \leftarrow b) \sim \mathsf{proc}(a, [\![\langle A_2 \Leftarrow B_2\rangle^\rho]\!]_{a,b}) :: (a : A_2)$. We then apply the $\otimes$R rule. By I.H. we have that $x : B_1 \vdash \mathsf{proc}(x, y \leftarrow x) \sim \mathsf{proc}(y, [\![\langle A_1 \Leftarrow B_1\rangle^\rho]\!]_{y,x}) :: (y : A_1)$. We can now apply the c_{3++} rule. Finally, we apply the $\otimes$L rule. We then get $(b : B_1 \otimes B_2); \Psi; \Gamma; \Delta \vdash [\![\langle A_1 \otimes A_2 \Leftarrow B_1 \otimes B_2\rangle^\rho]\!]_{a,b} :: []\,(a : A_1 \otimes A_2)$.

Case $\exists$.
$[\![\langle \exists\{n : \tau \mid e\}.\ A \Leftarrow \exists\{n : \tau' \mid e'\}.\ B\rangle^\rho]\!]_{a,b} = n \leftarrow \mathsf{recv}\ b\ ;\ \mathsf{assert}\ \rho\ e(x)\ ;\ \mathsf{send}\ a\ x;\ [\![\langle A \Leftarrow B\rangle^\rho]\!]_{a,b}$
By I.H. we have that $b : B \vdash \mathsf{proc}(b, a \leftarrow b) \sim \mathsf{proc}(a, [\![\langle A \Leftarrow B\rangle^\rho]\!]_{a,b}) :: (a : A)$. We apply the $\exists$R and $\exists$L rules to get that $(b : \exists\{n : \tau' \mid e'\}.\ B); \Psi; \Gamma; \Delta \vdash [\![\langle \exists\{n : \tau \mid e\}.\ A \Leftarrow \exists\{n : \tau' \mid e'\}.\ B\rangle^\rho]\!]_{a,b} :: []\,(a : \exists\{n : \tau \mid e\}.\ A)$.

Case $\&$.
$[\![\langle \&\{\ell : A_\ell\}_{\ell\in I} \Leftarrow \&\{\ell : B_\ell\}_{\ell\in J}\rangle^\rho]\!]_{a,b} = \mathsf{case}\ a\ (\ell \Rightarrow Q_\ell)_{\ell\in I}$ where $\forall \ell, \ell \in I \cap J,\ b.\ell\ ;\ [\![\langle A_\ell \Leftarrow B_\ell\rangle^\rho]\!]_{a,b} = Q_\ell$ and $\forall \ell, \ell \in I \wedge \ell \notin J,\ Q_\ell = \mathsf{abort}\ \rho$ By I.H. we have that $b : B \vdash \mathsf{proc}(b, a \leftarrow b) \sim \mathsf{proc}(a, [\![\langle A_\ell \Leftarrow B_\ell\rangle^\rho]\!]_{a,b}) :: (a : A)$. We apply the $\&$L and $\&$R rules to get that $[q](b : \&\{\ell : B_\ell\}_{\ell\in L}); \Psi; \Gamma; \Delta \vdash [\![\langle \&\{\ell : A_\ell\}_{\ell\in L} \Leftarrow \&\{\ell : B_\ell\}_{\ell\in L}\rangle^\rho]\!]_{a,b} :: (a : \&\{\ell : A_\ell\}_{\ell\in L})$. □

We have now proved that casts are safe and transparent. Finally, we prove preservation for the system in the presence of casts. In order to do so, we first prove an inversion lemma and a substitution lemma that is valid in the presence of subtyping. Because we make the assumption that any cast $\langle A \Leftarrow B\rangle^\rho$ will be well-typed (defined as $B \le A$), we have to define substitution rules (shown in Fig. 16) and prove a substitution lemma in the presence of subtyping. We show this lemma below. We define $|E|$ to be the size of the derivation of E.

Lemma 5.4 *(Subtype-substitution).*

1. *If $E = \Psi; \Delta, x : A \vdash P :: (c : C)$ and $B \le A$ then for any fresh variable $y : B$ we have $E' = \Psi; \Delta, y : B \vdash [y : B/x : A]P :: (c : C)$ and $|E| = |E'|$.*
2. *If $E = \Psi, n : \tau; \Delta \vdash P :: (c : C)$ and $\tau' \le \tau$ then for any fresh variable $m : \tau'$ we have $E' = \Psi, m : \tau'; \Delta \vdash [m : \tau'/n : \tau]P :: (c : C)$ and $|E| = |E'|$*
3. *If $E = \Psi; \Delta \vdash P :: (x : A)$ and $A \le B$ then for any fresh variable $y : B$ we have $E' = \Psi; \Delta \vdash [y : B/x : A]P :: (y : B)$ and $|E| = |E'|$.*

We prove the lemma by performing an induction over the derivation of $\Psi; \Delta \vdash P :: (c : C)$. We present some representative cases. We note that we implicitly encode any channel $c : A$ or value $v : \tau$ as $\langle A \Leftarrow A\rangle^\rho c : A$ and $\langle \tau \Leftarrow \tau\rangle^\rho v : \tau$ respectively.

Proof. Case *id_cast*.

$$\frac{A \sim A'}{\Psi\ ;\ b : A' \vdash a \leftarrow \langle A \Leftarrow A'\rangle^\rho\ b :: (a : A)}\ \mathsf{id\_cast}$$

Subcase left. We have $A'' \le A'$. Let $g : A''$. When we perform the substitution $[g : A''/b : A']$ we update the cast $\langle A \Leftarrow A'\rangle^\rho$ to $\langle A \Leftarrow A''\rangle^\rho$. We then get $\Psi; g : A'' \vdash a \leftarrow \langle A \Leftarrow A''\rangle^\rho g :: (a : A)$ which matches our substitution rules.
Subcase right. We have $A \le A''$. Let $g : A''$. When we perform the substitution $[g : A''/b : A']$ we update the cast $\langle A \Leftarrow A'\rangle^\rho$ to $\langle A'' \Leftarrow A'\rangle^\rho$. We then get $\Psi; b : A' \vdash g \leftarrow \langle A'' \Leftarrow A'\rangle^\rho b :: (g : A'')$ which matches our substitution rules.

Case *val_cast*.

$$\frac{\Psi \vdash v : \tau' \quad E' = \Psi, x : \tau; \Delta \vdash Q :: (c : C) \quad \tau \sim \tau'}{\Psi; \Delta \vdash x \leftarrow \langle \tau \Leftarrow \tau'\rangle^\rho\ v; Q :: (c : C)}\ \mathsf{val\_cast}$$

We have $C \le C'$. Apply I.H. to E' to get E''. For some fresh $g : C'$ we have $E'' = \Psi, x : \tau; \Delta \vdash [g : C'/c : C]Q :: (g : C')$. Now apply the *val_cast* rule to E'' to get $\Psi; \Delta \vdash x \leftarrow \langle \tau \Leftarrow \tau'\rangle^\rho v; [g : C'/c : C]Q :: (g : C')$ which matches our substitution rules.

Case $\oplus$L.

$$E = \frac{E' = \Psi; \Delta, c : A_\ell \vdash Q_\ell :: (d : D) \quad \text{for every } \ell \in L}{\Psi; \Delta, c : \oplus\{\ell : A_\ell\}_{\ell\in L} \vdash \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell\in L} :: (d : D)}\ \oplus L$$

$$[c : C/b : B](\mathsf{proc}(a, a \leftarrow \langle A \Leftarrow B\rangle^{\rho}\ b)) = \mathsf{proc}(a, a \leftarrow \langle A \Leftarrow C\rangle^{\rho}\ c)$$
$$[c : C/a : A](\mathsf{proc}(a, a \leftarrow \langle A \Leftarrow B\rangle^{\rho}\ b)) = \mathsf{proc}(c, c \leftarrow \langle C \Leftarrow B\rangle^{\rho}\ b)$$
$$[h : A' \otimes B'/c : A \otimes B]\mathsf{proc}(d, x \leftarrow \mathsf{recv}\ c;\ Q) = \mathsf{proc}(d, g \leftarrow \mathsf{recv}\ h; [h : B'/c : B][g : A'/x : A]Q)$$
$$[f : A'' \otimes B'/c : A \otimes B]\mathsf{proc}(c, \mathsf{send}\ c\ \langle A \Leftarrow A'\rangle^{\rho}\ (a : A');\ P) = \mathsf{proc}(f, \mathsf{send}\ f\ \langle A'' \Leftarrow A'\rangle^{\rho}\ a;\ [f : B'/c : B]P)$$
$$[h : A' \multimap B'/c : A \multimap B]\mathsf{proc}(c, x \leftarrow \mathsf{recv}\ c;\ P) = \mathsf{proc}(h, g \leftarrow \mathsf{recv}\ h; [h : B'/c : B][g : A'/x : A]P)$$
$$[f : A'' \multimap B/c : A \multimap B]\mathsf{proc}(d, \mathsf{send}\ c\ \langle A \Leftarrow A'\rangle^{\rho}\ (a : A');\ Q) = \mathsf{proc}(d, \mathsf{send}\ f\ \langle A'' \Leftarrow A'\rangle^{\rho}\ a;\ [f : B'/c : B]Q)$$
$$[f : \oplus\{\ell : A'_\ell\}_{\ell\in L}/c : \oplus\{\ell : A_\ell\}_{\ell\in L}]\mathsf{proc}(d, \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell\in L}) = \mathsf{proc}(d, \mathsf{case}\ f\ (\ell \Rightarrow [f : A'_\ell/c : A_\ell]Q_\ell)_{\ell\in L})$$
$$[f : \oplus\{\ell : A'_\ell\}_{\ell\in L}/c : \oplus\{\ell : A_\ell\}_{\ell\in L}]\mathsf{proc}(c, c.k;\ P) = \mathsf{proc}(f, f.k; [f : A'_\ell/c : A_\ell]P)$$
$$[f : \&\{\ell : A'_\ell\}_{\ell\in L}/c : \&\{\ell : A_\ell\}_{\ell\in L}]\mathsf{proc}(d, c.k;\ Q) = \mathsf{proc}(d, f.k; [f : A'_\ell/A_\ell]Q)$$
$$[f : \&\{\ell : A'_\ell\}_{\ell\in L}/c : \&\{\ell : A_\ell\}_{\ell\in L}]\mathsf{proc}(c, \mathsf{case}\ c\ (\ell \Rightarrow P_\ell)_{\ell\in L}) = \mathsf{proc}(f, \mathsf{case}\ f\ (\ell \Rightarrow [f : A'_\ell/c : A_\ell]P_\ell)_{\ell\in L}))$$
$$[f : \exists n : \tau'.A'/c : \exists n : \tau.A]\mathsf{proc}(d, n \leftarrow \mathsf{recv}\ c;\ Q) = \mathsf{proc}(d, m \leftarrow \mathsf{recv} f; [m : \tau'/n : \tau][f : A'/c : A]Q)$$
$$[f : \exists n : \tau''.A'/c : \exists n : \tau.A]\mathsf{proc}(c, \mathsf{send}\ c\ \langle \tau \Leftarrow \tau'\rangle^{\rho}\ (v : \tau');\ P) = \mathsf{proc}(f, \mathsf{send}\ f\ \langle \tau'' \Leftarrow \tau'\rangle^{\rho}\ (v : \tau');\ [f : A'/c : A]P)$$
$$[f : \forall n : \tau''.A'/c : \forall n : \tau.A]\mathsf{proc}(d, \mathsf{send}\ c\ \langle \tau \Leftarrow \tau'\rangle^{\rho}\ (v : \tau');\ Q) = \mathsf{proc}(d, \mathsf{send}\ f\ \langle \tau'' \Leftarrow \tau'\rangle^{\rho}(v : \tau');\ [f : A'/c : A]Q)$$
$$[f : \forall n : \tau'.A'/c : \forall n : \tau.A]\mathsf{proc}(c, n \leftarrow \mathsf{recv}\ c;\ P) = \mathsf{proc}(f, m \leftarrow \mathsf{recv} f; [m : \tau'/n : \tau][f : A'/c : A]\ ;\ P$$
$$[f : \downarrow A'/c : \downarrow A]\mathsf{proc}(c, \mathsf{send}\ c\ \mathsf{shift};\ P) = \mathsf{proc}(f, \mathsf{send}\ f\ \mathsf{shift}; [f : A'/c : A]P)$$
$$[f : \downarrow A'/c : \downarrow A]\mathsf{proc}(d, \mathsf{shift} \leftarrow \mathsf{recv}\ d;\ Q) = \mathsf{proc}(d, \mathsf{shift} \leftarrow \mathsf{recv}\ d; [f : A'/c : A]Q)$$
$$[f : \uparrow A'/c : \uparrow A]\mathsf{proc}(d, \mathsf{send}\ d\ \mathsf{shift};\ Q) = \mathsf{proc}(d, \mathsf{send}\ d\ \mathsf{shift}; [f : A'/c : A]Q)$$
$$[f : \uparrow A'/c : \uparrow A]\mathsf{proc}(c, \mathsf{shift}\ \mathsf{recv}\ c;\ P) = \mathsf{proc}(f, \mathsf{shift}\ \mathsf{recv}\ f; [f : A'/c : A]P)$$
$$[m : \tau''/v : \tau'](\mathsf{proc}(a, x \leftarrow \langle \tau \Leftarrow \tau'\rangle^{\rho}\ v;\ Q) = \mathsf{proc}(a, x \leftarrow \langle \tau \Leftarrow \tau''\rangle^{\rho}\ m;\ Q)$$
$$[k : A''/a : A']\mathsf{msg}(c, \mathsf{send}\ c\ \langle A \Leftarrow A'\rangle^{\rho} a\ ;\ c \leftarrow c') = \mathsf{msg}(c, \mathsf{send}\ c\ \langle A \Leftarrow A''\rangle^{\rho} k\ ;\ c \leftarrow c')$$
$$[k : A''/c : A]\mathsf{msg}(c, \mathsf{send}\ c\ \langle A \Leftarrow A'\rangle^{\rho} a\ ;\ c \leftarrow c') = \mathsf{msg}(k, \mathsf{send}\ k\ \langle A'' \Leftarrow A'\rangle^{\rho} a\ ;\ k \leftarrow c')$$
$$[m : \tau''/v : \tau']\mathsf{msg}(c, \mathsf{send}\ c\ \langle \tau \Leftarrow \tau'\rangle^{\rho} v\ ;\ c' \leftarrow c) = \mathsf{msg}(c, \mathsf{send}\ c\ \langle \tau \Leftarrow \tau''\rangle^{\rho} m\ ;\ c' \leftarrow c)$$

Fig. 16. Substitution rules.

We have $\oplus\{\ell : A'_\ell\}_{\ell\in J} \le \oplus\{\ell : A_\ell\}_{\ell\in L}$. By subtyping, we have $A'_\ell \le A_\ell$ for $\ell \in J$, $J \subseteq L$. Apply I.H. to E' to get E''. For any fresh $f : A'_\ell$ we have $E'' = \Psi, \Delta, f : A'_\ell \vdash [f : A'_\ell/c : A_\ell]Q :: (d : D)$ for $\ell \in J$. We then apply $\oplus L$ to E'' to get $\Psi; \Delta, f : \oplus\{\ell : A'_l\}_{\ell\in J} \vdash \mathsf{case}\ f\ l \Rightarrow [f : A'_\ell/c : A_\ell]Q :: (d : D)$. By our substitution rules, this is equivalent to $\Psi; \Delta, f : \oplus\{\ell : A'_l\}_{\ell\in J} \vdash [f : \oplus\{\ell : A'_\ell\}/c : \oplus\{\ell : A_\ell\}]\mathsf{case}\ c\ l \Rightarrow Q :: (d : D)$.

Case $\exists L$.

$$\frac{E' = \Psi, n{:}\tau\ ;\ \Delta, c : A \vdash Q :: (d : D)}{E = \Psi\ ;\ \Delta, c : \exists n{:}\tau.\,A \vdash n \leftarrow \mathsf{recv}\, c\ ;\ Q :: (d : D)}\ \exists L$$

We have $\exists n : \tau'.A' \le \exists n : \tau.A$. By subtyping, we get that $\tau' \le \tau$ and $A' \le A$. Apply I.H. to E' to get E''. For any fresh $f : A'$ we have $E'' = \Psi, n : \tau; \Delta, f : A' \vdash [f : A'/c : A]Q :: (d : D)$. By construction, $|E'| = |E''|$, so we can apply I.H. to E'' to get E'''. For any fresh $m : \tau'$ we have $E''' = \Psi, m : \tau'; \Delta, f' : A' \vdash [m : \tau'/n : \tau][f : A'/c : A]Q :: (d : D)$. We now apply $\exists L$ to E''' to get $\Psi; \Delta, f : \exists m : \tau'.A' \vdash m \leftarrow \mathsf{recv}\ f; [m : \tau'/n : \tau][f : A'/c : A]Q :: (d : D)$. By our substitution rules, this is equivalent to $\Psi; \Delta, f : \exists m : \tau'.A' \vdash [f : \exists n : \tau'.A'/c : \exists n : \tau.A]n \leftarrow \mathsf{recv}\ c;\ Q :: (d : D)$.

Case $\multimap$L.

$$\frac{E' = \Psi\ ;\ \Delta, c : B \vdash Q :: (d : D) \quad A \sim A'}{E = \Psi\ ;\ \Delta, a : A', c : A \multimap B \vdash \mathsf{send}\, c\ \langle A \Leftarrow A'\rangle^{\rho}\ a\ ;\ Q :: (d : D)}\ \multimap L$$

Subcase principal. We have $A'' \multimap B \le A \multimap B'$. By subtyping, we get $A \le A''$ and $B' \le B$. Apply I.H. to E' to get E''. For any fresh $f : B'$ we have $E'' = \Psi; \Delta, f : B' \vdash [f : B'/c : B]Q :: (d : D)$. Now apply $\multimap$L to E'' to get $\Psi; \Delta, a : A', f : A \multimap B' \vdash \mathsf{send}\ f\ \langle A \Leftarrow A'\rangle^{\rho}\ a; [f : B'/c : B]Q :: (d : D)$. We then update the cast as follows: $\Psi; \Delta, a : A', f : A \multimap B' \vdash \mathsf{send}\ f\ \langle A'' \Leftarrow$

$$
\begin{array}{lll}
\text{cut} & : & \mathsf{proc}(c, x{:}A \leftarrow \langle A \Leftarrow A'\rangle^{\rho}\ P\ ;\ Q) \\
& & \longrightarrow \mathsf{proc}(a, [a : A'/x : A']P), \mathsf{proc}(c, [a : A'/x : A]Q) \quad \textit{(a fresh)} \\
\text{plus\_s} & : & \mathsf{proc}(c, c.k\ ;\ P) \\
& & \longrightarrow \mathsf{proc}(c', [c' : A_k/c : A_k]P), \mathsf{msg}(c : A_k, c.k\ ;\ c \leftarrow c') \quad \textit{(c' fresh)} \\
\text{plus\_r} & : & \mathsf{msg}(c : A_k, c.k\ ;\ c \leftarrow c'), \mathsf{proc}(d, \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L}) \\
& & \longrightarrow \mathsf{proc}(d, [c' : A_\ell/c : A_\ell]Q_k) \\
\text{with\_s} & : & \mathsf{proc}(d, c.k\ ;\ Q) \\
& & \longrightarrow \mathsf{msg}(c' : A_k, c.k\ ;\ c' \leftarrow c), \mathsf{proc}(d, [c' : A_k/c : A_k]Q) \quad (c'\ \text{fresh}) \\
\text{with\_r} & : & \mathsf{proc}(c, \mathsf{case}\ c\ (\ell \Rightarrow P_\ell)_{\ell \in L}), \mathsf{msg}(c' : A_k, c.k\ ;\ c' \leftarrow c) \\
& & \longrightarrow \mathsf{proc}(c', [c' : A_\ell/c : A_\ell]P_k) \\
\text{tensor\_s} & : & \mathsf{proc}(c, \mathsf{send}\ c\ (a : A')\ ;\ P) \longrightarrow \mathsf{proc}(c', [c' : B/c : B]P), \\
& & \mathsf{msg}(c : B, \mathsf{send}\ c\ \langle A \Leftarrow A'\rangle^{\rho} a\ ;\ c \leftarrow c') \quad \textit{(c' fresh)} \\
\text{tensor\_r} & : & \mathsf{msg}(c : B, \mathsf{send}\ c\ \langle A \Leftarrow A'\rangle^{\rho} a\ ;\ c \leftarrow c'), \mathsf{proc}(d, (x : A) \leftarrow \mathsf{recv}\ c\ ;\ Q) \\
& & \longrightarrow \mathsf{proc}(d, [c' : B/c : B][a : A'/x : A]Q) \\
\text{lolli\_s} & : & \mathsf{proc}(d, \mathsf{send}\ c\ (a : A')\ ;\ Q) \longrightarrow \mathsf{msg}(c' : B, \mathsf{send}\ c\ \langle A \Leftarrow A'\rangle^{\rho} a\ ; \\
& & c' \leftarrow c), \mathsf{proc}(d, [c' : B/c : B]Q) \quad (c'\ \text{fresh}) \\
\text{lolli\_r} & : & \mathsf{proc}(c, (x : A) \leftarrow \mathsf{recv}\ c\ ;\ P), \mathsf{msg}(c' : B, \mathsf{send}\ c\ \langle A \Leftarrow A'\rangle^{\rho} a\ ; \\
& & c' \leftarrow c) \longrightarrow \mathsf{proc}(c', [c' : B/c : B][a : A'/x : A]P) \\
\text{close} & : & \mathsf{proc}(c, \mathsf{close}\ c) \longrightarrow \mathsf{msg}(c : \mathbf{1}, \mathsf{close}\ c) \\
\text{wait} & : & \mathsf{msg}(c : \mathbf{1}, \mathsf{close}\ c), \mathsf{proc}(d, \mathsf{wait}\ c\ ;\ Q) \longrightarrow \mathsf{proc}(d, Q) \\
\text{exists\_s} & : & \mathsf{proc}(c, \mathsf{send}\ c\ (v : \tau')\ ;\ P) \longrightarrow \mathsf{proc}(c', [c' : [v/n]A/c : [v/n]A]P), \\
& & \mathsf{msg}(c : [v/n]A, \mathsf{send}\ c\ \langle \tau \Leftarrow \tau'\rangle^{\rho} v\ ;\ c \leftarrow c') \\
\text{exists\_r} & : & \mathsf{msg}(c : [v/n]A, \mathsf{send}\ c\ \langle \tau \Leftarrow \tau'\rangle^{\rho} v\ ;\ c \leftarrow c'), \\
& & \mathsf{proc}(d, (n : \tau) \leftarrow \mathsf{recv}\ c\ ;\ Q) \longrightarrow \mathsf{proc}(d, [c' : A/c : A][v : \tau'/n : \tau]Q) \\
\text{forall\_s} & : & \mathsf{proc}(d, \mathsf{send}\ c\ (v : \tau')\ ;\ Q) \longrightarrow \mathsf{msg}(c' : [v/n]A, \mathsf{send}\ c\ \langle \tau \Leftarrow \tau'\rangle^{\rho} v\ ; \\
& & c' \leftarrow c), \mathsf{proc}(d, [c' : [v/n]A/c : [v/n]A]Q) \\
\text{forall\_r} & : & \mathsf{proc}(c, (n : \tau) \leftarrow \mathsf{recv}\ c\ ;\ P), \mathsf{msg}(c' : [v/n]A, \mathsf{send}\ c\ \langle \tau \Leftarrow \tau'\rangle^{\rho} v\ ; \\
& & c' \leftarrow c) \longrightarrow \mathsf{proc}(c', [c' : A/c : A][v : \tau'/n : \tau]P) \\
\text{assert\_f} & : & \mathsf{proc}(c, \mathsf{assert}\ l\ \mathsf{False};\ Q) \longrightarrow \mathsf{abort}(l) \\
\text{assert\_s} & : & \mathsf{proc}(c, \mathsf{assert}\ l\ \mathsf{True};\ Q) \longrightarrow \mathsf{proc}(c, Q) \\
\text{id\_cast} & : & \mathsf{proc}(a, a \leftarrow \langle A \Leftarrow A'\rangle^{\rho} b), \mathcal{C} \longrightarrow [b : A'/a : A]\mathcal{C} \\
\text{val\_cast} & : & \mathsf{proc}(a, x \leftarrow \langle \tau' \Leftarrow \tau\rangle^{\rho}\ v;\ Q) \longrightarrow \mathsf{proc}(a, Q[v : \tau'/x : \tau])
\end{array}
$$

Fig. 17. Typed semantics.

$A'\rangle^{\rho}\ a; [f : B'/c : B]Q :: (d : D)$. By our substitution rules, this is equivalent to $\Psi; \Delta, a : A', f : A'' \multimap B' \vdash [f : A'' \multimap B'/c : A \multimap B]\mathsf{send}\ c\ \langle A \Leftarrow A'\rangle^{\rho}\ a\ ;\ Q :: (d : D)$.

Subcase side. Let $A'' \leq A'$. We can update the cast as follows: $\Psi\ ;\ \Delta, a : A'', c : A \multimap B \vdash \mathsf{send}\ c\ \langle A \Leftarrow A''\rangle^{\rho}\ a\ ;\ Q :: (d : D)$. By our substitution rules, this is equivalent to $[a : A''/a : A']\mathsf{send}\ c\ \langle A \Leftarrow A'\rangle^{\rho}\ a\ ;\ Q :: (d : D)$. □

We are now ready to prove preservation.

Theorem 5.5 *(Preservation). If* $\Delta_1 \Vdash \mathcal{C} : \Delta_2$ *and* $\mathcal{C} \longrightarrow \mathcal{C}'$ *then* $\Delta_1 \Vdash \mathcal{C}' : \Delta_2$.

Proof. In this proof, we make use of typed semantics (shown in Fig. 17) which are a version of the semantics presented previously where the message processes are augmented with channel types and casts as appropriate. We prove the theorem by examining each semantic case, inverting the typing rules (shown in Fig. 10) and invoking the subtype-substitution lemma.

Proving preservation for this system was unexpectedly more challenging than proving the safety and transparency theorems presented earlier in the section. The main challenge arose when defining and proving a substitution lemma for a session-type system with subtyping. This preservation proof concludes our metatheory section.

Case *id_cast*.

We have $\mathsf{proc}(a, a \leftarrow \langle A \Leftarrow A'\rangle^{\rho} b), \mathcal{C} \longrightarrow [b : A'/a : A]\mathcal{C}$ where $A' \leq A$. We know that $\mathsf{proc}(a, a \leftarrow b)$ is a client of exactly one process, which we can call $\mathsf{proc}(q, Q) \in \mathcal{C}$. This process must be well typed. Let $E' = \Psi; \Delta, x : A \vdash Q :: (q : C)$. Given $A' \leq A$, we then apply the subtype substitution lemma to E'. For a fresh $b : A'$ we then have $\Psi; \Delta, b : A' \vdash [b : A'/x : A]Q :: (q : C)$.

Case *val_cast*.

We have $\mathsf{proc}(a, x \leftarrow \langle \tau' \Leftarrow \tau \rangle^{\rho}\ v;\ Q) \longrightarrow \mathsf{proc}(a, Q[v : \tau'/x : \tau])$. We need to show that $\mathsf{proc}(a, [v : \tau'/x : \tau]Q)$ is well-typed. By inversion of *val_cast*, we know that $E' = \Psi, x : \tau; \Delta \vdash Q :: (c : C) \quad \tau \sim \tau'$ is well-typed. Given that $\tau' \leq \tau$, we can apply the subtype-substitution lemma to E'. For a fresh $v : \tau'$, we get that $\Psi; v : \tau'; \Delta \vdash [v : \tau'/x : \tau]Q :: (c : C)$.

Case *cut*.

We have $\mathsf{proc}(c, x{:}A \leftarrow \langle A \Leftarrow A' \rangle^{\rho} P\ ;\ Q) \longrightarrow \mathsf{proc}(a, [a : A'/x : A']P), \mathsf{proc}(c, [a : A'/x : A]Q)$ *(a fresh)*. We need to show that $\mathsf{proc}(a, [a : A'/x : A']P)$ is well-typed. By inversion on cut, we have that $E' = \Psi\ ; \Delta \vdash P :: (x : A')$ is well-typed. Given that $A' \leq A'$, we can apply subtype-substitution to E'. For a fresh $a : A'$, we have $\Psi, \Delta \vdash [a : A'/x : A']P :: (a : A')$. We also need to show that $\mathsf{proc}(c, [a : A'/x : A]Q)$ is well-typed. By inversion on cut, we have that $E'' = \Psi; x : A, \Delta' \vdash Q :: (c : C)$ is well-typed. Given $A' \leq A$, we can apply the subtype-substitution lemma to E''. For a fresh $a : A'$, we get $\Psi; a : A', \Delta' \vdash [a : A'/x : A]Q :: (c : C)$.

Case $\otimes$recv.

We have $\mathsf{msg}(c, \mathsf{send}\ c\ \langle A \Leftarrow A' \rangle^{\rho}\ a\ ;\ c \leftarrow c'), \mathsf{proc}(d, x \leftarrow \mathsf{recv}\ c\ ;\ Q) \longrightarrow \mathsf{proc}(d, [c' : B/c : B][a : A'/x : A]Q)$. We need to show that $\mathsf{proc}(d, [c' : B/c : B][a : A'/x : A]Q)$ is well-typed given that $A' \leq A$. By inversion of $\otimes L$ we know that $E' = \Psi; \Delta, x : A, c : B \vdash Q :: (d : D)$ is well-typed. We apply the subtype-substitution lemma to E' to get E''. For any fresh $y : A'$ we have $E'' = \Psi; \Delta, y : A', c : B \vdash [y : A'/x : A]Q :: (d : D)$. Given that $B \leq B$, we can apply subtype-substitution to E''. For a fresh $c' : B$, we have $\Psi; \Delta, y : A', c' : B \vdash [c' : B/c : B][y : A'/x : A]Q :: (d : D)$.

Case $\&$send.

We have $\mathsf{proc}(d, c.k\ ;\ Q) \longrightarrow \mathsf{msg}(c', c.k\ ;\ c' \leftarrow c), \mathsf{proc}(d, [c' : A_k/c : A_k]Q)$ (c' fresh). We need to show that $\mathsf{proc}(d, [c' : A_k/c : A_k]Q)$ are well-typed. By inversion on $\&L$, we have that $E' = \Psi; \Delta, c : A_k \vdash Q :: (d : D)$ is well-typed. Given $A_k \leq A_k$, we can apply subtype-substitution to E'. For a fresh $c' : A_k$, we get $\Psi; \Delta, c' : A_k \vdash [c' : A_k/c : A_k]Q :: (d : D)$. We also have that $c : \&\{\ell : A_\ell\}_{\ell \in L} \vdash c.k\ ;\ c' \leftarrow c :: (c' : A_k)$ which types the message.

Case $\forall$recv.

We have $\mathsf{proc}(c, x \leftarrow \mathsf{recv}\ c\ ;\ P), \mathsf{msg}(c', \mathsf{send}\ c\ \langle \tau \Leftarrow \tau' \rangle^{\rho} v\ ;\ c' \leftarrow c) \longrightarrow \mathsf{proc}(c', [c' : A/c : A][v : \tau'/n : \tau]P)$. We need to show that $\mathsf{proc}(c', [c' : A/c : A][v : \tau'/n : \tau]P)$ is well-typed. By inversion of $\forall R$, we have that $E' = \Psi, n{:}\tau\ ;\ \Delta \vdash P :: (c : A)$ is well-typed. Because we have $\tau' \leq \tau$, we can apply the subtype-substitution lemma to E' to get E''. For a fresh $v : \tau'$ we have $E'' = \Psi, v{:}\tau'\ ;\ \Delta \vdash [v : \tau'/n : \tau]P :: (c : A)$. Given $A \leq A$ we can apply the subtype-substitution lemma to E''. For fresh $c' : A$, we have $\Psi, v{:}\tau'\ ;\ \Delta \vdash [c' : A/c : A][v : \tau'/n : \tau]P :: (c : A)$. □

6. Related work

There is a rich body of work on higher-order contracts and the correctness of blame assignments calculus [17,18,16,19,15,20–22]. The contracts in these papers frequently make use of refinement or dependent types to express desired properties. Our system is able to encode refinement-based contracts and other contracts that use internal state (i.e., the parenthesis monitoring) in a concurrent language without using dependent types.

Dimoulas et al. [23] check contracts in parallel while executing the main computation, and synchronize when observable effects occur. Our system is similar in that the monitors execute in parallel with the main computation, but our monitors do not synchronize with the main computation. Further, our system is able to monitor contracts that maintain state. Swords et al. [24] classify contract-checking systems into five categories: eager [17,16], semi-eager [19], promise-based [23], concurrent, and finally-concurrent. Our monitors are concurrent, while those of Dimoulas et al. [23] are promise-based.

More closely related to our is the work by Disney et al. [25], where behavioral contracts that enforce temporal properties for modules are investigated. Our contracts (i.e., session types) enforce temporal properties as well; the session types specify the order in which messages are sent and received by the processes. Our contracts can also make use of internal state, as those of Disney et al. but our system is concurrent, while their system does not consider concurrency.

There is also a body of work on monitoring and blame assignment in a distributed setting. Our prior work [1] involves a session-typed system with high-order processes that sit on communication channels and monitor communication patterns. When an alarm is raised, the monitors are able to assign blame to the process that caused the contract violation. This work goes beyond our prior work by providing a mechanism to monitor contracts that cannot be expressed as communication patterns, for example, contracts that require the monitor to maintain state.

Work by Bocchi et al. [26] and Chen et al. [27] tackles the problem of monitoring multi-party session types. They assume a similar asynchronous message passing model as ours and their monitors monitor communication patterns. Our monitors are able to monitor more complex contracts such as refinements and contracts that require the monitor to maintain state. Using global types, their monitors can additionally enforce global properties such as deadlock freedom, which our monitors cannot. Our work supports higher-order processes, that is, processes that can spawn other processes and delegate communication to other processes, while their work does not.

Bocchi et al. [28] introduce an assertion based monitoring system for multi-party session types. Their system specifies a global assertion type that is projected to an assertion type for each endpoint in the system. Each process in the system is

then verified to match its endpoint assertion type. Similarly to our system, their system can model refinement contracts and contracts that maintain state. In contrast, their system is global in that it requires that every process be monitored, while our system allows both monitored and unmonitored processes.

The Whip system [29] addresses a similar problem, but does not use session types. They use a dependent type system to implement a contract monitoring system that can connect services written in different languages. Their system is also higher order, and allows processes that are monitored by Whip to interact with unmonitored processes. Another distinguishing feature of our monitors is that they are partial identity processes encoded in the same language as the processes to be monitored.

Recently, gradual typing for two-party session-type systems has been developed [30,31]. Even though it is a very different formalism from our contracts, the way untyped processes are gradually typed at run time resembles how we monitor type casts. Because of dynamic session types, their system has to keep track of the linear use of channels, which is not needed for our monitors.

Most recently, Melgratti and Padovani have developed chaperone contracts for higher-order session types [2]. Their work is based on a classic interpretation of session types, instead of an intuitionistic one like ours. Also, their work does not handle spawning or forwarding processes. While their contracts also inspect messages passed between processes, they use dependent types to model contracts which rely on the monitor making use of internal state (e.g., the parenthesis matching). They proved a blame soundness theorem stating that locally correct modules, which is a semantic categorization of whether a module satisfies a contract, cannot be blamed even in the presence of malicious parties. We did not prove a similarly general blame theorem; instead, we prove a somewhat standard safety theorem for cast-based contracts.

7. Conclusion

We have presented a novel approach for contract-checking for concurrent processes. Our model uses partial identity monitors which are written in the same language as the original processes and execute transparently. We define what it means to be a partial identity monitor and prove our characterization correct. We provide multiple examples of contracts we can monitor including ones that make use of the monitor's internal state, ones that make use of the idea of probabilistic result checking, and ones that cannot be expressed as dependent or refinement types. We translate contracts in the refinement fragment into monitors, and prove a safety theorem for that fragment.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The work is supported by the National Science Foundation under Grant NS1423168 and a Carnegie Mellon Presidential Fellowship.

References

[1] L. Jia, H. Gommerstadt, F. Pfenning, Monitors and blame assignment for higher-order session types, in: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16, 2016, pp. 582–594.
[2] H. Melgratti, L. Padovani, Chaperone contracts for higher-order sessions, Proc. ACM Program. Lang. 1 (ICFP) (2017) 35:1–35:29.
[3] H. Gommerstadt, L. Jia, F. Pfenning, Session-typed concurrent contracts, in: 27th European Symposium on Programming, ESOP 2018, Proceedings, 2018, pp. 771–798.
[4] F. Pfenning, D. Griffith, Polarized substructural session types, in: 18th International Conference on Foundations of Software Science and Computation Structures, FoSSaCS 2015, 2015, invited talk.
[5] P. Thiemann, V.T. Vasconcelos, Context-free session types, in: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, 2016, pp. 462–475.
[6] H. Wasserman, M. Blum, Software reliability via run-time result-checking, J. ACM 44 (6) (1997) 826–849.
[7] L. Caires, F. Pfenning, Session types as intuitionistic linear propositions, in: 21st International Conference on Concurrency Theory, CONCUR 2010, 2010.
[8] B. Toninho, A logical foundation for session-based concurrent computation, Ph.D. thesis, Carnegie Mellon University and New University of Lisbon, 2015.
[9] L. Caires, F. Pfenning, B. Toninho, Linear logic propositions as session types, Math. Struct. Comput. Sci. 26 (3) (2016) 367–423, Special Issue on Behavioural Types.
[10] I. Cervesato, A. Scedrov, Relating state-based and process-based concurrency through linear logic, Inf. Comput. 207 (10) (2009) 1044–1077.
[11] K. Honda, Types for dyadic interaction, in: 4th International Conference on Concurrency Theory, CONCUR 1993, 1993.
[12] S.J. Gay, M. Hole, Subtyping for session types in the π-calculus, Acta Inform. 42 (2–3) (2005) 191–225.
[13] S. Balzer, F. Pfenning, Manifest sharing with session types, Proc. ACM Program. Lang. 1 (ICFP) (2017) 37:1–37:29.
[14] B. Toninho, L. Caires, F. Pfenning, Higher-order processes, functions, and sessions: a monadic integration, in: 22nd European Symposium on Programming, ESOP 2013, 2013.
[15] P. Wadler, R.B. Findler, Well-typed programs can't be blamed, in: 18th European Symposium on Programming Languages and Systems, ESOP 2009, 2009.
[16] C. Dimoulas, R.B. Findler, C. Flanagan, M. Felleisen, Correct blame for contracts: no more scapegoating, in: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '11, 2011, pp. 215–226.

[17] R.B. Findler, M. Felleisen, Contracts for higher-order functions, in: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming, ICFP '02, 2002, pp. 48–59.
[18] C. Dimoulas, S.T. Hochstadt, M. Felleisen, Complete monitors for behavioral contracts, in: 21st European Conference on Programming Languages and Systems, ESOP 2012, 2012.
[19] C. Dimoulas, M. Felleisen, On contract satisfaction in a higher-order world, ACM Trans. Program. Lang. Syst. 33 (5) (2011) 1–29.
[20] P. Wadler, A complement to blame, in: 1st Summit on Advances in Programming Languages, SNAPL 2015, 2015.
[21] M. Keil, P. Thiemann, Blame assignment for higher-order contracts with intersection and union, in: 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, 2015.
[22] A. Ahmed, R.B. Findler, J.G. Siek, P. Wadler, Blame for all, in: 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, 2011.
[23] C. Dimoulas, R. Pucella, M. Felleisen, Future contracts, in: Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP '09, 2009, pp. 195–206.
[24] C. Swords, A. Sabry, T.-H. Sam, An extended account of contract monitoring strategies as patterns of communication, J. Funct. Program. 28 (2018) e4.
[25] T. Disney, C. Flanagan, J. McCarthy, Temporal higher-order contracts, in: 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011, 2011.
[26] L. Bocchi, T.-C. Chen, R. Demangeon, K. Honda, N. Yoshida, Monitoring networks through multiparty session types, in: Formal Techniques for Distributed Systems, FMOODS 2013, 2013.
[27] T.-C. Chen, L. Bocchi, P.-M. Deniélou, K. Honda, N. Yoshida, Asynchronous distributed monitoring for multiparty session enforcement, in: 6th International Symposium on Trustworthy Global Computing, TGC 2011, 2012, pp. 25–45.
[28] L. Bocchi, K. Honda, E. Tuosto, N. Yoshida, A theory of design-by-contract for distributed multiparty interactions, in: Proceedings of the 21st International Conference on Concurrency Theory, CONCUR 2010, in: LNCS, vol. 6269, Springer, 2010, pp. 162–176.
[29] L. Waye, S. Chong, C. Dimoulas, Whip: higher-order contracts for modern services, Proc. ACM Program. Lang. 1 (ICFP) (2017) 36:1–36:28.
[30] P. Thiemann, Session types with gradual typing, in: 9th International Symposium on Trustworthy Global Computing, TGC 2014, 2014.
[31] A. Igarashi, P. Thiemann, V.T. Vasconcelos, P. Wadler, Gradual session types, Proc. ACM Program. Lang. 1 (ICFP) (2017) 38:1–38:28.