Session-Typed Concurrent Contracts

Hannah Gommerstadt, Limin Jia, and Frank Pfenning

Carnegie Mellon University, Pittsburgh PA
{hgommers, fp}@cs.cmu.edu, liminjia@cmu.edu

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 are well-typed.

Keywords: contracts, session types, monitors

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 [15, 17]. 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. 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 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 is useful because we can show 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 the contracts cannot be expressed as types.

The main contributions of this paper are:

- A novel approach to contract checking via partial-identity monitors
- A method for verifying that monitors are partial identities, and a proof that the method is correct
- Examples showing the breadth of 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 review the background on session types in Section 2. Next, we show a range of example contracts 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. Due to space constraints, we only present the key theorems. Detailed proofs can be found in our companion technical report [12].

2 Session Types

Session types prescribe the communication behavior of message-passing concurrent processes. We approach them here via their foundation in intuitionistic linear logic [4, 22, 5]. 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.

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

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 $x \leftarrow P \,;\, Q$ 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.

In order to define the operational semantics rigorously, we use *multiset rewriting* [6]. 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). All the channels c provided by processes and messages in a configuration must be distinct.

A cut 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.

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

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*.

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 labelled 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 (which would violate session fidelity). 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.

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

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. 6.

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

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.

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

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

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

Receiving a channel (written as a linear implication $A \multimap B$) works symmetrically. Below are the semantics and the typing is shown in Figure 6.

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

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.

The proof of an existential quantifier contains a witness term, whose value is what is sent. 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 context 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.

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

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

The situation for universal quantification is symmetric. The semantics are given below and the typing is shown in Figure 6.

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

Processes may also make internal transitions while computing ordinary values, which we don't fully specify here. Such a transition would have the form

$$\mathsf{proc}(c, P[e]) \longrightarrow \mathsf{proc}(c, P[e']) \quad \text{if} \quad e \mapsto e'$$

where $P[e]$ would denote a process with an ordinary value expression in evaluation position and $e \mapsto e'$ would represent a step of computation.

Shifts For the purpose of monitoring, it is important to track the direction of communication. To make this explicit, we *polarize* the syntax and use *shifts* to change the direction of communication (for more detail, see prior work [18]).

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

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 Figure 6.

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

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

Recursive types Practical programming with session types requires them to be recursive, and processes using them also must allow recursion. For example, lists with elements of type int can be defined as the purely positive type list^+.

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

A provider of type $c : \mathsf{list}$ is required to send a sequence such as $\mathsf{cons}{\cdot}v_1{\cdot}\mathsf{cons}{\cdot}v_2 \cdots$ where each v_i is an integer. If it is finite, it must be terminated with $\mathsf{nil} \cdot \mathsf{end}$. In the form of a grammer, we could write

$$\mathit{From} ::= \mathsf{cons} \cdot v \cdot \mathit{From} \mid \mathsf{nil} \cdot \mathsf{end}$$

A second example is a multiset (bag) of integers, where the interface allows inserting and removing elements, and testing if it is empty. If the bag is empty when tested, the provider terminates after responding with the empty label.

$$\begin{aligned}\mathsf{bag}^- = \&\{\; & \mathsf{insert} : \forall n{:}\mathsf{int}.\, \mathsf{bag}^-,\; \mathsf{remove} : \forall n{:}\mathsf{int}.\, \mathsf{bag}^-, \\ & \mathsf{is\_empty} : \uparrow \oplus\{\mathsf{empty} : \mathbf{1}, \mathsf{nonempty} : \downarrow \mathsf{bag}^-\} \;\}\end{aligned}$$

The protocol now describes the following grammar of exchanged messages, where *To* goes to the provider, *From* comes from the provider, and v stands for integers.

$$\begin{aligned}\mathit{To} \quad &::= \mathsf{insert} \cdot v \cdot \mathit{To} \mid \mathsf{remove} \cdot v \cdot \mathit{To} \mid \mathsf{is\_empty} \cdot \mathsf{shift} \cdot \mathit{From} \\ \mathit{From} &::= \mathsf{empty} \cdot \mathsf{end} \mid \mathsf{nonempty} \cdot \mathsf{shift} \cdot \mathit{To}\end{aligned}$$

For these protocols to be realized in this form and support rich subtyping and refinement types without change of protocol, 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* [11], 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 2017 [3]) is to use an explicit definition just as we stated, for example, list and bag, and consider the left-hand side *equal* to the right-hand side in our discourse. In typing, this works without a hitch. When we consider subtyping explicitly, we need to make sure we view inference systems on types as being defined *co-inductively*. Since a co-inductively defined judgment essentially expresses the absence of a counterexample, this is exactly what we need for the operational properties like progress, preservation, or absence of blame. We therefore adopt this view.

Recursive processes In addition to recursively defined types, we also need recursively defined processes. We follow the general approach of Toninho et al [23] 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.

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

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

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

which we check with the rule

$$\frac{(\Psi \vdash e_i : \tau_i)_{i\in\{1,\ldots,k\}} \quad \Delta' = (d_1{:}A_1, \ldots, d_m{:}A_m) \quad \Psi\ ;\ \Delta, c : A \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, \Delta' \vdash c \leftarrow p\, e_1 \ldots e_k \leftarrow d_1, \ldots, d_m\ ;\ Q :: (d : D)}\ \mathsf{pdef}$$

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'. Depending on how forwarding is implemented, however, it may be much more efficient [13].

Stopping computation Finally, in order to be able to successfully monitor computation, we need the capability to stop the computation. We add an $\mathsf{abort}\ l$ construct that aborts on a particular label. We also add assert blocks to check conditions on observable values. The semantics are given below and the typing is in Figure 6.

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

Progress and preservation were proven for the above system, with the exception of the abort and assert rules, in prior work [18]. The additional proof cases do not change the proof significantly.

3 Contract Examples

In this section, we present monitoring processes that can enforce a variety of contracts. The examples will mainly use lists as defined in the previous section. Our monitors are transparent, that is, they do not change the computation. We accomplish this by making them act as partial identities (described in more detail in Section 4). Therefore, any monitor that enforces a contract on a list must peel off each layer of the type one 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).

Refinement The simplest kind of monitoring process we can write is one that models a refinement of an integer type; for example, a process that checks whether every element in the list is positive. This is a recursive process that receives the head of the list from channel b, checks whether it is positive (if yes, it continues to the next value, if not it aborts), and then sends the value along to reconstruct the monitored list a. We show three refinement monitors in Figure 1. The process `pos` implements the refinement mentioned above.

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

```
empty : {list ← list}
a ← empty ← b =
  case b of
  | nil ⇒ wait b ;
    a.nil ; close a
  | cons ⇒ abort^ρ; ;
```

```
nempty : {list ← list}
a ← nempty ← b =
  case b of
  | nil ⇒ abort^ρ
  | cons ⇒ a.cons ;
    x ← recv b ;
    send a x ; a ← b; ;
```

Fig. 1. Refinement examples

Our monitors can also exploit information that is contained in the labels in the external and internal choices. The `empty` process checks whether the list b is empty and aborts if b sends the label `cons`. Similarly, the `nempty` monitor checks whether the list b is not empty and aborts if b sends the label `nil`. These two monitors can then be used by a process that zips two lists and aborts if they are of different lengths. These two monitors enforce the refinements $\{\texttt{nil}\} \subseteq \{\texttt{nil}, \texttt{cons}\}$ and $\{\texttt{cons}\} \subseteq \{\texttt{nil}, \texttt{cons}\}$. We discuss how to generate monitors from refinement types in more detail in Section 5.

Monitors with internal state We now move beyond refinement contracts, and model contracts that have to maintain some internal state (Figure 2).

We first present a monitor that checks whether the given list is sorted in ascending order (`ascending`). The monitor's state consists of a lower bound on the subsequent elements in the list. This value has an option type, which 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

```
ascending : option int → {list ← list}; ;
m ← ascending 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 (Some x) ← n
    | Some a ⇒ assert (x ≥ a)^ρ ;
      m.cons ; send m x ;
      m ← ascending (Some x) ← n; ;
```

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

Fig. 2. Monitors using internal state

failure. Note that the output list m is the same as the input list n because every element that we examine is then passed along unchanged to m.

We can use the `ascending` monitor to verify that the output list of a sorting procedure is in sorted order. To take the example one step further, we can verify that the elements in the output list are in fact a permutation of the elements in the input list of the sorting procedure as follows. Using a reasonable hash function, we hash each element as it is sent to the sorting procedure. Our monitor then keeps track of a running total of the sum of the hashes, and as elements are received from the sorting procedure, it computes their hash and subtracts it from the total. After all of the elements are received, we check that the total is 0 – if it is, with high probability, the two lists are permutations of each other. This example is an instance of *result checking*, inspired by Wasserman and Blum [26]. The monitor encoding is straightforward and omitted from the paper.

Our next example `match` validates whether a set of right and left parentheses match. The monitor can use 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 parenthesis because adding all of the integer representations does not yield 0. 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 [21].

Mapper Finally, we can also define monitors that check higher-order contracts, such as a contract for a mapping function (Figure 3). Consider the mapper which takes an integer and doubles it, and a function `map` that applies this mapper to

```
mapper_tp : {&{done : 1 ; next : ∀n : int.∃n : int.mapper_tp}}
m ← mapper =
  case m of
  | done ⇒ close m
  | next ⇒ x ← recv m ; send m (2 ∗ x) ; m ← mapper
map : {list ← mapper_tp ; list}
k ← map ← m l =
  case l of
  | nil ⇒ m.done ; k.nil ; wait l ; close k
  | cons ⇒ m′ ← mapper_mon ← m;   //run monitor
    x ← recv l ; send m′ x ; y ← recv m′ ; k.cons ; send k y ; k ← map m′ l;;
mapper_mon : {mapper_tp ← mapper_tp}
n ← mapper_mon ← m =
  case n of
  | done ⇒ m.done ; wait m ; close n
  | next ⇒ x ← recv n ; assert(x > 0)^ρ1   //checks precondition
    m.next ; send m x ; y ← recv m ; assert(y > x)^ρ2   //checks postcondition
    send n y ; n ← mapper_mon ← m
```

Fig. 3. Higher-Order monitor

a list of integers to produce a new list of integers. We can see that any integer that the mapper has produced will be strictly larger than the original integer, assuming the original integer is positive. In order to monitor this contract, it makes sense to impose a contract on the mapper itself. This `mapper_mon` process enforces both the precondition, that the original integer is positive, and the postcondition, that the resulting integer is greater than the original. We can now run the monitor on the mapper, in the `map` process, before applying the mapper to the list l.

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, they 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.

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 to which you send the arguments and which sends back the return value *along the same channel*. 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 constraint is that *contracts are expressed as session-typed processes*, just like functional contracts should be expressed within the functional language, or object contracts within the object oriented language, etc.

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. All the contracts in this paper can be verified to be partial identities under our definition.

4.1 Buffering Values

As a first simple example let's take a process that receives one positive integer n and factors it into two integers p and q that are sent back where $p \leq q$. The part of the specification that is *not* enforced is that if n is not prime, p and q should be proper factors, but we at least enforce that all numbers are positive

and $n = p * q$. We are being very particular here, for the purpose of exposition, marking the place where the direction of communication changes with a shift ($\uparrow$). Since a minimal number of shifts can be inferred during elaboration of the syntax [18], we suppress it in most examples.

$$
\begin{array}{l}
\mathsf{factor\_t} = \forall n{:}\mathsf{int}.\uparrow \exists p{:}\mathsf{int}.\, \exists q{:}\mathsf{int}.\, \mathbf{1} \\
\mathsf{factor\_monitor} : \{\mathsf{factor\_t} \leftarrow \mathsf{factor\_t}\} \\
c \leftarrow \mathsf{factor\_monitor} \leftarrow d = \\
\quad n \leftarrow \mathsf{recv}\ c\ ;\ \mathsf{assert}\ (n > 0)^{\rho_1}\ ;\ \mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ \mathsf{send}\ d\ n\ ;\ \mathsf{send}\ d\ \mathsf{shift}\ ; \\
\quad p \leftarrow \mathsf{recv}\ d\ ;\ \mathsf{assert}(p > 0)^{\rho_2}\ ;\ q \leftarrow \mathsf{recv}\ d\ ;\ \mathsf{assert}(q > 0)^{\rho_3}\ ;\ \mathsf{assert}(p \leq q)^{\rho_4}\ ; \\
\quad \mathsf{assert}(n = p * q)^{\rho_5}\ ;\ \mathsf{send}\ c\ p\ ;\ \mathsf{send}\ c\ q\ ;\ c \leftarrow d
\end{array}
$$

This is a one-time interaction (the session type $\mathsf{factor\_t}$ is not recursive), so the monitor terminates. It terminates here by forwarding, but we could equally well have replaced it by its identity-expanded version at type $\mathbf{1}$, which is $\mathsf{wait}\ d\ ;\ \mathsf{close}\ c$.

The contract could be invoked by the provider or by the client. Let's consider how a provider factor might invoke it:

$$
\begin{array}{l}
\mathsf{factor} : \{\mathsf{factor\_t}\} \\
c \leftarrow \mathsf{factor} = \\
\quad c' \leftarrow \mathsf{factor\_raw}\ ;\ c' \leftarrow \mathsf{factor\_monitor} \leftarrow c'\ ;\ c \leftarrow c'
\end{array}
$$

To check that $\mathsf{factor\_monitor}$ 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 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. The context Ψ declares types for local variables.

A monitor, at the top level, is defined with

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

where 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

$$
\begin{array}{lllll}
m ::= l_k & \text{labels} & \oplus, \& & & \\
\quad \mid\ c & \text{channels} & \otimes, \multimap & \mid\ n & \text{value variables}\ \exists, \forall \\
\quad \mid\ \mathsf{end} & \text{close} & \mathbf{1} & \mid\ \mathsf{shift} & \text{shifts}\ \uparrow, \downarrow
\end{array}
$$

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

When a process P receives a message, we add it to the end of the queue q. We also need to add it to Ψ context, marked as *unrestricted* (non-linear) 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.

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

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 simple 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.

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

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 Rule summary

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

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

So far, it is given by the following rules

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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:

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

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

The second premise (that is, the continuation of the monitor) remains the monitor, while the first premise corresponds to a freshly spawned local progress 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.

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

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.

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

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.

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

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

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.

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. This is important, because monitors for recursive types usually have a recursive structure. An illustration of this can be seen in pos in Figure 1.

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. 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.

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

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

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 these in the following table. In each case, c is the channel along with the message is transmitted, and c' is the continuation channel.

Message to client of c		Message to provider of c	
$\mathsf{msg}^+(c, c.k \; ; \; c \leftarrow c')$	$(\oplus)$	$\mathsf{msg}^-(c', c.k \; ; \; c' \leftarrow c)$	$(\&)$
$\mathsf{msg}^+(c, \mathsf{send}\ c\ d \; ; \; c \leftarrow c')$	$(\otimes)$	$\mathsf{msg}^-(c', \mathsf{send}\ c\ d \; ; \; c' \leftarrow c)$	$(\multimap)$
$\mathsf{msg}^+(c, \mathsf{close}\ c)$	$(\mathbf{1})$		
$\mathsf{msg}^+(c, \mathsf{send}\ c\ v \; ; \; c \leftarrow c')$	$(\exists)$	$\mathsf{msg}^-(c', \mathsf{send}\ c\ v \; ; \; c' \leftarrow c)$	$(\forall)$
$\mathsf{msg}^+(c, \mathsf{send}\ c\ \mathsf{shift} \; ; \; c \leftarrow c')$	$(\downarrow)$	$\mathsf{msg}^-(c', \mathsf{send}\ c\ \mathsf{shift} \; ; \; c' \leftarrow c)$	$(\uparrow)$

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.

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 complex, so we will walk the reader through a sequence of generalizations that account for various phenomena.

The $\oplus, \&$ fragment. For this fragment, we have no value variables, nor are we passing 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:

Message to client of c			Message to provider of c		
$\langle\!\langle \mathsf{msg}^+(c, c.k\ ;\ c \leftarrow c') \rangle\!\rangle$	$= k$	$(\oplus)$	$\langle\!\langle \mathsf{msg}^-(c', c.k\ ;\ c' \leftarrow c) \rangle\!\rangle$	$= k$	$(\&)$
$\langle\!\langle \mathsf{msg}^+(c, \mathsf{send}\ c\ d\ ;\ c \leftarrow c') \rangle\!\rangle$	$= d$	$(\otimes)$	$\langle\!\langle \mathsf{msg}^-(c', \mathsf{send}\ c\ d\ ;\ c' \leftarrow c) \rangle\!\rangle$	$= d$	$(\multimap)$
$\langle\!\langle \mathsf{msg}^+(c, \mathsf{close}\ c) \rangle\!\rangle$	$= \mathsf{end}$	$(\mathbf{1})$			
$\langle\!\langle \mathsf{msg}^+(c, \mathsf{send}\ c\ v\ ;\ c \leftarrow c') \rangle\!\rangle$	$= v$	$(\exists)$	$\langle\!\langle \mathsf{msg}^-(c', \mathsf{send}\ c\ v\ ;\ c' \leftarrow c) \rangle\!\rangle$	$= v$	$(\forall)$
$\langle\!\langle \mathsf{msg}^+(c, \mathsf{send}\ c\ \mathsf{shift}\ ;\ c \leftarrow c') \rangle\!\rangle$	$= \mathsf{shift}$	$(\downarrow)$	$\langle\!\langle \mathsf{msg}^-(c', \mathsf{send}\ c\ \mathsf{shift}\ ;\ c' \leftarrow c) \rangle\!\rangle$	$= \mathsf{shift}$	$(\uparrow)$

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 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 $y{:}B^- \vdash \mathcal{E} \sim_M \mathsf{proc}(x, P) :: (x{:}xA)$.

Next, we can permit local state in the monitor (rules cut_1^+ and cut_1^-). The fact that neither of the two critical endpoints y and x, nor any (non-local) channel,s 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 cut_2^+ and cut_2^-) just comes down to the compositionally 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 cut_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 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:

Corollary 1. *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 2). 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" [25], 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 type B to type A, where ρ is a unique label for the cast. The cast for values is written as $(\langle \tau \Leftarrow \tau' \rangle^\rho)$. Here, the types τ' and τ are refinement types of the form $\{n{:}t \mid b\}$, where b is a boolean expression that expresses simple properties of the value n.

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

Adding casts to forwarding is expressive enough to encode a more general cast $\langle A \Leftarrow B \rangle^\rho P$. For instance, the process $x{:}A \leftarrow \langle A \Leftarrow B \rangle^\rho P \; ; Q_x$ can be encoded as: $y{:}B \leftarrow P; x{:}A \leftarrow \langle A \Leftarrow B \rangle^\rho \; y \; ; Q_x$.

One of the additional rules to type casts is shown below (both rules can be found in Figure 6). 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 omitted from the paper).

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

5.2 Translation to Monitors

At run time, casts are translated into monitoring processes. A cast $a \leftarrow \langle A \Leftarrow B \rangle^\rho \; b$ is 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.

Figure 4 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 a label of the monitor (i.e., the contract).

Note that this differs from blame labels for high-order functions, where the monitor carries two labels, one for the argument, and one for the body of the function. Here, the communication between processes is bi-directional. Though the blame is always triggered by processes sending messages to the monitor, our contracts may depend on a set of the values received so far, so it does not make sense to blame one party. Further, in the case of forwarding, the processes at either end of the channel are behaving according to the types (contracts) assigned to them, but the cast may forcefully connect two processes that have incompatible types. In this case, it is unfair to blame either one of the processes. Instead, we raise an alarm of the label of the failed contract.

The translation is defined inductively over the structure of the types. The tensor rule generates a process that first receives a channel (x) from the channel being monitored (b). It then spawns a new monitor (denoted by the @monitor keyword) to monitor channel x, making sure that it behaves as type A_1, and

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

$$\frac{}{\begin{array}{l}[\![\langle A_1 \multimap A_2 \Leftarrow B_1 \multimap B_2 \rangle^{\rho}]\!]_{a,b} = \\ x \leftarrow \mathsf{recv}\ a; \\ @\mathsf{monitor}\ y \leftarrow [\![\langle B_1 \Leftarrow A_1 \rangle^{\rho}]\!]_{y,x} \leftarrow x \\ \mathsf{send}\ b\ y; \\ [\![\langle A_2 \Leftarrow B_2 \rangle^{\rho}]\!]_{a,b}\end{array}}\ \multimap \qquad \frac{}{\begin{array}{l}[\![\langle A_1 \otimes A_2 \Leftarrow B_1 \otimes B_2 \rangle^{\rho}]\!]_{a,b} = \\ x \leftarrow \mathsf{recv}\ b; \\ @\mathsf{monitor}\ y \leftarrow [\![\langle A_1 \Leftarrow B_1 \rangle^{\rho}]\!]_{y,x} \leftarrow x \\ \mathsf{send}\ a\ y; \\ [\![\langle A_2 \Leftarrow B_2 \rangle^{\rho}]\!]_{a,b}\end{array}}\ \otimes$$

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

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

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

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

$$\frac{}{\begin{array}{l}[\![\langle \uparrow A \Leftarrow \uparrow B \rangle^{\rho}]\!]_{a,b} = \\ \mathsf{shift} \leftarrow \mathsf{recv}\ b; \\ \mathsf{send}\ a\ \mathsf{shift}\ ;\ [\![\langle A \Leftarrow B \rangle^{\rho}]\!]_{a,b}\end{array}}\ \uparrow \qquad \frac{}{\begin{array}{l}[\![\langle \downarrow A \Leftarrow \downarrow B \rangle^{\rho}]\!]_{a,b} = \\ \mathsf{shift} \leftarrow \mathsf{recv}\ a; \\ \mathsf{send}\ b\ \mathsf{shift}\ ;\ [\![\langle A \Leftarrow B \rangle^{\rho}]\!]_{a,b}\end{array}}\ \downarrow$$

Fig. 4. Cast translation

passes the new monitor's offering channel y to channel 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 exists rule generates a process that first receives a value from the channel b, then checks the boolean condition e to validate the contract. The forall rule is similar, except the argument position is contravariant, so the boolean expression e' is checked on the offering channel a. 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 $\oplus\{\ell : A_\ell\}_{\ell \in I}$.

For ease of explanation, we omit details for translating casts involving recursive types. Briefly, these casts are translated into recursive processes. For each pair of compatible recursive types A and B, we generate a unique monitor name

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

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

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

$$\frac{\mathsf{def}(A) \leq \mathsf{def}(B)}{A \leq B}\ \mathsf{def} \qquad \frac{\forall v{:}\tau, [v/x]b_1 \mapsto^* \mathsf{true} \text{ implies } [v/x]b_2 \mapsto^* \mathsf{true}}{\{x{:}\tau \mid b_1\} \leq \{x{:}\tau \mid b_2\}}\ \mathsf{refine}$$

Fig. 5. Subtyping

f and record its type $f : \{A \leftarrow B\}$ in a context Ψ. The translation algorithm needs to take additional arguments, including Ψ 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 $@\mathsf{monitor}\ y \leftarrow\ f\ \leftarrow x$.

5.3 Metatheory

We prove two formal properties of cast-based monitors: safety and transparency.

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 Figure 5. 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 Acay and Pfenning [1]. For recursive types, we directly examine their definitions. Because of these recursive types, our subtyping rules are co-inductively defined.

We prove a safety theorem (i.e., well-typed casts do not raise alarms) via the standard preservation theorem. The key is to show that the monitor process generated from the translation algorithm in Figure 4 is well-typed under a typing relation which guarantees that no abort state can be reached. We refer to the type system presented thus far in the paper as T, where monitors that may evaluate to abort can be typed. We define a stronger type system S which consists of the rules in T with the exception of the abort rule and we replace the assert rule with the $\mathsf{assert\_strong}$ rule. The new rule for assert, which semantically verifies

Both System T and S

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

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

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

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

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

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

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

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

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

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

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

System T only

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

System S only

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

Fig. 6. Typing process expressions

that the condition b is true using the fact that the refinements are stored in the context Ψ, is shown below. The two type systems are summarized in Figure 6.

Theorem 2 (Monitors are well-typed). *Let Ψ be the context containing the type bindings of all recursive processes.*

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

Proof. The proof is by induction over the monitor translation rules. For 2, we need to use the sub-typing relation to show that (1) for the internal and external choice cases, no branches that include abort are generated; and (2) for the forall and exists cases, the assert never fails (i.e., the assert_strong rule applies). □

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

Finally, we prove that monitors translated from casts are partial identify processes.

Theorem 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 in Section 4. 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.

6 Related Work

There is a rich body of work on higher-order contracts and the correctness of blame assignments in the context of the lambda calculus [10, 8, 7, 25, 24, 16, 2]. The contracts in these papers are mostly based on refinement or dependent types. Our contracts are more expressive than the above, and can encode refinement-based contracts. While our monitors are similar to reference monitors (such as those described by Schneider [19]), they have a few features that are not inherent to reference monitors such as the fact that our monitors are written in the target language. Our monitors are also able to monitor contracts in a higher-order setting by spawning a separate monitor for the sent/received channel.

Disney et al.'s [9] work, which investigates behavioral contracts that enforce temporal properties for modules, is closely related to our work. Our contracts (i.e., session types) also enforce temporal properties; 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.

Recently, gradual typing for two-party session-type systems has been developed [20, 14]. Even though this formalism is different 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 [17]. Their work is based on a classic interpretation of session types, instead of an intuitionistic one like ours, which means that they do not handle spawning or forwarding processes. While their contracts also inspect messages passed between processes, unlike ours, they cannot model contracts which rely on the monitor making use of internal state (e.g., the parenthesis matching). They proved a blame theorem relying on the notion of locally correct modules, which is a semantic categorization of whether a module satisfies the contract. We did not prove a general blame theorem; instead, we prove a somewhat standard safety theorem for cast-based contracts.

The Whip system [27] addresses a similar problem as our prior work [15], 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. While Whip can express dependent contacts, Whip cannot handle stateful contracts. Another distinguishing feature of our monitors is that they are partial identity processes encoded in the same language as the processes to be monitored.

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.

Acknowledgment

This research was supported in part by NSF grant CNS1423168 and a Carnegie Mellon University Presidential Fellowship.

References

1. Acay, C., Pfenning, F.: Intersections and unions of session types. In: Proceedings Eighth Workshop on Intersection Types and Related Systems, ITRS 2016, Porto, Portugal, 26th June 2016. pp. 4–19 (2016), https://dx.doi.org/10.4204/EPTCS.242.3
2. Ahmed, A., Findler, R.B., Siek, J.G., Wadler, P.: Blame for all. In: 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011) (2011), https://doi.acm.org/10.1145/1570506.1570507
3. Balzer, S., Pfenning, F.: Manifest sharing with session types. Proc. ACM Program. Lang. 1(ICFP), 37:1–37:29 (Aug 2017), https://doi.acm.org/10.1145/3110281
4. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: 21st International Conference on Concurrency Theory (CONCUR 2010) (2010), https://dx.doi.org/10.1007/978-3-642-15375-4_16
5. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Mathematical Structures in Computer Science 26(3), 367–423 (2016), special Issue on Behavioural Types.
6. Cervesato, I., Scedrov, A.: Relating state-based and process-based concurrency through linear logic. Information and Computation 207(10), 1044–1077 (2009), https://dx.doi.org/10.1016/j.ic.2008.11.006
7. Dimoulas, C., Findler, R.B., Flanagan, C., Felleisen, M.: Correct blame for contracts: No more scapegoating. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 215–226. POPL '11, ACM, New York, NY, USA (2011), https://doi.acm.org/10.1145/1926385.1926410
8. Dimoulas, C., Hochstadt, S.T., Felleisen, M.: Complete Monitors for Behavioral Contracts. In: 21st European Conference on Programming Languages and Systems (ESOP 2012) (2012), https://dx.doi.org/10.1007/978-3-642-28869-2_11
9. Disney, T., Flanagan, C., McCarthy, J.: Temporal higher-order contracts. In: 16th ACM SIGPLAN International Conference on Functional Programming (ICFP 2011) (2011), https://doi.acm.org/10.1145/2034773.2034800
10. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming. pp. 48–59. ICFP '02, ACM, New York, NY, USA (2002), https://doi.acm.org/10.1145/581478.581484
11. Gay, S.J., Hole, M.: Subtyping for session types in the π-calculus. Acta Informatica 42(2–3), 191–225 (2005), https://dx.doi.org/10.1007/s00236-005-0177-z
12. Gommerstadt, H., Jia, L., Pfenning, F.: Session-typed concurrent contracts. Tech. Rep. CMU-CyLab-17-004, CyLab, Carnegie Mellon University (Feb 2018)
13. Griffith, D.: Polarized Substructural Session Types. Ph.D. thesis, University of Illinois at Urbana-Champaign (Apr 2016)
14. Igarashi, A., Thiemann, P., Vasconcelos, V.T., Wadler, P.: Gradual session types. Proc. ACM Program. Lang. 1(ICFP), 38:1–38:28 (Aug 2017), https:/doi.acm.org/10.1145/3110282
15. Jia, L., Gommerstadt, H., Pfenning, F.: Monitors and blame assignment for higher-order session types. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 582–594. POPL '16, ACM, New York, NY, USA (2016), https://doi.acm.org/10.1145/2837614.2837662
16. Keil, M., Thiemann, P.: Blame assignment for higher-order contracts with intersection and union. In: 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015) (2015), https://doi.acm.org/10.1145/2784731.2784737

17. Melgratti, H., Padovani, L.: Chaperone contracts for higher-order sessions. Proc. ACM Program. Lang. 1(ICFP), 35:1–35:29 (Aug 2017), https://doi.acm.org/10.1145/3110279
18. Pfenning, F., Griffith, D.: Polarized substructural session types. In: 18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2015) (2015), https://doi.acm.org/10.1007/978-3-662-46678-0_1, invited talk.
19. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. 3(1), 30–50 (Feb 2000), https://doi.acm.org/10.1145/353323.353382
20. Thiemann, P.: Session Types with Gradual Typing. In: 9th International Symposium on Trustworthy Global Computing (TGC 2014) (2014), https://dx.doi.org/10.1007/978-3-662-45917-1_10
21. Thiemann, P., Vasconcelos, V.T.: Context-free session types. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming. pp. 462–475. ICFP 2016, ACM, New York, NY, USA (2016), https:/acm.doi.org/10.4230/LIPIcs.ECOOP.2016.9
22. Toninho, B.: A Logical Foundation for Session-based Concurrent Computation. Ph.D. thesis, Carnegie Mellon University and New University of Lisbon (2015)
23. Toninho, B., Caires, L., Pfenning, F.: Higher-order processes, functions, and sessions: A monadic integration. In: 22nd European Symposium on Programming (ESOP 2013) (2013), https://dx.doi.org/10.1007/978-3-642-37036-6_20
24. Wadler, P.: A Complement to Blame. In: 1st Summit on Advances in Programming Languages (SNAPL 2015) (2015), https://doi.acm.org/10.4230/LIPIcs.SNAPL.2015.309
25. Wadler, P., Findler, R.B.: Well-Typed Programs Can't Be Blamed. In: 18th European Symposium on Programming Languages and Systems (ESOP 2009) (2009), https://dx.doi.org/10.1007/978-3-642-00590-9_1
26. Wasserman, H., Blum, M.: Software reliability via run-time result-checking. J. ACM 44(6), 826–849 (Nov 1997), https://doi.acm.org/10.1145/268999.269003
27. Waye, L., Chong, S., Dimoulas, C.: Whip: Higher-order contracts for modern services. Proc. ACM Program. Lang. 1(ICFP), 36:1–36:28 (Aug 2017), https://doi.acm.org/10.1145/3110280